KAERI/AR-565/00

소프트웨어 신뢰도의 정량적 평가 기법에 대한 고유 현안 분석

A technical survey on issues of the quantitative evaluation of software reliability

한 국 원 자 력 연 구 소

Please be aware that all of the Missing Pages in this document were originally blank pages

제 출 문

한국원자력연구소장 귀하

본 보고서를 2000 연도 "차세대원자로 설계관련 요소기술 개발" 과제의 기술현황분석보고서로 제출합니다.

2000. 4. 3.

주 저 자 : 박진균

공 저 자 : 성태용
엄홍섭
정환성
박진희
강현국
이기영
박종균

요 약 문

최근, 시스템의 정확성, 보수성(maintainability), 운전신뢰도 향상 및 복잡성 등과 같은 기존 아날로그 계통설계의 한계성이 대두됨에 따라 울진 5/6호기 등과 같이 신규 건설중인 원자력 발전소에 대해 디지털 계측제어 시스템이 도입되고 있다. 또한 가동중인 원자력 발전소의 경우도 기존 아날로그 계측제어 시스템의 성능저하 및 노후화가 발생하지만 이를 교체할 만한 예비품 또는 교체부품의 확보가 어렵기 때문에 디지털 계측제어 시스템으로의 교체가 점차 증가하는 추세이다.

기존 아날로그 계측제어 시스템에 비해 디지털 계측제어 시스템이 가지는 가장 큰 특징 중 하나는 소프트웨어를 포함한 컴퓨터 기술의 도입이다. 그러나 소프트웨어는 하드웨어와는 달리 신뢰도를 평가하는데 있어서 몇 가지 문제점 (aging이 없고, non-linearity 특성을 가지며 자원을 공유하는 점 등)으로 인해 원자력 발전소의 안전성 평가(PSA)를 위해 반드시 필요한 소프트웨어의 고장 확률 (또는 신뢰도 자료)를 얻기가 힘들다.

따라서 본 연구에서는 PSA 수행에 있어서 반드시 필요한 소프트웨어의 고장 확률을 구할 수 있는 방법론 개발을 위한 기초 연구로서, 다음과 같은 연구를 수행하였다.

- ㅇ 기존에 소프트웨어 신뢰도 공학 분야에서 제안된 많은 연구 결과들을 검토한 뒤, 이들을 각각의 특징에 따라 이들을 분류 (총 13개의 category)
- ㅇ 각 category에 대한 장·단점을 분석하고, 원자력 발전소의 디지털 계측제어 시스템에 적용될 수 있는지를 판단

이러한 연구의 결과로서, 원자력 발전소의 디지털 계측제어 시스템에 포함된 소프트웨어의 고장 확률을 직접적으로 구할 수 있는 방법론은 아직 없는 것으로 판단된다. 따라서 기존에 제안된 여러 가지 방법들 중 가장 적합하다고 판단되는 몇가지를 선택하여 이들의 조합을 통해 소프트웨어의 고장 확률을 구하는 방법에 대한 연구가 추가적으로 수행되어야 할 것으로 판단된다.

Summary

Recently, newly being developed nuclear power plants including Ulchin 5/6 accept digital instrumentation and control (I&C) systems because the limitations such as correctness, maintainability, enhancement of the operational reliability and complexity of conventional analog systems arise. In addition, in the case of currently being operated nuclear power plants, the tendency of adopting digital I&C systems is increasing because it is difficult to prepare/establish spear parts of installed analog I&C systems.

One of the most significant difference between conventional analog and digital I&C systems is using computer systems in which various kinds of software were included. However, in contrast to hardware, it is very difficult to obtain software reliability data (or software failure rate) that are requisite for the PSA of nuclear power plants because there are some problems in evaluating software reliability (i.e., no aging, non-linearity and resource sharing).

Therefore, as the preliminary phase, the following researches are performed to determine appropriate methodology for evaluating software reliability.

- ㅇ Categorizing methodologies that have been proposed for evaluating software reliability in the software reliability engineering field.
- ㅇ Analyzing strong · weak points of each category to determine the most appropriate methodologies that can be applied for the digital I&C systems

As the results, methodologies that can be directly applied for the evaluation of the software reliability for the digital I&C systems are not exist. Thus additional researches to combine the most appropriate methodologies/techniques from existing ones would be needed to evaluate the software reliability.

목 차

표 목 차

그 림 목 차

제 1 장 서 론

최근 들어 일반 산업계의 컴퓨터 시스템 사용이 보편화되고, 원자력 산업계에서도 시스템의 정확성, 보수성(maintainability), 운전신뢰도 향상 및 복잡성 등과 같은 기존 아날로그 계통설계의 한계성이 대두됨에 따라[1, 2], 울진 5/6호기 등과 같이 신규 건설중인 원자력 발전소에 대해 디지털 계측제어 시스템이 도입되고 있다. 또한 가동중인 원자력 발전소의 경우 기존 아날로그 계측제어 시스템의 성능저하 및 노후화가 발생하지만 이를 교체할 만한 예비품 또는 교체부품의 확보가 어렵기 때문에 디지털 계측제어 시스템으로의 교체가 점차 증가하는 추세이다. 기존 아날로그 계측제어 시스템에 비해 디지털 계측제어 시스템이 가지는 가장 큰 특징 중 하나는 소프트웨어를 포함한 컴퓨터 기술의 도입이다 [3-7].

일반적으로 소프트웨어를 포함하는 디지털 계측제어 시스템은 아날로그 계통에 비해 데이터의 전송 및 처리능력이 월등하고, 적은 표류현상(drift) 및 CPU나 메모리 등과 같은 각종 자원(resources)들을 공유할 수 있다는 점이 큰 장점으로 부각되고 있다. 그러나 디지털 계측제어 시스템은 아날로그 시스템에 비해 온도, 습도 및 전자파와 같은 주변 환경에 민감할 뿐 아니라 많은 자원을 공유하기 때문에 설계 및 프로그램 과정중에 포함될 수 있는 오류(error)들로 인해 다중성을 저해하는 공통원인고장(Common Cause Failure; CCF)에 취약한 면을 가지고 있다는 것이 널리 알려져 있다 [8-10].

지금까지 원자력 산업계에서는 컴퓨터 기술의 잠재적인 위험성으로 인해 적극적인 디지털 계측제어 시스템의 수용을 주저해 왔다. 즉, 컴퓨터 기술은 정확히만 구현된다면 운전효율 및 신뢰도를 크게 향상시킬 수 있을 것으로 예상되지만, 잘못 구현될 경우 안전성을 크게 떨어뜨릴 수 있기 때문이다. 비록, 컴퓨터 기술을 사용하고 있는 다른 산업계의 운전경험에 의하면, 대다수의 설비들이 성공적으로 운영되고 있지만 이는 원자력 발전소와 같이 높은 신뢰도를 요구하는 설비에 비해 상대적으로 낮은 신뢰도를 갖는 설비들이기 때문에 원자력 발전소에 사용되기 위해서는 추가적인 상세 분석을 수행해야 한다.

확률론적 안전성 평가(Probabilistic Safety Analysis; PSA)는 원자력 발전소의 안전성을 평가하는 주요한 수단으로 수행되어 왔으며, 지금까지 원자력 발전소 대부분은 아날로그 계측제어 시스템을 사용했기 때문에, PSA의 대부분은 하드웨어적인 특성에 맞추어 수행되어 왔다. 그러나 소프트웨어를 포함한 컴퓨터 기술을 사용하는 디지털 계측제어 시스템의 도입에 따라, PSA 방법으로 디지털 계측제어 시스템의

안전성을 평가하는 방법이 요구되고 있다. 그러나 소프트웨어는 aging이 일어나지 않는다는 점 (즉 소프트웨어의 고장은 random process로 볼 수 없다) 과 소프트웨어 의non-linearity 특성으로 인해 기존 PSA 방법의 직접적인 적용이 매우 어려운 상황이다 [9, 11, 12].

따라서 본 보고서에서는 디지털 계측제어 시스템의 PSA 적용시 반드시 요구되는 소프트웨어의 고장 확률 (failure probability)을 예측할 수 있는 방법론을 도출하기 위해, 소프트웨어 신뢰도 공학분야(software reliability engineering)에서 제시된 다양한 종류의 소프트웨어 신뢰도 평가 방법을 조사하고 이들의 특징을 분석하였다.

제 2 장 디지털 계측제어 시스템에 대한 PSA

제 1 절 PSA 기법의 소개

전통적으로, 원자력 발전소의 안전성은 결정론적 안전성 분석(deterministic safrty analysis) 및 PSA 기법을 조합하여 평가하게 된다. 여기에서 결정론적 안전성 분석은 주로 설계기준 사고분석(design basis accident analysis)을 통해 수행되고, PSA는 시스템 수준의 안전성 혹은 신뢰성에 기여하는 사건들의 상대적인 영향을 평가하기 위해 사용된다. 그림 2.1은 일반적인 PSA의 절차를 보여주고, 표 2.1은 그림 2.1의 각 단계에 대한 개략적인 설명을 보여준다.

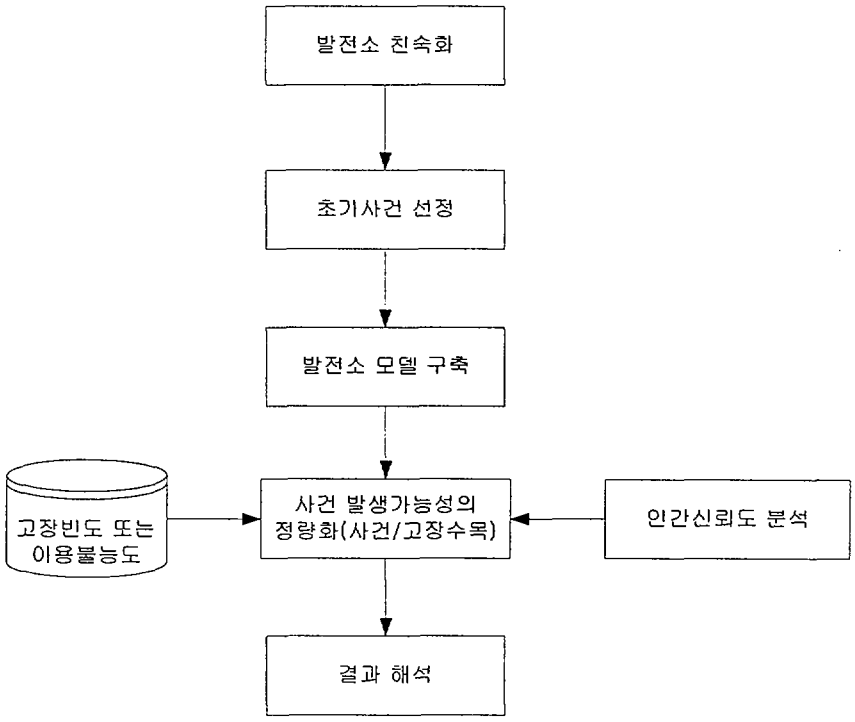


그림 2.1 일반적인 PSA 수행절차

표 2.1 PSA 수행단계에 대한 상세 내용

절차	내용
발전소 친숙화	PSA 수행을 위해 원자력 발전소에 대한 전반적인 정보를 얻는 단계로서, 주로 발전소 설계자료나 다양한 운전 절차서들을 통해 발전소의 안전성에 영향을 줄 수 있는 계통 및 기기들에 대한 정보를 얻게된다.
초기사건 선정	내부사건 선정: 원자력 발전소에 설치된 계통이나 기기들에 의해 발생할 수 있는 사건들 중 안전성에 영향을 줄 수 있는 사건들을 선정하는 단계로서, 주로 설계자료에 의한 논리적인 평가나 기존 PSA 경험에 의해 선정된다. 외부사건 선정: 원자력 발전소의 외부에서 기인한 사건들에 의해 발전소의 안전성이 영향을 받을 수 있는 사건들을 선정하는 단계로서, 주로 지진, 태풍, 홍수 및 화재 등이 고려된다.
발전소 모델 구축	선정된 내부 및 외부사건들에 의해 발전소의 안전성이 어떠한 경로로 위협을 받는지를 구체적인 기기수준까지 모델링하는 단계로서, 주로 사건수목(Event Tree)이나 고장수목(Fault Tree) 기법이 사용된다. 여기에서, 사건수목은 내부 및 외부사건이 일어났을 때 발전소의 안전성에 영향을 줄 수 있는 사건들의 진행상황을 표현하고, 고장수목은 이러한 사건수목에 포함된 사건들이 각각의 기기에 대한 고장이 발생했을 경우 어떻게 영향을 받는지를 표현하게 된다.
사건발생 가능성의 정량화	사건수목이나 고장수목으로 표현된 모델에 대해, 기기들에 대한 고장빈도나 이용불능도를 입력하여 사건수목이나 고장수목을 분석한 후, 발전소 안전성에 대한 정량적인 분석결과를 얻는 부분이다. 일반적으로 PSA 분석에 있어서 가장 중요한 부분으로 인식되고, 이때 발전소 안전성은 노심손상빈도 또는 계통 이용불능도로 표현된다.
인간신뢰도 분석	발전소 운전을 위해 필요한 운전원, 보수요원 및 기타 발전소 업무 종사자들의 실수가 발전소의 안전성에 얼마나 영향을 미치는지에 대한 정량적인 분석을 수행하는 부분이다.
결과해석	인간신뢰도 분석 및 정량화된 사건 발생 가능성의 종합적인 분석을 통해 발전소의 취약부분을 찾아내고 이를 보완하기 위한 대응방안을 제시하는 부분이다.

PSA는 물리적인 고장, 복구 과정, 인간 행위 및 고도의 불확실성을 가지는 기타 사건들로 인해 발전소의 안전성이 얼마나 위협받는지를 종합적으로 평가할 수 있는 수단을 제공한다. 따라서 PSA의 목적은 현재 분석된 발전소의 안전성이 얼마라는 것(bottom line number)을 제공하는 것이 아니라, 현재 발전소 안전성에 대해 이를 향상시키려면 어떤 조치를 수행해야 하는지에 대한 종합적인 결정 (decision-making)

을 지원할 수 있는 근거자료를 제공하는 것이 된다. 이를 위해, PSA 분석은 크게 발전소의 정성적 모델링과 정량적인 분석을 수반하게 된다. 여기에서, 정성적인 발전소 모델링은 주로 사건수목(event tree; ET)와 고장수목(fault tree; FT)이 사용되고, 이렇게 모델링된 부분에 대해 각 고장수목에 포함된 기본사건(basic event)의 발생확률(기기의 고장빈도 또는 이용불능도)을 입력하여 최종적으로 발전소의 안전성을 정량적으로 평가하게 된다. 이때 기기의 고장빈도 및 이용불능도의 의미는 다음과 같고, 이들은 모두 발전소의 운전이력이나 고장이력의 조사를 통해 얻어진다.

- 고장빈도: 기기가 시간당 또는 작동 회수 당 고장이 발생하는 발생율을 의미한다. 시간당 고장빈도는 같은 유형의 기기에 대해 고장횟수의 합을 운전시간의 합으로 나눈 것이고, 작동 회수당 고장빈도는 고장횟수의 합을 작동횟수의 합으로 나눈 것이다.
- 이용불능도: 기기가 고장/정비 등의 이유로 인해 이용될 수 없는 가능성을 의미하고, 같은 종류의 기기에 대해 각 기기가 정지되어 있던 시간의 합을 전체 운전시간으로 나누어 얻을 수 있다.

이러한 자료 이외에, 정량적인 평가를 위해 PSA에서 고려하는 부분은 dependency이다. 예를 들어 HPSI 펌프(High Pressure Safety Injection Pump; 고압 안전주입 펌프)의 기동신호를 발생하는 계측기가 오동작을 일으킬 경우, 비록 펌프 자체는 고장나지 않았지만 필요할 때 동작되지 않았기 때문에, PSA 모델에서는 기기의 이용불능으로 표현되게 된다. 표 2.2는 일반적으로 PSA에서 고려하고 있는 dependency의 종류를 보여준다.

표 2.2 PSA에서 고려되는 dependency 종류

Dependency	적용 예
Functional dependency	Signal 고장으로 인한 HPSI 펌프의 기동 실패
Spatial dependency	화재로 인한 영향
Human dependency	계측기 고장으로 인해 잘못 표시된 정보를 운전원이 사용할 경우
CCF(Common Cause Failure; 공통원인고장)	동일한 제작자에 의해 제작된 기기(이러한 경우, 고장이 발생할 환경이 동일하기 때문에 redundancy의 개념이 존재하지 않는다)

제 2 절 디지털 계측제어 시스템에 대한 PSA 적용

디지털 계측제어 계통에 대한 PSA 수행에 있어서, 가장 중요한 부분 중 하나는 기본사건의 발생 확률이다. 즉, 기본사건에 대한 발생확률을 구할 수 없을 경우 최종적인 신뢰도 값을 정량적으로 평가할 수 없기 때문이다. 그러나 소프트웨어가 포함된 경우 다음과 같은 문제들로 인해 소프트웨어의 고장 확률을 예측하는 것이 상당히 힘들다는 주장이 제기되고 있다 [9, 11, 12].

o Aging 문제

아날로그 계측제어 시스템에 대한 PSA 기법은 aging으로 인한 하드웨어의 무작위성 고장 (random failure)을 고려하여 구해진 고장 확률을 사용하고 있다. 그러나 디지털 계측제어 시스템의 경우 하드웨어 뿐 아니라 소프트웨어가 반드시 포함되고, 이때 소프트웨어는 aging이 일어나지 않기 때문에 무작위성 고장을 고려하는 기존 PSA 방법에는 직접적으로 사용될 수 없다.

o 소프트웨어의 non-linearity 문제

아날로그 계측제어 시스템에 대한 기존의 PSA에서는 하드웨어의 고장 확률을 구하기 위해 원자력 발전소 뿐 아니라 다른 설비(예를 들어, 화학공장이나 석유정제공장 등)에 설치된 동일하거나 유사한 계측제어 시스템의 운전이력을 참고하고 있다. 예를 들어 100°C~200°C 사이에서만 사용될 수 있게 설계된 A란 형태의 아날로그 계측제어 시스템은 모든 설비에 대해 주어진 물리적 환경에 맞게 설치되고 운영될 것이기 때문에 원자력 발전소를 제외한 다른 설비에서의 운전이력을 사용하여 고장 확률을 예측하더라도 큰 문제가 없었다. 그러나 소프트웨어의 경우 각 설비의 물리적 환경에 맞추어 선택적으로 사용되기보다는, 소프트웨어 자체의 수정을 통해 원하는 기능을 수행하도록 설치되고 운영되게 된다. 따라서 아무리 적은 수정이 소프트웨어에 가해졌다고 하더라도 완전히 동일한 소프트웨어라고 볼 수 없기 때문에, 이렇게 얻어진 운전이력을 원자력 발전소에 설치된 소프트웨어의 고장 확률을 예측하기 위한 참고자료로 쉽게 사용할 수 없다. 또 다른 예로서는 A 및 B라는 컴퓨터에 설치된 동일한 소프트웨어 (Windows 98과 같은)의 경우를 들 수 있다. 이때, 같은 소프트웨어라도 A란 컴퓨터에서는 잘 동작하는 반면 B라는 컴퓨터에서는 그렇지 못한 경우를 주위에서 쉽게 볼 수 있다. 이러한 경우, 동일한 소프트웨어를 사용하지만 그 고장이력

은 소프트웨어가 설치된 환경 등에 의해 크게 영향을 받을 수 있기 때문에, 과거의 운전 이력을 신뢰도 예측에 사용하기가 힘들게 된다.

o 소프트웨어의 dependency 문제(또는 공통유형고장 문제)
일반적으로 알려진 바와 같이, 소프트웨어를 포함하는 디지털 계측제어 시스템은 CPU나 메모리 또는 I/O 등과 같은 자원(resource)들을 서로 공유하면서 기능을 수행할 수 있도록 구성되기 때문에, 아날로그 계측제어 시스템에 비해 상대적으로 높은 자원 이용도를 가지게 된다. 그러나 소프트웨어에는 설계 및 프로그램 과정 중에 오류 (error) 또는 결함 (fault) 들이 포함될 수 있고, 비록 이러한 오류로 인해 소프트웨어의 전체 기능 중 일부분만이 고장을 일으킨다 하더라도 이러한 고장의 영향은 많은 자원들을 공유하는 다른 기능들로 전파되어 결국 모든 기능에 대한 고장을 발생시킬 수 있다.

이에 반해, 소프트웨어가 포함된 경우라도 기존의 PSA 방법론을 그대로 사용할 수 있다는 반론도 제기되고 있다. 즉, 디지털 계측제어 시스템은 반드시 소프트웨어와 하드웨어가 조합되어 사용되고, 소프트웨어 고장의 원인이 되는 오류가 소프트웨어에 포함되어 있다고 하더라도 이 소프트웨어의 V&V (Validation and Verification) 가 충실히 수행되었다면 결국 소프트웨어의 고장 원인이 되는 결함들은 거의 제거 되었다고 가정할 수 있다. 이러한 경우, 고장이 발생되는 것은 소프트웨어 입력 오류에만 기인한다고 볼 수 있고 소프트웨어의 입력 오류는 인적오류 및 aging으로 인한 하드웨어 고장이 그 원인이 된다. 이때 이들은 모두 무작위성 고장으로 취급할 수 있기 때문에 결국 소프트웨어 고장을 무작위성 고장으로 볼 수도 있다는 주장이다 [1, 2. 12].

이러한 주장은 소프트웨어 고장 확률 (또는 고장율) 예측에 매우 중요한 사항으로서, 만일 소프트웨어 고장이 무작위성 고장이라는 것이 인정된다면 이미 다른 설비들에 설치되어 운영되고 있는 디지털 계측제어 시스템의 운전이력도 소프트웨어 고장 확률을 예측하기위해 사용될 수 있기 때문이다. 그러나 "소프트웨어의 고장은 무작위성 고장으로 볼 수 있다 또는 볼 수 없다" 라고 하는 의견이 팽팽히 양립하고 있기 때문에, 디지털 계측제어 시스템에 대해서는 현재까지 소프트웨어의 고장 확률을 예측할 수 있는 공인된 방법이 없는 실정이다.

따라서 전 세계적으로도 디지털 계측제어 시스템에 대한 체계적이고 정량적인 신뢰도 평가 방법이 확립되지 않는 상황이고, 또한 현재 각국에서 진행중인 평가방법 개발에 대한 연구도 아직 초기단계이기 때문에 실제적인 PSA에 사용되기 까지

는 아직 많은 연구가 필요하다고 판단된다. 그러나 이렇게 체계적이고 정량적인 방법이 없음에도 불구하고, 많은 원자력 선진국에서는 나름대로의 정량적인 평가기준을 도입하여 사용하고 있다. 예를 들어 캐나다의 경우에는 각 정지계통의 이용불능도를 10^{-3}/Ry로 규정하고 있기 때문에 각 정지계통의 구성요소인 소프트웨어의 고장 확률은 10^{-4}/demand 이하일 것을 요구하고 있고, 영국의 경우에도 원자로 보호계통에 포함된 소프트웨어의 고장 확률을 10^{-4}~10^{-3}/demand로 규정하고 있다 [1, 2, 9, 13, 14]. 즉, 소프트웨어가 포함된 디지털 계측제어 시스템을 PSA 방법을 통해 평가했을 때 개략적이긴 하지만 이러한 정량적인 규정이 없다면 PSA 본연의 목적인 "발전소 안전성에 대해 이를 향상시키려면 어떤 조치를 수행해야 하는지에 대한 종합적인 결정 (decision-making)을 지원할 수 있는 근거자료의 제공" 을 만족시킬 수 없기 때문이다.

이 외에, 민간인 수송기에 포함된 소프트웨어의 고장 확률을 10^{-9}/hr로 경우나 중요한 컴퓨터 시스템의 총 이용 불능도를 10^{-7}/yr로 규정한 미 연방 항공국의 경우 및 소프트웨어 고장 확률을 10^{-12}/hr로 규정한 도시 철도 시스템의 경우에서 [15], 이 값들은 비록 현재 기술로 달성하기 힘들 수도 있지만 어떤 목표를 가지고 소프트웨어의 평가를 한다는 점에 있어서는 원자력 발전소의 경우와 유사하다고 판단된다.

이러한 외국의 접근 방법들을 바탕으로 생각해 볼 때, 신규 및 차세대 원자력 발전소의 건설을 추진하고 있는 우리나라의 경우는 소프트웨어를 포함한 디지털 계측제어 시스템의 신뢰도를 정량적으로 평가할 수 있는 방법론의 개발이 시급하다고 판단된다. 이러한 연구는 1) 소프트웨어 신뢰도 평가방법의 조사 및 타당성 결정, 2) 소프트웨어를 포함하는 디지털 계측제어 시스템 평가를 위해, 소프트웨어에 대해 선정된 신뢰도 평가방법과 하드웨어에 적용되는 신뢰도 평가방법과의 연계 및 3) 개발된 신뢰도 평가방법에 대한 적용성 검증과 같은 3단계로 추진되어야 할 것으로 판단된다. 따라서 본 연구에서는 가장 첫 단계인 소프트웨어 신뢰도를 평가하기 위해 기존에 제시되거나 개발된 방법론들을 조사하여, 그 타당성을 검토하였다.

제 3 장 소프트웨어 신뢰도 평가 방법

ANSI(American National Standards Institute)에 따르면 소프트웨어 신뢰도는 "the probability of failure-free software operation for a specific period of time in a specified environment" 와 같이 정의된다 [16]. 여기에서, 소프트웨어 고장(failure)은 다음과 같은 정의를 갖는 오류(error), 결함(fault) 및 결점(defect)에 의해 발생한다고 고려된다 [17].

- o 오류: 소프트웨어 구현중 발생할 수 있는 모든 종류의 인적 오류(human error)
- o 결함: 소프트웨어의 일부 또는 전체 기능 상실 (고장)을 실제적으로 발생시킬 수 있는 모든 종류의 오류들로서 때때로 bug라고도 불린다.
- o 결점: 소프트웨어에 포함된 모든 종류의 오류나 결함으로 인해 발생할 수 있는 소프트웨어의 이상(anomaly)으로서, 소프트웨어 작성(coding) 뿐 아니라 소프트웨어의 부적절한 설계 및 요건(requirement) 등으로 인한 소프트웨어의 이상을 모두 포함한다.

이러한 오류, 결함, 결점 및 고장에 대한 관계는 그림 3.1를 통해 보다 명확해 질 수 있다.

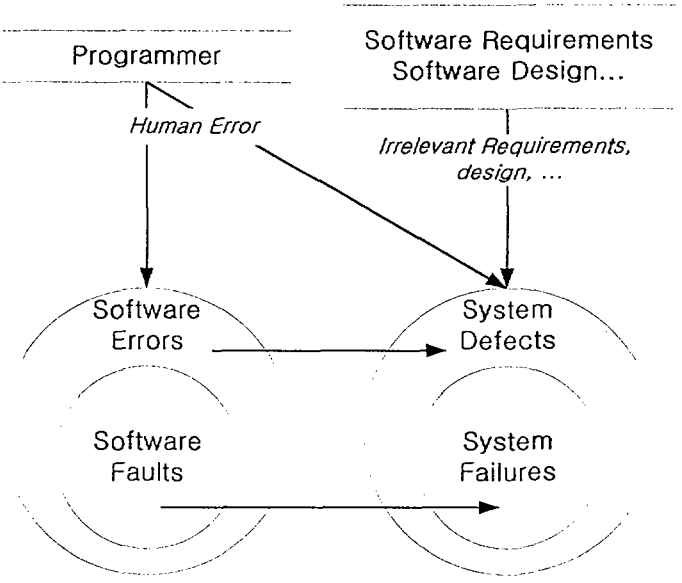


그림 3.1 소프트웨어 오류, 결함, 결점 및 고장

즉 소프트웨어에 많은 오류가 포함되었다 하더라도, 시스템의 고장을 발생시키는 결함들이 상대적으로 적다면 이 소프트웨어는 좋은 신뢰도를 가진다고 말할 수 있게 된다. 따라서 ANSI의 정의에 의하면, 소프트웨어의 신뢰도를 구한다는 것은 소프트웨어가 주어진 환경에서 적절히 동작할 확률 또는 적절히 동작하지 못할 확률을 구하는 것이고 [17, 18], 일반적으로는 후자의 경우에 대해 소프트웨어 고장확률(또는 고장율)을 소프트웨어 신뢰도(software reliability)로 표현하고 있다.이러한 관점에서, 현재까지 소프트웨어의 신뢰도를 평가하기 위해 제안/개발된 평가 방법들은 표 3.1과 같이 정리될 수 있다.

표 3.1 소프트웨어 신뢰도 평가 방법론

<table>
<tr><td rowspan="11">Indirect estimation (predicting software reliability using development process quality)</td><td rowspan="4">Implementation quality</td><td rowspan="2">Software size</td><td>Physical size (based on the number of line of code)</td><td>Category 1</td></tr>
<tr><td>Functional size (based on the number of software functions)</td><td>Category 2</td></tr>
<tr><td rowspan="2">Software complexity</td><td>Based on software structure or architecture</td><td>Category 3</td></tr>
<tr><td>Based on psychology</td><td>Category 4</td></tr>
<tr><td>Requirement quality</td><td colspan="2">Evaluating software requirement quality</td><td>Category 5</td></tr>
<tr><td>Design process quality</td><td colspan="2">Evaluating the quality of software design, design organization, design environment and Implementation</td><td>Category 6</td></tr>
<tr><td rowspan="3">Testing process quality</td><td colspan="2">Evaluating test coverage</td><td>Category 7</td></tr>
<tr><td colspan="2">Evaluating test effort</td><td>Category 8</td></tr>
<tr><td colspan="2">Evaluating availability/capability of software testing team</td><td>Category 9</td></tr>
<tr><td>Miscellaneous</td><td colspan="2">Multivariate approach</td><td>Category 10</td></tr>
<tr><td rowspan="3">Direct estimation (estimating software reliability)</td><td rowspan="3">Predicting software failure rate through analytical or mathematical methods</td><td rowspan="2">Concerning end product only</td><td>Based on software testing + statistical techniques</td><td>Category 11</td></tr>
<tr><td>Based on software modeling</td><td>Category 12</td></tr>
<tr><td>Including development processes + End Product</td><td>Based on models which can synthesize/integrate of each development process and end product (software)</td><td>Category 13</td></tr>
</table>

표 3.1에서 볼 수 있는 바와 같이, 소프트웨어의 신뢰도를 평가하기 위해 지금까지 개발/제안된 방법들은 크게 소프트웨어의 품질(quality)을 여러 가지 평가척도를 통해 평가한 후 이들을 통해 소프트웨어의 신뢰도를 간접적으로 예측하는 방법(category 1 ~ 10)과 확률/통계 처리 또는 소프트웨어의 모델링을 통해 직접적으로 예측하는 방법(category 11 ~ 13)들로 구분될 수 있다. 이들 중, 소프트웨어의 신뢰도를 직접 예측하는 방법들은 비교적 최근에 개발된 방법들이기 때문에 편의상 소프트웨어의 신뢰도를 간접적으로 예측하는 방법론에 대해 먼저 설명하기로 한다.

제 1 절 간접적인 평가 방법론

소프트웨어의 신뢰도를 간접적으로 평가하는 방법론들은, 소프트웨어 신뢰도에 영향을 주는 많은 인자(factor) 들 중 어떤 것을 주된 요인으로 보느냐에 따라 여러 가지 방법들로 구분될 수 있다. 일반적으로는 소프트웨어 신뢰도에 영향을 주는 인자들로서는 소프트웨어 개발에 있어서 전통적으로 고려되는 "소프트웨어 life cycle" 에 포함되는 각각의 개발단계(phase 또는 process)가 주로 고려되고, 그림 3.2는 전형적인 소프트웨어 life cycle의 예를 보여준다 [19].

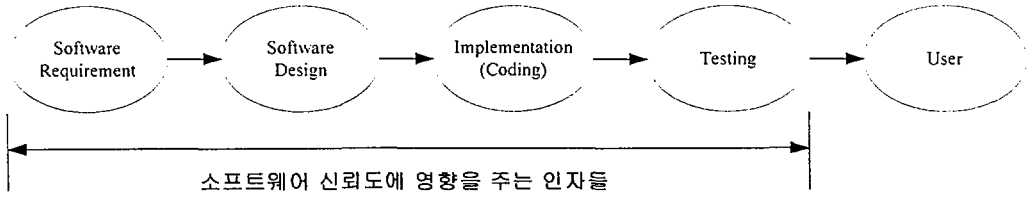


그림 3.2 소프트웨어 life cycle

그림 3.1에서 볼 수 있는 바와 같이, 소프트웨어의 개발을 위해서는 크게 소프트웨어의 설계요건(requirement)을 정하고, 소프트웨어의 구조 등을 설계한 후, 소프트웨어를 구현하여 최종적으로 개발된 소프트웨어를 시험하는 4단계를 거치게 된다. 이때 잘못된 소프트웨어 요건을 설정하거나 소프트웨어의 잘못된 구현 등과 같은 경우를 방지하기 위해, 각 단계에서는 반드시 V&V(Verification & Validation)을 수행하도록 되어 있다. 따라서 각 개발 단계의 소프트웨어 품질(quality) 평가를 통해 최종적으로 개발된 소프트웨어의 신뢰도를 예측할 수 있을 것이라는 생각이 간접적인 소프트웨어 신뢰도 평가의 기본 개념이다. 이때 소프트웨어의 품질은 다양한 평가척도 (소프트웨어에 포함된 결함 수나 소프트웨어의 구조적 복잡도 등)를 통해 평가된다.

1. Implementation Quality (소프트웨어 구현 적합성 평가)

소프트웨어의 신뢰도에 영향을 주는 인자들의 평가에 있어서, 가장 먼저 제안/개발된 평가척도는 소프트웨어의 구현에 관련된 복잡도 평가척도(complexity measure)이다. 즉, 소프트웨어의 복잡도가 크면 클수록 소프트웨어 프로그래머 (programmer) 혹은 보수/관리 요원(maintenance personnel)의 오류 발생 가능성이 커질 수 있고, 이

러한 프로그래머의 오류는 결국 소프트웨어의 고장(failure)을 발생시킬 수 있기 때문이다.

이때 소프트웨어의 복잡도를 평가하기 위해 제안된 방법들은 크게 소프트웨어의 물리적 크기를 평가하는 방법 (category 1), 기능적 크기를 평가하는 방법 (category 2), 프로그래머가 느끼는 심리적 복잡도를 평가하는 방법 (category 3) 및 소프트웨어의 구조적 복잡도를 평가하는 방법 (category 4) 등으로 세분될 수 있다.

가. Category 1 - 소프트웨어의 물리적 크기를 통한 평가

소프트웨어의 물리적 크기라는 것은 말 그대로 소프트웨어에 포함된 원시코드(source code)의 라인 수를 의미한다. 즉, 원시코드의 길이가 길면 길수록 소프트웨어 프로그래머나 보수/관리요원이 이해해야 하는 부분이 증가하기 때문에, 이로 인해 프로그래머나 보수/관리요원의 오류를 유발시킬 수 있는 소프트웨어의 복잡도가 증가할 것이라는 단순한 가정에서 출발하게 된다. 이 범주에 포함되는 대표적인 방법들은 1) bugs per line of code, 2) fault density, 3) design defect density 등을 들 수 있고, 이들 중 bugs per line of code에 대한 개략적인 설명은 표 3.2와 같다 [20, 21].

표 3.2 Bugs per line of code에 대한 이론적 배경 및 평가 결과

이론적 배경	Rough industry estimation (5~30 failures/thousand executable line of code)을 바탕으로 제안된 방법이다. Si = The number of executable source lines in the ith module Fi = Estimated number of failures in the ith module F = Estimated number of failures in the whole program N = The number of modules contained in a program $F_i = 4.2 + 0.0015 \cdot S_i^{\frac{4}{3}}$ $F = \sum_{i=1}^{N} F_i$
평가 결과	Crude estimation for the number of failures in a program per line of codes

표 3.2에서 알 수 있는 바와 같이, 소프트웨어의 물리적 크기를 통한 평가방법들은 기존의 경험(1000 라인 당 약 5~30개 정도의 소프트웨어 결함이 포함)을 바탕으로 제안된 것들이기 때문에 평가결과 자체가 불확실할 뿐 아니라, 주로 C나

FORTRAN 및 어셈블리 언어 등과 같이 프로그래머가 일일히 코딩을 해야 하는 low level language를 대상으로 한 것이기 때문에 현재 자주 사용되는 객체지향 프로그램(object-oriented program) 또는 high level language의 평가에는 적용되기 힘든 단점이 있다.

나. Category 2 - 소프트웨어의 기능적 크기를 통한 평가

소프트웨어의 크기를 고려하는데 있어서 또 다른 기준은, 단순히 소프트웨어의 물리적인 크기보다 소프트웨어에 얼마나 많은 기능(function)들이 포함되는가를 사용하는 것이다. 즉, 소프트웨어가 아무리 많은 code line수를 갖더라도 소프트웨어가 포함하고 있는 기능들의 수가 작다면 프로그래머나 보수/관리요원들이 비교적 쉽게 소프트웨어의 구조를 이해할 수 있을 것이기 때문이다.

소프트웨어의 크기를 포함된 기능들의 수로 평가하는 대표적인 방법으로는 function point analysis(또는 feature point analysis)를 들 수 있다. 이 방법은 평가하고자 하는 소프트웨어의 크기를 다음과 같은 5개의 기능들의 개수를 통해 결정한다.

- The number of functions that are related to internal logical files (logical groups of data maintained in an application)
- The number of functions that are related to external interface files (logical groups of data used by one application but maintained by another application)
- The number of functions that are related to external inputs (maintained by internal logical files)
- The number of functions that are related to external outputs (reports and data leaving the application)
- The number of functions that are related to external inquiries (combination of a data request and data retrieval)

이렇게 정해진 기능들의 개수에 대해, 현재까지 발견된 소프트웨어의 결함 수를 고려하여 최종적으로 소프트웨어 품질을 다음과 같이 평가하게 된다.

$$\text{소프트웨어 품질} \propto \left[\frac{\text{현재까지 발견된 결함들의 총 수}}{\text{기능들의 총 수}}\right]^{-1}$$

그러나 이러한 방법은, 비록 소프트웨어에 포함된 기능들의 수는 쉽게 계산될 수 있지만, 현재까지 발견된 소프트웨어 결함 수에 따라 소프트웨어의 품질이 변하게 된다. 따라서 소프트웨어에 포함된 기능의 수 뿐만 아니라 다음에 설명될 소프트웨어 testing quality(소프트웨어에 포함된 결함들을 얼마나 잘 감지할 수 있는가?)와도 관련이 있기 때문에, 소프트웨어 품질에 대한 어떤 결과를 얻었을 때 이에 대한 해석이 불분명한 경우도 발생될 수 있다.

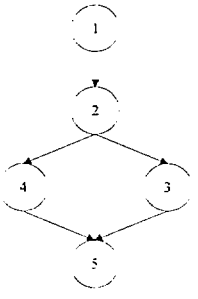
다. Category 3 - 소프트웨어의 구조적 복잡도를 통한 평가

이 category에 포함되는 평가 방법들의 기본 개념은 category 1 및 2와 유사하다. 즉 앞의 방법들이 소프트웨어의 크기나 포함된 기능들의 수를 통해 소프트웨어의 복잡도를 평가하는데 반해, 이 category에 포함된 방법들은 소프트웨어의 구조(architecture 또는 structure)를 통해 소프트웨어의 복잡도를 평가하게 된다.

현재, 소프트웨어의 구조적 복잡도 평가를 위해 사용되는 여러 가지 방법들은 단순히 소프트웨어의 내용을 분석하거나 소프트웨어의 구조를 그래프로 표현한 후 이를 그래프 이론(graph theory)을 통해 분석하는 방법을 취하고 있다 [22]. 그래프를 사용하여 분석하는 방법들 중 대표적인 것은 cyclomatic complexity로서 이는 소프트웨어의 원시코드(source code) 구조를 분석하여 그 복잡도를 평가하는 방법이다. 예를 들어, 표 3.3과 같은 원시코드 및 원시코드의 수행구조를 가정해 보자.

표 3.3 임의의 원시코드 및 원시코드의 수행구조

Textual Program (원시코드)	원시코드의 수행구조

********** ***BLOCK NO. 1*** **********
NTOT = (NCHARS + 7)*NWORDS
********** ***BLOCK NO. 2*** **********
IF(…) GOTO 900
********** ***BLOCK NO. 3*** **********
ELSE
********** ***BLOCK NO. 4*** **********
900 PRINT NTOT
********** ***BLOCK NO. 5*** **********
990 STOP
END
	
	
	The number of edges = e = 5
	The number of nodes = n = 5
	Cyclomatic complexity
	= e - n + 2 = 2

여기에서 원시코드의 수행구조라는 것은 일종의 흐름도(flow chart 또는 flow graph)로서, 원시코드가 수행되는 내용을 그래프를 통해 축약적(abstractive)으로 표현한 것이다.

이러한 구조에 대해, cyclomatic complexity는 단순히 원시코드의 수행구조에서 얻은 edge 수와 node 수만을 바탕으로 간단히 계산된다. 이 외에도 많은 평가 방법들이 소프트웨어의 구조적 복잡도를 평가하기 위해 제안되어 있다. 표 3.4는 이러한 평가 방법들의 종류를 보여주고, 이들 방법들에 대한 자세한 설명은 부록에 기술되어 있다.

표 3.4 소프트웨어의 구조적 복잡도 평가 방법

평가 방법	비고
BVA model	소프트웨어에 포함된 자료구조를 평가
Data structure metrics	
Operational or functional complexity (Interface/Micro/Operator)	소프트웨어의 기능 및 모듈의 연관관계를 평가
Number of entries & exits per module	
Design structure	
Data flow complexity	
Cohesion/Coupling	
Functional complexity	
Data structure metrics	
System design complexity	
Entropy measure	소프트웨어의 구조적 복잡도 및 소프트웨어에 포함된 자료구조를 평가

라. Category 4 - 프로그래머가 느끼는 심리적 복잡도를 통한 평가

Halstead에 의해 제안된 이 평가방법의 경우(Halstead's E measure), 평가를 위한 기본 개념은 앞에서 설명한 category 1~3에 포함되는 방법들과 동일하다. 즉, 소프트웨어가 복잡할 경우 소프트웨어에 대한 프로그래머나 관리/보수 요원의 이해도가 떨어지기 때문에 이들로 인한 오류가 소프트웨어에 포함될 가능성이 증가하고, 이는 곧 개발된 소프트웨어의 신뢰도와 직결되기 때문에 소프트웨어의 복잡도 평가를 통해 소프트웨어의 신뢰도를 간접적으로 예측한다는 것이다.

그러나 category 1~3까지의 평가 방법들은 모두 소프트웨어 자체만 고려한 데 반해 category 4의 경우는 공업 심리학(engineering psychology) 또는 인지공학(cognitive engineering) 을 바탕으로 하여 프로그래밍 직무에 대한 난이도(또는 프로그램의 이

해도)를 평가한다는 것이 큰 차이점이다. 표 3.5는 임의의 원시코드에 대해 Halstead가 제안한 방법으로 평가하는 예를 보여준다.

표 3.5 임의의 원시코드에 대한 심리적 복잡도 평가 예

$$E = \frac{\eta_1 N_2 (N_1 + N_2)}{2\eta_2} \log_2 (\eta_1 + \eta_2)$$

η_1 = the number of unique operators

η_2 = the number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

E = Average efforts for programming

소스코드	Operator	Frequency	Operand	Frequency

IF (A=0)
THEN
A = B;
ELSE
A = C;
	;	2	A	3

	=	3	B	1
	( )	1	C	1
	THEN	1	$\eta_1 = 6$ $N_1 = 9$ $\eta_2 = 3$ $N_2 = 5$ $E = 221.90$	
	ELSE	1		
	IF	1		

이렇게 계산된 E 값에 대해, Halstead는 다음과 같은 평가식을 제안하였다.

$$\text{원시코드에 포함된 결함의 수} \cong \frac{E^{\frac{2}{3}}}{3000}$$

위 식에 의하면 원시코드에 포함된 operator나 operand의 수가 증가할수록 프로그래머나 관리/보수 요원이 이해해야 하는 정보의 양이 증가하기 때문에 복잡도가 증가하고, 이로 인해 결국 결함의 수가 증가한다는 것을 표현하고 있다.

2 Category 5 - Requirement Quality (설계요건 적합성 평가)

그림 3.1에서 볼 수 있는 바와 같이, 소프트웨어 개발에 있어서 가장 먼저 수행되는 단계는 소프트웨어의 설계요건(requirement)을 정하는 부분이다. 따라서 논리적인 오류나 서로 상충되는 부분이 설계요건에 설정된다면, 이 설계요건에 따라 설계,

구현 및 시험된 소프트웨어는 필연적으로 오류가 포함되어 있을 가능성이 매우 높기 때문에 소프트웨어 신뢰도는 매우 낮아지게 된다. 또한 소프트웨어 개발업체의 경험에 따르면, 소프트웨어를 설계 및 구현 단계에서 고치는 비용에 비해 설계요건을 수정하는 비용은 약 1% 정도로 추산되므로 [17], 가능한 한 소프트웨어 초기단계에서 포함될 수 있는 오류를 수정하는 것이 매우 바람직하다는 의견이 지배적이다. 따라서 표 3.6에 정리된 것과 같이 소프트웨어 설계요건의 적절성을 평가하는 다양한 방법들이 개발/제안되었다. 이들 방법들에 대한 자세한 설명은 부록에 기술되어 있다.

표 3.6 소프트웨어 설계요건의 적합성 평가 방법

평가 방법	비고
Cause & effect graphing	설계요건들을 그래프 형태로 표현하여 애매하거나 잘못된 요건들을 구별
Requirements traceability	설계요건들 중에서 실제로 구현되지 않거나 중복되어 구현된 요건들을 구별
Completeness	
Completeness	
Number of conflicting requirements	설계요건들 중에서 서로 중복 또는 상충되는 요건들을 구별
Requirements compliance	
Requirements specification change requests	소프트웨어 구현 도중 바뀐 설계요건들의 수(requirement stability)를 평가

3. Category 6 - Design Quality (소프트웨어 설계 품질 평가)

이미 설명한 설계요건 평가 (category 5)와 마찬가지로, 소프트웨어 신뢰도는 소프트웨어를 얼마나 잘 설계하느냐에 따라 많은 차이를 보일 수 있기 때문에, 소프트웨어 설계의 품질을 평가할 수 있는 평가 방법들이 많이 개발/제안되었다. 이러한 평가 방법들은 소프트웨어 설계만 고려하는 것이 아니라 소프트웨어 설계를 담당한 팀의 능력(team capability), 소프트웨어가 사용될 환경 및 소프트웨어 구현환경 등에 대한 평가도 포함하고 있다. 표 3.7은 소프트웨어 설계의 품질 평가를 위해 제안된 방법들을 정리한 것이고, 이들에 대한 자세한 설명은 부록에 기술되어 있다.

표 3.7 소프트웨어 설계 품질 평가 방법

평가 방법	비고
Software process capability determination (SPICE)	소프트웨어 개발 팀을 평가
Cost/Schedule	
Reliability prediction as a function of development environment	소프트웨어가 사용될 환경을 고려하여 소프트웨어 신뢰도를 평가
Reliability prediction for the operational environment	
Reliability prediction as a function of SW characteristics	

4. Testing Quality (소프트웨어 시험의 적절성 평가)

소프트웨어의 신뢰도에 영향을 미치는 또 다른 중요한 인자는 개발중이거나 개발된 소프트웨어에 대해 수행되는 시험의 적절성이다. 즉, 설계요건 적합성이나 소프트웨어 구현의 적합성을 충분히 했다고 하더라도, 시험이 부적절하게 수행되었다면 소프트웨어의 신뢰도에 큰 영향을 줄 수도 있는 숨겨진 결함을 찾아낼 수 없기 때문이다.

이러한 소프트웨어 시험의 적절성을 평가하기 위해 제안된 방법들은 크게 "category 7 - 소프트웨어가 수행할 수 있는 기능에 대해 충분한 시험이 수행되었는가? (test coverage)," "category 8 - 소프트웨어를 시험하기 위해, 충분한 인력 및 노력이 투입되었는가? (test effort)" 및 "category 9 - 소프트웨어 시험을 수행한 기관 또는 팀의 능력은 적절한가? (test team availability or capability)" 의 세가지로 분류될 수 있다.

가. Category 7 - Test Coverage 평가

Test coverage 평가의 목적은 개발중 이거나 개발된 소프트웨어에 포함된 기능(function)이나 모듈(module)들이 빠짐 없이 적절히 시험되었는지를 평가하거나 또는 목표로 하는 신뢰도를 만족시키기 위해 수행해야 하는 시험 회수를 구하는 것으로 대표될 수 있다. 표 3.8은 test coverage 평가를 위해 제안/개발된 방법들을 정리하여 보여주고, 이들 방법들에 대한 자세한 설명은 부록에 기술되어 있다.

나. Category 8 - Test Effort 평가

이 category에 포함되는 평가 방법들은 소프트웨어의 시험을 위해 투입한 시간을 평가하여, 적정 수준 이상의 노력이 소프트웨어 시험을 위해 투입되었는지를 평가하는 데 그 목적이 있다. 표 3.9는 test effort 평가를 위해 제안된 방법들을 보여주고, 이들에 대한 개략적인 설명은 부록에 정리되어 있다.

표 3.8 소프트웨어의 Test Coverage 평가 방법

평가 방법	비고
Functional test coverage	시험된 기능의 수 / 소프트웨어에 포함된 기능(*function*)의 수
Minimal unit test case determination	실제로 시험된 *test case*의 수 / 모든 모듈을 포함하는 *test case*의 수
Test coverage	시험된 기능의 수 / 소프트웨어에 포함된 기능(*function*)의 수
Testing sufficiency	시험으로 인해 발견된 결함 수 / 소프트웨어에 포함되어 있을 것이라고 예측되는결함수
PIE (Propagation, Infection and Execution) technique	각각의 시험에 대해, 시험시 수행되는 원시코드의 특정한 위치, 수행된 원시코드에 의해 변한 값을 갖는 변수들의 수 및 변화된 변수 값에 의해 영향을 받은 결과(output)의 수 등을 모두 고려하여, 주어진 신뢰도를 검증하기 위해 필요한 시험 회수를 구한다.

표 3.9 소프트웨어의 Test Effort 평가 방법

평가 방법	비고
Fault number days	발견된 결함에 대해 이를 고칠 때까지 걸린 시간을 평가
Man-hours per major defect detected	결함 발견 당 투입된 시간을 평가

다. Category 9 - Test Team Availability/Capability 평가

이 category에 포함되는 방법들은, 독립된 팀/조직에서 소프트웨어에 결함들을 심어둔 후, 소프트웨어 시험 맡은 평가팀에서 삽입된 결함들을 얼마나 많이/정확히/빨리 찾는지를 통해 소프트웨어 평가 팀의 적절성을 평가하는 데 그 목적이 있다. 이

러한 방법들은 오류 삽입(error seeding) 기법 또는 오류 추출(sampling) 기법이라고 불리고, 표 3.10은 이러한 평가를 위해 제안된 방법들을 보여준다.

표 3.10 Test Team Availability/Capability 평가 방법

평가 방법	비고
Number of faults remaining	삽입된 결함들의 수에 대해 평가팀에서 발견한 결함 수의 비를 통해 평가팀의 적절성을 평가.
Test accuracy	
Mutation testing	

5. Category 10 - Multivariate approach

지금까지 설명된 여러 가지 방법들은 모두 소프트웨어 신뢰도에 영향을 주는 여러 가지 인자들 중 한가지에 대해서만 평가하는 방법들이었다. 그러나 이미 소프트웨어 신뢰도에 영향을 주는 인자들이 여러 가지라는 것을 알고 있는 상태에서, 특별히 한가지 인자에 대해서만 평가하여 소프트웨어의 신뢰도를 예측하는 것보다 여러 가지 인자들을 종합하여 소프트웨어의 신뢰도를 예측하는 것이 보다 정확하다고 가정할 수 있다. 따라서 multivariate approach는 category 1 ~ 9에 포함된 다양한 평가 방법들을 모두 취합하여 다음과 같은 형태로 소프트웨어 신뢰도를 예측하게 된다.

$$\text{소프트웨어 신뢰도} \approx \sum (Weighting)_i \times (Measure)_i$$

이때, 각 평가척도에 대한 가중치(weighting)는 전문가 판단이나 factor analysis 또는 principal component analysis를 통해 얻게 된다.

Multivariate approach의 대표적인 예는 software release readiness라는 방법으로서, 여기에서는 전문가 판단에 의한 가중치와 functional test coverage (category 2), software document source listings 및 software capability maturity model (category 6) 등의 평가 방법에서 얻은 척도들을 결합하여 소프트웨어의 신뢰도를 예측하는 방법을 제시하고 있다.

제 2 절 직접적인 평가 방법론

지금까지 설명한 소프트웨어 신뢰도 평가 방법들은 모두 소프트웨어 신뢰도에 영향을 주는 인자들의 평가를 통해 간접적으로 소프트웨어의 신뢰도를 평가하는 것들 이었다. 즉, 이런 방법들을 적용할 경우 소프트웨어에 포함된 결함의 수 등과 같이 대부분 소프트웨어의 고장 확률이 아닌 다른 값들을 결과로 얻게 되고, 이렇게 얻어진 값들을 통해 소프트웨어의 신뢰도를 예측하게 된다.

그러나 최근 들어, 이러한 방법과는 대조적으로 소프트웨어의 고장확률을 직접 예측하기 위한 여러 가지 방법들이 제안되고 있다. 이러한 방법들은, 그림 3.1에서 볼 수 있는 다양한 소프트웨어 개발 단계에 대해 평가하는 것이 아니라, 개발이 완료되어 사용자가 사용하기 직전 단계의 소프트웨어에 대한 평가를 수행하는 것이다.

직접적인 소프트웨어의 신뢰도 예측 방법의 기본 가정은 "Pareto Principle" 이라고 알려진 것으로 [23], 그 내용은 "소프트웨어 개발 기간 동안에 관찰된 고장들은 소프트웨어가 포함하고 있는 결함들의 극히 일부만이 원인이 되어 발생된 것이다. 따라서 개발 단계에서 아무리 자세한 평가를 수행하더라도 대부분의 결함들에 의한 고장은 소프트웨어 개발 기간 동안에 나타나지 않기 때문에, 개발이 완료된 소프트웨어에 대한 신뢰도 평가가 가능하다" 라는 것이다. 따라서 소프트웨어 life cycle 입장에서 볼 때, 직접적인 소프트웨어의 신뢰도 평가는 그림 3.3과 같은 시점에서 수행되는 것이 보통이다.

직접적으로 소프트웨어의 신뢰도를 평가하는 방법은 1) 소프트웨어에 대해 많은 시험을 수행하여 이를 통해 얻어진 고장 횟수를 통계적으로 처리하거나, 2) 소프트웨어를 다양한 방법으로 모델링한 후 이를 수학적으로 분석하는 방법이 있다. 이외에 소프트웨어 뿐 아니라 소프트웨어의 개발단계까지를 포함하여 분석하는 방법도 최근 들어 제안되고 있다.

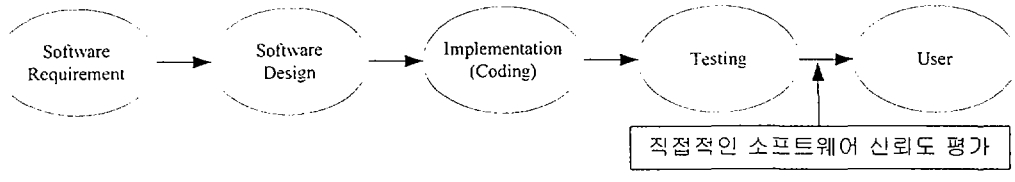


그림 3.3 직접적인 소프트웨어 신뢰도 평가 적용 시점

1. Category 11 - 확률/통계적 처리를 통한 소프트웨어 신뢰도 직접적인 예측

소프트웨어의 신뢰도 예측에 있어서 가장 먼저 발표되었으며 현재도 자주 사용되며 개선되고 있는 방법은 확률/통계적 처리를 통한 신뢰도 예측 방법이고, 그림 3.4는 확률/통계적 처리를 통한 소프트웨어 신뢰도 예측의 일반적인 순서를 보여준다.

여기에서, 사용된 가정 및 도입된 확률분포의 종류 등에 따라 다양한 신뢰도 예측 방법들이 제안/개발되어 있고, 이들 중 현재까지 대표적으로 사용되고 있는 방법들의 종류 및 특징은 표 3.11과 같이 정리할 수 있다 [19, 24-30].

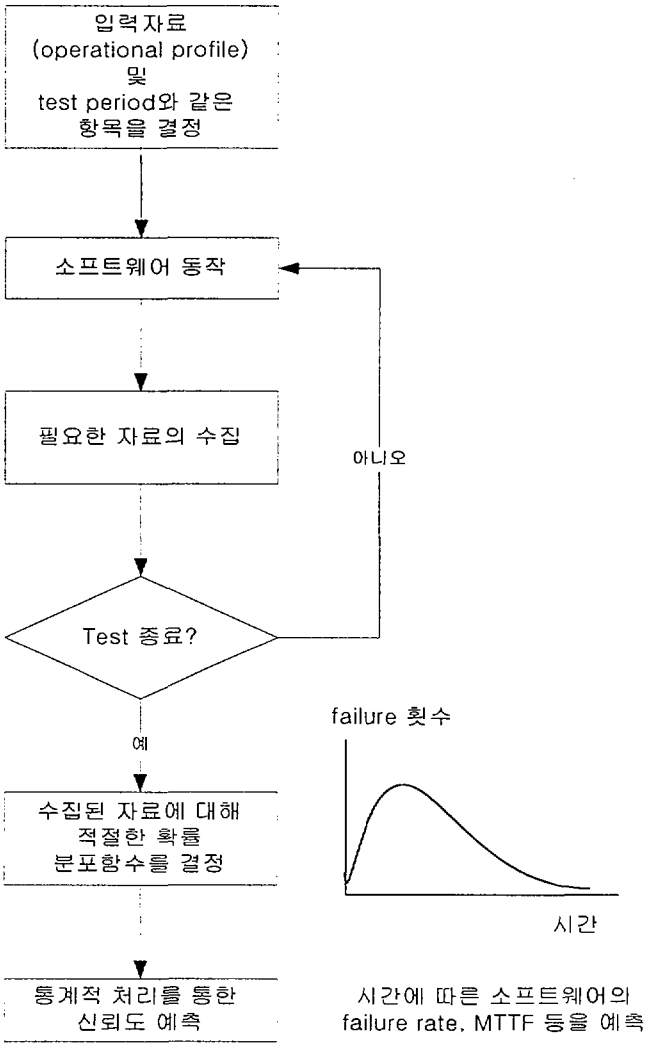


그림 3.4 확률/통계적 처리를 통한 소프트웨어 신뢰도 예측 방법

표 3.11 통계적 처리를 사용한 소프트웨어 신뢰도 예측 방법들 (1/2)

제안된 방법	특징
Jelinski & Moranda model	- 가장 오래된 모델(1972년) 중 하나로서, McDonald사의 Navy project를 위해 개발되어 현재까지 사용되고 있다. - 소프트웨어의 MTTF를 예측하기 위한 모델 - 소프트웨어에는 N개의 유한한 결함이 포함되어 있다고 가정 - Constant 고장 rate를 가정 - MTTF 는 exponential distribution을 따른다고 가정
Non-homogeneous Poisson process (NHPP) model	- 1979년에 Amrit Goel과 Kazu Okumoto에 의해 처음으로 제안 - 시간 t에서 단위시간당 발생되는 고장 수를 예측하기 위한 모델 - 소프트웨어에는 N개의 유한한 결함이 포함되어 있다고 가정 - Constant 고장 rate를 가정 - 단위 시간당 발생하는 결함 수는 Poisson process를 따른다고 가정
Schneidewind's model	- 1975년에 Schneidewind에 의해 처음 제안되었고, space shuttle 제어를 위한 IBM의 비행제어 software에 대해 현재까지 적용되고 있음 - AIAA(American Institute of Aeronautic and Aeronautics)에서, software reliability modeling을 위해 선정한 4개 model 중 하나임 - 시간 t에서 단위시간당 발생되는 고장 수를 예측하기 위한 모델 - 입력자료(고장 data)로서 동일한 간격의 test interval ti 및 ti 동안 발생한 고장 count fi가 필요하다. **(test interval 간격이 동일하지 않으면 쓸 수 없다)** - Constant 고장 rate를 가정 - 소프트웨어에는 N개의 유한한 결함이 포함되어 있다고 가정 - 단위 시간당 발생하는 결함 수는 Poisson process를 따른다고 가정

표 3.11 통계적 처리를 사용한 소프트웨어 신뢰도 예측 방법들 (2/2)

제안된 방법	특징
Musa's basic execution model	- 1975년에 John Musa에 의해 처음 제안 - 현재, software reliability 모델들 중 가장 광범위하게 사용되는 모델 - 총 누적 실험시간(tn)에서의 경과시간 x에 대한 reliability를 예측 - 입력자료로서 고장이 발생한 시간 ti들이 필요 - Constant 고장 rate를 가정 - 단위 시간당 발생하는 결함 수는 Poisson process를 따른다고 가정
Hyper-exponential model	- 1984년에 Ohba에 의해 처음으로 제안 - Non-homogeneous Poisson process(NHPP) model의 확장된 형태 - 소프트웨어가 K개의 class로 구성되어 있다고 가정 (다른 프로그래머가 coding 하거나 다른 프로그래밍 언어 등으로 구성되어 있을 경우, 각 class의 고장 rate는 서로 틀릴 수 있다는 것을 고려하기 위해) - 입력자료(고장 data)로서 동일한 간격의 test interval ti 및 ti 동안 발생한 고장 count fi가 필요하다. **(test interval 간격이 동일하지 않으면 쓸 수 없다)** - 소프트웨어에는 N개의 유한한 결함이 포함되어 있다고 가정 - Constant 고장 rate를 가정 - 단위 시간당 발생하는 결함 수는 Poisson process를 따른다고 가정

이러한 확률/통계적 처리를 통한 소프트웨어 신뢰도 예측은 크게 1) time domain approach, 2) error seeding & tagging approach 및 3) input domain approach로 분류될 수 있다. 그러나 error seeding 방법은 category 9에서 이미 소개된 test team availability/capability 평가와 유사하기 때문에 본 보고서에서는 time domain approach 및 input domain approach만을 설명하였다.

여기에서, time domain approach 및 input domain approach의 차이를 이해하기 위해서는 소프트웨어의 신뢰도의 정의를 다시 한번 살펴볼 필요가 있다. 이미 설명한 바와 같이, ANSI의 정의에 의하면 "소프트웨어의 신뢰도는 소프트웨어가 주어진 환경에서 적절히 동작할 확률" 로 주어져 있다. 그런데 이러한 주어진 환경이라는 것은 소프트웨어 개발자 및 사용자의 입장에서 볼 때 다시 두 가지로 구분될 수 있다. 즉 소프트웨어 사용자의 입장에 있어서는 단순히 "소프트웨어 사용중에 고장이 발생하지 않음" 이 주된 관심사 이지만 소프트웨어 개발자의 입장에서는 "소프트웨어가 설정된 설계요건을 만족시킬 수 있음" 이 주된 관심사가 된다 [17]. 따라서 개발자 입장에서는 소프트웨어를 사용자에게 제공하기 전까지 많은 test를 통해 소프트웨어에 포함된 결함들을 제거하는 것이 강조되지만 사용자 입장에서는 소프트웨어가 고장 없이 동작된다는 점이 강조된다. 이러한 입장차이에 따라 소프트웨

어 시험을 통한 신뢰도 평가방법은 개발자의 입장이 고려되는 time domain approach와 사용자의 입장이 고려되는 input domain approach의 두 가지로 구분할 수 있다.

Time domain approach란 소프트웨어의 신뢰도 예측에 있어서 가장 먼저 발표되었고 또한 현재까지도 사용되고 있는 방법으로서, 소프트웨어에 포함된 전체적인 결함들에 의해 발생될 수 있는 소프트웨어의 고장 확률을 구하기 위해 입력자료로서 시험기간(test period) 동안 발생된 소프트웨어의 고장 횟수를 기록하여 시간에 따른 소프트웨어 고장 횟수의 추이를 구하고, 이에 대한 적절한 확률 분포 함수를 가정하여 시간에 따른 고장확률 또는 MTTF(Mean Time beTween Failure) 등을 예측하는 방법이다. 따라서 time domain approach를 적용하기 위해서는 소프트웨어에 대한 입력 영역 (input range)을 일정한 간격으로 분할하여 각 영역에 대한 시험을 수행하는 것이 보통이다.

이에 반해 input domain approach란 소프트웨어를 평가하기 위해 시험을 수행한다는 측면에서는 time domain approach와 동일하나, 소프트웨어에 대한 입력 영역을 균일하게 분할하지 않고 사용자가 주로 사용하는 기능 또는 특정한 기능에 대해 영향을 미칠 수 있는 소프트웨어의 특정한 입력 영역을 결정하고 (operational profile), 이 부분에 대해서 특히 자세한 시험을 수행하는 방법을 취하고 있다. 따라서 소프트웨어 사용자들에 대해 소프트웨어의 적절한 입력영역을 구하는 것이 관건이 된다 [29, 30].

이러한 두 가지 접근 방법의 차이를 보다 확실히 이해하기 위해 다음과 같은 예를 생각해 볼 수 있다. 예를 들어, "가능한 한 많은 사용자가 이 시스템을 사용할 수 있어야 한다" 라는 설계 요건에 따라 만들어진 전화교환기 시스템의 소프트웨어를 고려해 보자.

시스템 개발자들은 이러한 요건을 만족시키기 위해 사용자가 많을 경우 약간의 시간지연(예를 들어 "사용자가 1000명 이상일 경우는 최대한 10초 이내의 연결시간 지연을 허용한다" 와 같은)을 허용할 수 있다. 이러한 경우 time domain approach 입장에서는 사용자들의 수가 입력영역이 되기 때문에, 이에 대해 일정한 영역 (예를 들어 100명 단위)에 대한 시험을 수행한 후 이에 따라 소프트웨어의 신뢰도를 평가할 수 있다. 그러나 operational profile을 고려해 본다면999번째 사용자 까지는 별다른 문제가 없지만 1000번째 사용자부터는 10초 정도의 연결시간 지연이 발생할 수 있고, 이러한 형태로 연결시간이 지연된다면 사용자 입장에서는 "시스템이 잘못되었다" 라고 생각할 수 있다. 따라서 시스템의 입장에서 볼 때는 결함이 아니라 할지라도 사용자 입장에서는 결함이 발생한 것으로 판단할 수 있기 때문에 일정한 입

력영역 분할은 의미가 없게 된다.

따라서 time domain approach와 input domain approach는 소프트웨어를 시험을 통해 평가한다는 측면에서는 동일하기 때문에 특별한 경우를 제외하고는 표 3.11에 제안된 방법들을 그대로 사용할 수는 있지만, 소프트웨어의 입력영역을 어떻게 구분하느냐에 따라 그 결과에 큰 차이가 있을 수 있다.

2. Category 12 - 소프트웨어 모델링을 통한 신뢰도 예측

이 category에 포함된 방법들의 특징은 소프트웨어를 다양한 방법을 통해 모델링을 한 뒤, 수학적인 분석을 통해 이를 해석하여 소프트웨어의 신뢰도를 예측하게 된다. 표 3.12는 현재까지 개발/제안된 대표적인 방법들을 정리하여 보여준다. 여기에서, 표 3.12에 제시된 여러 가지 방법들에 대한 기본적인 자료는 참고 문헌 [20, 21]에 잘 정리되어 있다.

표 3.12 소프트웨어 모델링을 통한 신뢰도 평가 방법 (1/2)

평가 방법	모델링 방법	비고
System Performance Reliability	QNM (Queueing Network Model)	o 이 모델은 제한된 R0 시간 동안 시스템이 주어진 task를 수행할 확률을 구해준다. System Reliability = p(R<R0) R = System Response Time R0 = Design Goal or Requirement
Independent Process Reliability	Markov Modeling	o Loop가 없거나 local dependency(즉, hardware와 연결되지 않음)가 없는 소프트웨어에 대한 신뢰도를 구한다. o 소프트웨어 신뢰도는 소프트웨어가 초기 상태에서 원하는 상태로 올바르게 전이할 확률로 정의된다.
System Operational Availability	-	o 운전시간에 따른 소프트웨어의 신뢰도를 구한다. o 소프트웨어의 신뢰도는 시간 t에서 사용자의 요구가 입력되었을 때, 소프트웨어가 이용 가능할 확률이다. o Dependency가 있는 소프트웨어에도 적용될 수 있다.

표 3.12 소프트웨어 모델링을 통한 신뢰도 평가 방법 (2/2)

평가 방법	모델링 방법	비고
Reliability Block Diagrams (RBD)	RBD (Reliability Block Diagram)	o 소프트웨어의 모듈 또는 기능을 하나의 블록으로 보고, 이들의 연관관계를 모델링 o 각 블록에 대해, 시간 t에서의 신뢰도는 exponential로 가정.
k-out-of-n model		o RBD 모델에 대해, n개 중 k개의 모듈 또는 기능이 성공해야 하는 것을 모델링 할 때 적용될 수 있다.
Fault tree	Fault Tree Modeling	o 소프트웨어의 모듈 또는 기능들을 고장수목으로 모델링 o 소프트웨어의 기능/모듈에 대한 신뢰도를 구할 수 있다.

3. Category 13 - 소프트웨어 개발단계를 포함한 직접적인 신뢰도 평가

이 category에 포함되는 방법은 비교적 최근에 LLNL(Lawrence Livermore National Laboratory) 및 미 공군 연구소인 ROME에서 제안된 것들로서, category 11 및 12가 오직 개발된 소프트웨어만을 대상으로 신뢰도를 평가하는데 반해, 소프트웨어 뿐 아니라 소프트웨어 개발단계에서의 품질(quality)이나 노력(efforts)등을 모두 포함하여 소프트웨어의 신뢰도를 평가하는 것이 바람직할 것이라는 생각에서 출발하고 있다. 이때, 소프트웨어의 개발단계를 어떤 방법으로 신뢰도 평가에 포함시키느냐에 따라 RADC 및 BBN(Bayesian Belief Network) 모델로 구분되고, 그림 3.5는 이러한 방법들이 소프트웨어 평가를 위해 고려하는 소프트웨어 life cycle의 범위를 보여준다.

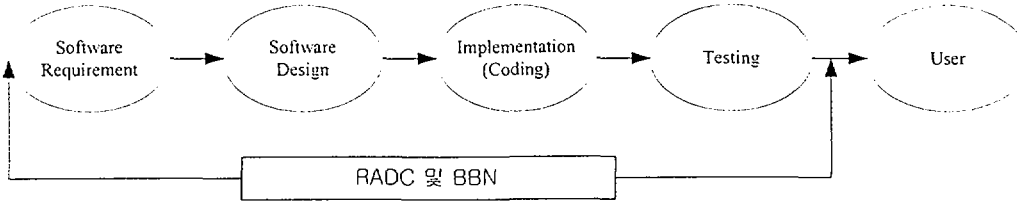


그림 3.5 RADC 및 BBN 모델에서 고려되는 소프트웨어 life cycle

가. RADC 모델

ROME에서 제시한 이 방법은, 소프트웨어의 신뢰도를 평가하기 위해, category 10에서 이미 소개한 multivariate approach와 유사한 접근을 시도하고 있다. 즉, 소프트웨어 life cycle에 포함된 설계요건, 설계, 구현, 시험 및 category 1~9에서 이미 소개된 소프트웨어의 평가에 사용되는 다양한 평가척도/방법들에서 얻어진 결과를 모두 조합하여 최종적인 소프트웨어 신뢰도를 예측하게 된다. 여기에서 multivariate approach에서는 소프트웨어 개발의 모든 단계를 고려하지 않는 반면 RADC에서는 모든 단계가 고려된다는 차이가 있다.

그림 3.6은 RADC 모델에 포함된 소프트웨어 평가척도들 및 이들을 사용하여 소프트웨어 신뢰도를 얻는 과정을 개략적으로 보여준다.

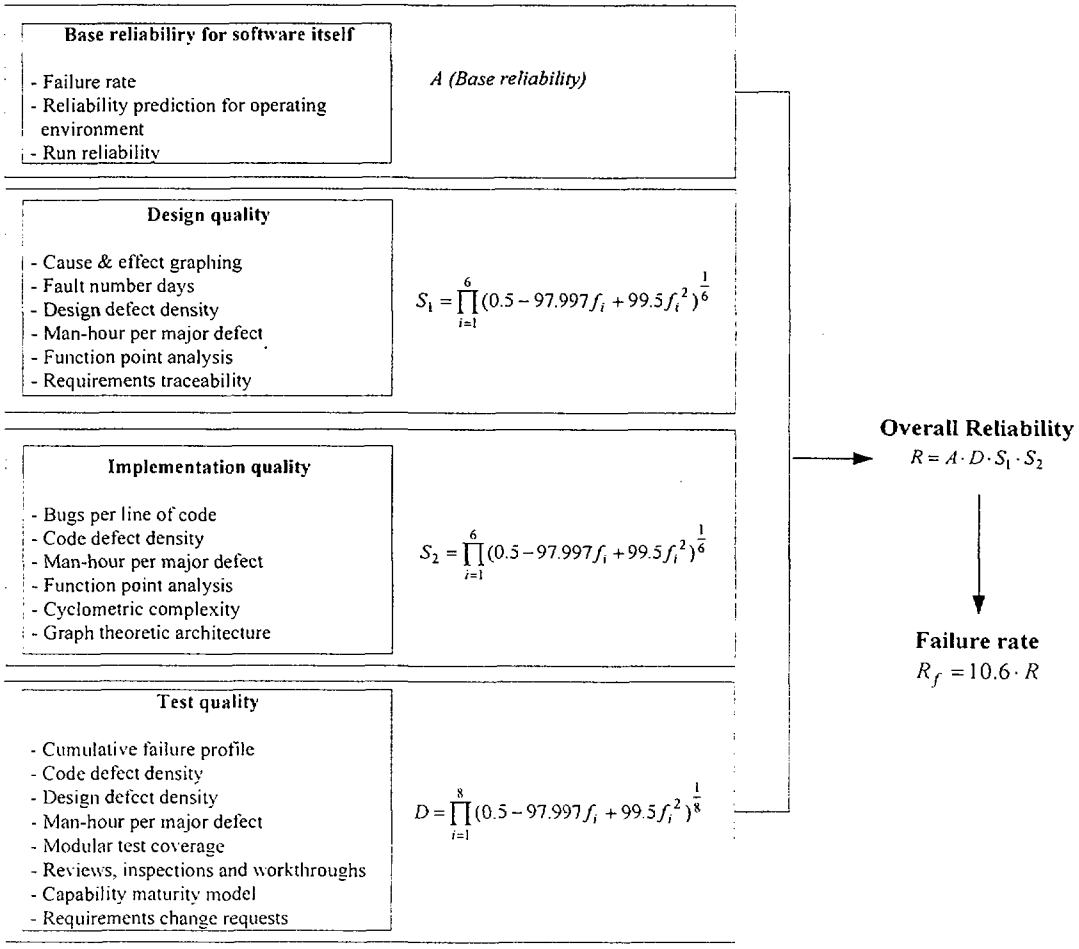


그림 3.6 RADC 모델을 통한 소프트웨어 신뢰도 평가

나. BBN 모델

이 방법은 소프트웨어 신뢰도를 평가하기 위해 소프트웨어 life cycle에 포함된 부분들을 고려한다는 것은 동일하지만, RADC와는 달리 BBN (Bayesian Belief Network)을 사용하여 causal network를 구성한 후, 이를 평가하는 형태의 접근을 취하고 있다. BBN은 일종의 네트웍 모델(network model)로서, 결과에 영향을 주는 인자들이 매우 많고 그 인자들간의 인과관계(causality)가 매우 복잡할 때 인자들간의 인과관계를 수학적인 방법(Bayesian Theory)를 써서 가장 합리적인 결론을 내릴 수 있도록 구성되어 있기 때문에, 진단(diagnosis) 및 의사결정(decision-making) 분야 등에 자주 적용되는 방법이다.

BBN 모델은 결과에 영향을 주는 인자를 표현하는 노드(node)와 각각의 노드의 특성/특징에 영향을 주는 인자들을 연결선(arc)들로 연결한 후, 이들에 대한 조건부 확률을 할당하여 최종적으로 원하는 결과를 구하게 된다.

BBN에 대한 이해를 돕기 위해, 다음과 같은 간단한 예제를 생각해 볼 수 있다. 사과나무를 기르는 어떤 과수원을 가정해 보자. 이때 사과나무를 오랫동안 길러온 경험에 따르면, 사과나무가 병들었거나 오랫동안 비가 오지않아 건조한 것할 경우 사과나무의 잎이 떨어지는 것으로 알려져 있다. 또한 병든 경우와 건조한 경우가 사과나무의 잎이 떨어지는 것에 얼마나 영향을 주는지도 경험을 통해 표 3.13과 같이 표현할 수 있다고 가정하자.

표 3.13 경험에 의한 조건부 확률 할당, p(Loses | Sick & Dry)

	Dry = "yes"		Dry = "no"	
	Sick = "yes"	Sick = "no"	Sick = "yes"	Sick = "no"
Loses = "yes"	0.95	0.85	0.90	0.02
Loses = "no"	0.05	0.15	0.10	0.98

이럴 경우, 사과나무의 잎이 떨어지는 사건에 대한 BBN 모델은 그림 3.7과 같은 형태로 표현될 수 있다.

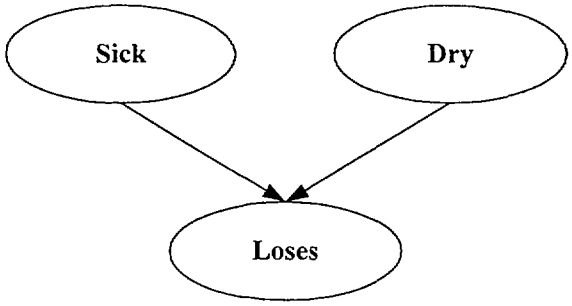


그림 3.7 BBN 모델의 예

이렇게 표현된 BBN 모델에서는 크게 3개의 인자(Sick, Dry 및 Loses)가 있고 각각의 인자에 대해 발생가능한 상태(state)는 모두 "yes" 또는 "no" 의 두 가지 뿐이다. 또한 "Loses" 노드의 상태는 "Sick" 와 "Dry" 노드의 조건에 따라 영향을 받게 되기 때문에(조건부 확률), 결국 초기 상태(Sick 또는 Dry가 yes/no일 확률)만 안다면 최종적으로 사과나무의 잎이 떨어질 확률을 구할 수 있게 된다.

이러한 BBN 모델을 바탕으로, LLNL에서 소프트웨어 신뢰도 평가를 위해 현재 개발중인 BBN 모델의 일부는 그림 3.8과 같다. 그림 3.8에서, 각각의 하위 노드(node)는 상위 노드의 특성/특징에 영향을 주는 인자들을 의미하고 각각의 연결선(arc)들은 노드들 간의 연관관계를 나타내고 있다. 예를 들어, 소프트웨어 개발 품질(development process quality)에는 소프트웨어 설계 및 소프트웨어 구현에 대한 품질이 영향을 준다고 모델링 되어 있고, 소프트웨어 설계 및 구현 노드는 다시 다양한 하위노드들(세부 영향인자)을 포함하게 된다. 또한 각각의 연결선들은 하위노드들이 상위노드들의 특성/특징에 얼마나 영향을 주는지를 확률 또는 가중치 형태로 가지고 있기 때문에, 각 하위노드들의 변화들은 결국 최상위 노드인 소프트웨어 신뢰도에 영향을 주게 된다. 따라서 각 세부 영향인자들 간의 가중치를 모두 알 수 있다면 모든 영향인자들을 고려하여 소프트웨어 신뢰도를 평가할 수 있게 된다.

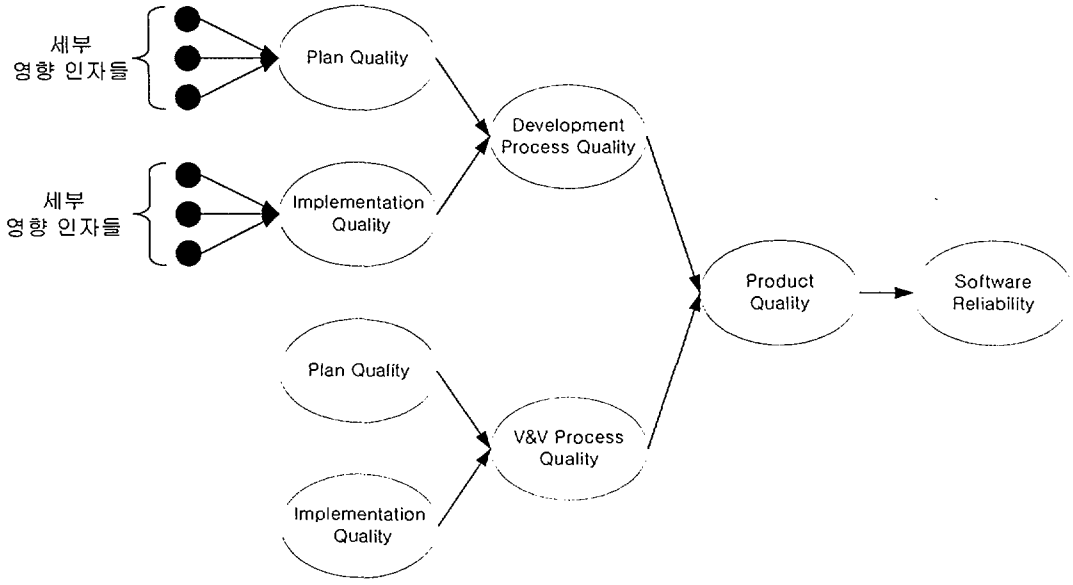


그림 3.8 소프트웨어 신뢰도 평가를 위한 BBN 모델링의 예

제 3 절 소프트웨어 신뢰도 평가 방법들의 문제점

지금까지 살펴본 소프트웨어 신뢰도 평가 방법들은 1960년대부터 지금까지 꾸준히 제안 및 개발되어 온 것들로서, 실제적으로 소프트웨어 개발분야에서 사용되는 방법들도 있지만 대부분의 평가 방법들은 그들이 가지고 있는 자체적인 한계 또는 문제점들로 인해 현재는 사용되지 못하고 있다. 다음은 지금까지 소개된 소프트웨어 신뢰도 평가 방법들에 대한 문제점들은 개략적으로 정리한 것이다.

1. 간접적인 평가 방법론의 문제점

앞에서 이미 설명한 바와 같이, 간접적인 소프트웨어 신뢰도 평가 방법론은 소프트웨어 설계요건, 설계, 구현 등의 단계에 대한 품질평가를 통해 소프트웨어의 신뢰도를 간접적으로 예측하는 방법이다. 이때 이러한 접근방법의 기본 가정은 소프트웨어 신뢰도는 각 개발단계에 의해 영향을 받는다는 것이다. 그러나 기존에 개발/제안된 방법론들은 모두 소프트웨어 개발단계 전체를 고려하기 보다는 소프트웨어 life cycle에 포함된 각 단계만을 고려한 평가를 할 수 있도록 되어 있다. 예를 들어, 소프트웨어 구현의 적합성을 평가하는 방법론들(category 1~4)은 오직 구현단계만 평가할 수 있고, 소프트웨어 설계요건의 적합성을 평가하는 방법론들(category 5)은 오직 소프트웨어의 설계요건만 평가할 수 있으며, 소프트웨어 설계 품질을 평가하는 방법론(category 6)은 오직 설계 단계만을 평가할 수 있다.

따라서 비록 평가를 수행한 부분에서 좋은 결과를 얻었다 할 지라도, 이 결과를 가지고 소프트웨어 신뢰도가 좋다라는 것을 확신하기에는 부적절하다. 물론 이러한 단점을 극복하기 위해 multivariate approach(category 10)가 도입되기는 하였지만, 이 방법 역시 단점을 가지고 있다. 즉, multivariate approach는 소프트웨어 신뢰도 평가 방법들이 소프트웨어 신뢰도에 영향을 주는 각각의 인자에 대해서만 평가할 수 있기 때문에, 소프트웨어 신뢰도에 영향을 주는 모든 인자들을 factor analysis나 principal component analysis 등을 사용해 구한 가중치(weighting factor)를 통해 다음과 같은 형태로 평가하게 된다.

$$\text{소프트웨어 신뢰도} \approx \sum_{i=1}^{n} a_i RM_i$$

a_i = i번째 평가척도에 대한 가중치

RM_i = i번째 소프트웨어 신뢰도 평가척도

따라서 이 방법은 소프트웨어 신뢰도에 영향을 주는 몇 가지 인자들을 통합하여 종합적으로 신뢰도를 평가할 수 있지만, 좋지 않은 평가 결과가 나왔을 때 과연 이것이 어떤 인자(설계, 구현 및 requirement 등)로 인해 이렇게 된 것인지를 명확히 알 수 없기 때문에, 이 평가 결과를 소프트웨어 설계 및 구현 등에 적용하는 것은 거의 불가능하다는 것이 알려져 있다.

이 외에, 소프트웨어의 시험을 통한 평가 방법론들(category 7 ~ 9)에 대한 가장 큰 문제점으로 지적되는 사항은 소프트웨어에 포함된 모든 내용에 대한 평가가 불가능하다는 점이다.

예를 들어 세금을 계산해주는 소프트웨어가 있다고 가정해 보자. 이때 소프트웨어에는 세금을 계산하기 위한 많은 계산식(equation)들을 프로그래밍하는 단계가 포함되기 때문에 소프트웨어의 고장은 프로그래밍 도중 포함될 수 있는 인적오류로 인해 발생될 수 있지만, 원래 틀린 계산식의 사용이 원인이 되어 발생할 수도 있다. 그러나 소프트웨어 시험을 통한 평가 방법들의 경우 소프트웨어의 내용에 대한 평가는 가능하지만 계산식들 자체에 대한 평가는 불가능하기 때문에, 평가 결과로 얻어진 소프트웨어 신뢰도 값 자체에 대한 신뢰성이 떨어질 수 있다.

지금까지 설명한 것들 외에, 다양한 문제점들이 간접적인 소프트웨어 신뢰도 평가 방법론들에 대해 제기되고 있고, 표 3.14는 이들 중 대표적인 것을 정리하여 보여준다.

표 3.14 간접적인 소프트웨어 평가 방법에 대한 주요 문제점

평가를 위한 기본 가정	문제점
소프트웨어 신뢰도는 포함된 결함 들의 수가 증가할수록 감소한다.	소프트웨어에 포함된 결함의 수 만으로 소프트웨어 신뢰도를 예측할 수 없다. 그 이유로는 o 과거의 경험을 바탕으로 판단해 볼 때, 많은 수의 결함들을 제거했다 하더라도 이것이 소프트웨어 신뢰도의 개선을 반드시 의미하지는 않는다. 즉, 주어진 시험기간동안에 관찰된 소프트웨어의 고장들은 소프트웨어가 포함하고 있는 결함들의 극히 일부만이 원인이 되어 발생된 것이다. 따라서 대부분의 결함들에 의한 고장은 test 기간동안에는 나타나지 않는다 o 만일 정확한 결함 수를 알아도, 소프트웨어가 실제로는 어떻게 동작될 지 알 수 없다. 그 이유는 포함된 결함들의 심각도(seriousness)를 결정하기 힘들고, 소프트웨어는 다양한 사용자들이 사용하기 때문에 어떤 결함이 실제로 고장을 일으켰는지 알 수 없기 때문이다.
소프트웨어에 포함된 결함은 소프트웨어의 크기 또는 복잡도와 직접적인 관련이 있다.	o 과거의 경험에서 소프트웨어의 크기나 복잡도는 결함 발생에 많은 영향을 준다는 것이 밝혀지기는 했지만, 이들은 소프트웨어 신뢰도에 영향을 주는 한가지 인자일 뿐이기 때문에, 이들간의 관계식을 통해 소프트웨어 신뢰도를 예측할 수는 없다. 이러한 사실은 통계적으로 신장과 IQ가 비례한다고 해서, 신장으로 IQ를 예측하는 것은 불합리하다는 것과 동일한 문제점을 갖는다. o 소프트웨어의 설계요건에 포함된 결함들의 경우, 이 방법으로는 잡아낼 수 없다.

2. 직접적인 소프트웨어 평가 방법론의 문제점

이미 설명한 바와 같이, 직접적인 소프트웨어 평가 방법들은 여러 번 시험을 수행하여 얻은 소프트웨어 고장 자료를 확률/통계적으로 처리하거나 수학적 모델링 기법을 사용하여 소프트웨어의 신뢰도를 예측하는 접근을 취하고 있다. 여기에서, 현재 개발중인 category 13에 포함된 방법들을 제외하고, 소프트웨어 자체를 평가하고자 하는 방법들(category 11 및 12)에 대한 문제점들은 비교적 명확한 편이고, 다

음은 제기된 문제점 들 중 대표적인 것들을 정리한 것이다.

가. 소프트웨어 시험을 통한 신뢰도 평가

소프트웨어의 시험을 통해 신뢰도를 평가하기 위한 기본적인 가정 및 문제점은 다음과 같다.

기본 가정
소프트웨어 시험을 통해 얻은 결과들을 확률 및 통계적 기법으로 처리하여 소프트웨어 신뢰도를 예측할 수 있다

○ 문제점

1. 소프트웨어 고장을 무작위성 고장(random failure)으로 볼 수 없다는 주장이 제기되어 있다.
2. Data quality 문제
   - 부적절한 operational profile 설정
   - 얻어진 자료(data point)의 가공

     (부적절한 자료의 제거 및 평균치 사용 등)
3. 자료 분석 (자료의 fitting을 위한 확률 및 통계모델 선택)
   - 잘못된 모델의 선정
   - 모델에 포함되는 모수(parameter)들의 잘못된 선정
4. V&V와의 연계성
   - 자료를 얻기 위해 수행된 시험의 적절성이 강조된다. (만일 부적절한 시험이 수행되어서 얻어진 고장 자료가 별로 없는 상태에서 분석을 수행한다면, 상대적으로 높은 신뢰도가 예측될 수 있다. 반면 많은 시험을 통해 충분한 자료가 확보된 경우, 실제적으로는 높은 신뢰도를 가질 수 있지만, 분석결과는 낮은 신뢰도로 예측될 수 있다. 따라서 시험을 자세히 하면 할수록 낮은 신뢰도가 얻어지는 현상이 발생할 수 있다.
5. 원하는 신뢰도 값이 높으면 높을 수록 수행해야 하는 시험횟수 및 시간이 증가한다. 기존의 연구결과에 의하면, 소프트웨어 시험에 대해 다음과 같은 식이

제안되어 있다 [31].

$$u = \frac{\ln(1-C)}{\ln(1-p)}$$

여기에서, u = 목적값을 증명하기 위해 요구되는 시험횟수

C = 확신도 (confidence level)

p = 목적값 (target reliability)

예를 들어, 90%의 확신도로 10^{-7} 정도의 신뢰도를 검증하기 위해서는 약 23,000,000번의 시험이 수행되어야 하고, 만일 한번의 시험당 소요시간을 1분이라고 가정할 경우 총 소요시간은 약 44년이 된다.

나. 소프트웨어 모델링을 통한 신뢰도 평가

소프트웨어의 모델링을 통한 신뢰도 평가시 고려되는 기본 가정 및 문제점들은 다음과 같다.

기본 가정
소프트웨어를 수학적으로 모델링한 후 모델에 대한 분석을 통해 소프트웨어의 신뢰도를 예측할 수 있다

○ 문제점

1. 부적절한 모델링 범위: 소프트웨어 모델링하기 위해 제안된 방법들은 작고 간단한 경우에는 적용이 쉽지만, 크고 복잡한 경우에는 적용하기 힘들다. (즉 너무 많은 시간과 노력 및 비용이 소모된다)
2. 부적절한 모델링 내용: 모델링은 소프트웨어에 문제가 있는 부분을 쉽게 판별할 수 있도록 해야 하지만, 대부분의 경우 소프트웨어 자체의 표현에 그치고 있다.

다. 직접적인 소프트웨어 신뢰도 평가를 위해 새롭게 제안되고 있는 방법

앞에서 설명된 문제점들을 해결하기 위해 현재 새롭게 제안되고 있는 방법들은 category 13에 포함되는 RADC 및 BBN 방법이고, 이 외에 인공신경망(artificial neural network)을 사용한 소프트웨어 신뢰도 평가 방법이 제안되었다. 그러나 인공신경망을 사용한 소프트웨어 신뢰도 평가는 평가 방법 자체가 아직 확실치 않고, 많은 소프트웨어의 시험 결과가 필요하다는 점에서 제기된 문제점을 아직 해결하지 못한 방법으로 판단되어 방법론에 대한 자세한 설명은 생략하기로 한다.

Category 13에 포함되는 RADC 및 BBN 방법은 많은 장점들을 가지고 있다. 예를 들어 RADC 은 모두 소프트웨어 life cycle 전체를 평가대상으로 삼는다는 점과 사용하기가 상대적으로 쉽다는 장점이 있다. 그러나 이 방법은 category 10에서 소개된 multivariate approach와 유사한 접근방법을 취하고 있기 때문에 평가결과에 대한 해석이 불확실해질 수 있고, 또한 소프트웨어의 고장율 R_f를 단순한 변환식(즉, $R_f = 10.6 \times R$)을 사용하여 구하고 있기 때문에 수학적인 배경이 뒷받침되지 않는다는 단점을 가지고 있다.

이에 반해 BBN 방법은 명확한 수학적인 배경(조건부 확률) 을 가지고 있는 장점이 있다. 그러나 네트웍 모델의 특성상 소프트웨어의 신뢰도에 영향을 주는 인자들을 자세히 모델링 할 경우 모델자체가 매우 복잡해 질 뿐 아니라 모든 인자들간의 인과관계 및 영향의 정도가 명확하지 않은 상태에서 각 인자들간의 조건부 확률을 모두 할당해야 한다는 것이 문제점으로 지적되고 있다. 또한 일단 모델을 구성했다 하더라도 모델에 의해 예측된 값과 실제 관찰된 값들의 차이를 보정(calibration) 하기 위해서는 최소한 몇 년간의 시간이 필요하다라는 점도 고려해 볼 때 [32], 실제적으로 사용되기에는 아직 많은 문제가 있다고 판단된다.

제 4 장 소프트웨어 평가 방법론 조사 결과 및 결론

지금까지 설명한 소프트웨어 평가 방법들이 가지고 있는 문제점들은 각 category 별로 표 4.1과 같이 정리될 수 있다.

표 4.1 소프트웨어 평가 방법론들에 대한 category별 문제점 정리

Category	평가 방법	문제점
1 ~ 10	간접적인 평가 방법	o 소프트웨어 life-cycle을 전체적으로 고려하지 못함 o 결함의 수 만으로는 소프트웨어 신뢰도를 예측할 수 없음 o 소프트웨어 복잡도가 신뢰도와 반드시 연계되는 것은 아님 o 잘못 설정된 소프트웨어 설계요건들에 의한 결함들은 고려 할 수 없음
11 ~ 12	직접적인 평가 방법	o 소프트웨어 life-cycle을 전체적으로 고려할 수 없음 o 소프트웨어의 고장을 무작위성 고장으로 볼 수 없음 o 평가에 사용되는 자료들의 신뢰성 문제 o 자료 분석을 위한 확률적 모델의 선택 문제 o 자료 수집을 위해 수행된 시험의 신뢰성 문제 o 원하는 신뢰도 수준이 높을 경우, 자료수집에 필요한 시험횟수 및 시간이 증가 o 부적절한 모델링 범위 o 부적절한 모델링 내용
13		o 평가 결과에 대한 해석이 불확실 o 수학적 배경 결여 o 모델링을 위해 많은 노력이 요구됨 o 모델의 보정(calibration)에 많은 시간과 노력이 필요함

표 4.1에 나타난 바와 같이 현재까지 제안되거나 개발된 많은 소프트웨어 평가 방법들은 나름대로의 문제점들을 가지고 있고, 아직까지 검증된 방법이 없으며 [17], 소프트웨어 life-cycle에 대해 각 부분에만 적용되도록 구성되어 있을 뿐 아니라 이러한 평가 방법들을 사용해서 얻은 결과들도 소프트웨어 신뢰도로서 직접적으로 사용하는 것도 어렵기 때문에, PSA 수행에 있어서 요구되는 체계적이며 정량적인 평가 방법은 아직 결정될 수 없다고 판단된다.

또한 요즈음 새롭게 제안되고 있는, 미국 NRC 주도로 LLNL에서 수행하고 있는 디지털 계측제어 시스템의 정량적인 평가방법 개발에서 제안된 BBN이나 RADC 방법들은 소프트웨어 life-cycle 전반에 걸친 평가가 가능하도록 개발되기는 하지만, 이

들 역시 PSA에서 요구하는 정도의 객관성을 유지하고 있다고 볼 수 없기 때문에, 이 분야에 대한 추가적인 연구가 수행되어야 한다고 판단된다.

그러나 만일 현재까지 개발된 방법들을 통해 소프트웨어의 신뢰도를 평가해야 한다면 다음과 같은 접근 방법이 가장 타당할 것으로 생각된다. 즉 소프트웨어 신뢰도 평가에 있어서 현실적으로 사용 가능한 방법은 소프트웨어의 시험을 통한 평가이지만, 이 방법은 소프트웨어의 고장을 무작위성 고장으로 볼 수 없다는 점이 가장 큰 문제로 지적되고 있다. 다시 말하면, 소프트웨어 life cycle에 있어서 설계요건, 설계 및 소프트웨어 구현 과정중에 포함될 수 있는 결함들은 인적오류에 기인하고 이들은 모두 결정론적인(deterministic) 특성을 가지고 있기 때문에 확률 및 통계적인 기법을 써서 소프트웨어의 신뢰도를 평가할 수 없다는 주장이다. 그러나 이러한 결정론적인 오류들을 어느 정도 제거할 수 있다면 소프트웨어의 고장을 무작위성 고장으로 고려해도 큰 문제가 없을 것으로 가정할 수 있을 것이다 [33]. 따라서 시험을 통한 소프트웨어 신뢰도 평가의 전 단계로서 소프트웨어에 포함된 결정론적인 오류들을 제거할 수 있는 방법을 개발하거나 기존에 제안된 방법들(category 1 ~ 10)을 적절히 이용한다면, 확률 및 통계 기법을 써서 소프트웨어의 신뢰도를 평가하는 것이 가능할 것이라고 판단된다.

참고 문헌

[1] D. L. Parnas, G. J. K. Asmis and J. Madey, Assessment of Safety- Critical Software in Nuclear Power Plants, Nuclear Safety Vol. 32, No. 2, pp. 189-198, 1991.

[2] D. Welbourne, Safety Critical Software in Nuclear Power, The GEC Journal of Technology, Vol. 14, No. 1, pp. 33-40, 1997.

[3] J. P. Burel, The Use of Digital Technology for Protection and Safety Applications at French Nuclear Reactors, Kerntechnik, Vol. 60, No. 5-6, pp. 220-224, 1995.

[4] T. Graae and L. Engdahl, The Reliability of the Software of the Digital Control System Nuclear Advantage, Kerntechnik, Vol. 61, No. 5-6, pp. 236-238, 1996.

[5] H. W. Bock and A. Graf, Reliability Aspects of Computer-based Safety Systems for Nuclear Power Plants, Kerntechnik, Vol. 60, No. 5-6, pp. 207-214, 1995.

[6] H. Hofmann and H. J. Sauer, Effect of Fieldbus Technology on Digital Instrumentation and Control for Nuclear Power Plants, Kerntechnik, Vol. 60, No. 5-6, pp. 245-247, 1995.

[7] U. Kunze and V. Streicher, Advanced Monitoring Systems for Preventive Maintenance of Mechanical Systems and Components, Kerntechnik, Vol. 60, No. 5-6, pp. 238-241, 1995.

[8] 오성현, 원전 디지털 계측제어시스템 안전성 평가방향, 원자력 안전, 통권 11호, pp. 58-67, 1999.

[9] 김복렬, 소프트웨어 신뢰도 및 심층방어설계 안전성 평가, 원자력 안전, 통권 11호, pp. 68-75, 1999.

[10] 황희수, 디지털 상용등급제품 인정관련 안전성 평가, 원자력 안전, 통권 11호, pp. 76-83, 1999.

[11] USNRC, Digital Instrumentation and Control Systems in Nuclear Power Plants-Safety and Reliability Issues, Final Report, National Academy Press, Washington D.C., 1997.

[12] COOPRA working document, What PRA Needs From A Digital I&C Systems Analysis: An Opinion, www.coopra.org, 1999.

[13] NEA/CSNI/R(97)23, Operating and Maintenance Experience with Computer-based Systems in Nuclear Power Plants, Nuclear Energy Agency, Committee on the Safety of Nuclear Installations, September, 1998.

[14] N. M. Ichiyen and P. K. Joannou, Safety Critical Software Design Approaches Developed for Canadian Nuclear Power Plants, Kerntechnik, Vol. 60, No. 5-6, pp. 232-237, 1995.

[15] Bev Littlewood and Lorenzo Strigini, Validation of Ultrahigh Dependability for Software-based Systems, Comunications of the ACM, Vol. 36, No. 11, pp. 69-80, 1993.

[16] ANSI/IEEE, Standard Glossary of Software Engineering Terminology, STD-729-1991, ANSI/IEEE, 1991.

[17] S. L. Pfleeger, Measuring Software Reliability, IEEE Spectrum, Vol. 29, Iss. 8, pp. 56-60, 1992.

[18] IEEE, Charter and Organization of the Software Reliability Engineering Committee, 1995.

[19] Michael R. Lyu, et al., Handbook of Software Reliability Engineering, IEEE Computer Society Press, 1995.

[20] IEEE Std. 982.1, IEEE Standard Dictionary of Measures to Produce Reliable Software, 1988.

[21] IEEE Std. 982.2, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, 1988.

[22] Ronald E. Prather, Design and analysis of hierarchical software metrics, ACM Computing Surveys, Vol. 27, No. 4, pp. 497-517, 1995.

[23] Norman Fenton and Martin Neil, A Critique of Software Defect Prediction Models, to appear IEEE Transactions on Software Engineering, 1999 (http://www.csr.city.ac.uk/papers/1999.html).

[24] Alan Wood, Predicting Software Reliability, IEEE Computer, pp. 69-77, November 1996.

[25] Norman F. Schneidewind, Software Reliability Model with Optimal Selection of Failure Data, IEEE Transactions on Software Engineering, Vol. 19, No. 11, pp. 1095-1104, 1993.

[26] FT. popentiu and D. N. Boros, Software Reliability Growth Supermodels, Microelectronics and Reliability, Vol. 36, No. 4, pp. 485-491, 1996.

[27] Bev Littlewood and David Wright, Some Conservative Stopping Rules for the Operational Testing of Safety-Critical Software, IEEE Transactions on Software Engineering, Vol. 23, No. 11, pp. 673-683, 1997.

[28] Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M.

Nicol, Branson W. Murrill and Jeffrey M. Voas, Estimating the Probability of Filure when Testing Reveals No Failures, IEEE Transactions on Software Engineering, Vol. 18, No. 1, pp. 33-42, 1992.

[29] Jeff Tian, Integrating Time Domain and Input Domain Analyses of Software Reliability using Tree-Based Models, IEEE Transactions on Software Engineering, Vol. 21, No. 12, pp. 945-958, 1995.

[30] Walter J. Gutjahr, Optimal Test Distributions for Software Failure Cost Estimation, IEEE Transactions on Software Engineering, Vol. 21, No. 3, pp. 219-228, 1995.

[31] Paul E. Ammann, Susan S. Brilliant and John C. Knight, "The effect of Imperfect Error Detection on Reliability Assessment via Life Testing," IEEE Transactions on Software Engineering, Vol. 20, No. 2, pp. 142-148, 1994.

[32] Marc Bouissou, Assessment of a Safety-Critical System Including Software: A Bayesian Belief Networks for Evidence Sources, Edf, 1999.

[33] Jeffrey M. Voas and Keith W. Miller, Software Testability: The New Verification, IEEE Software, Vol. 12, No. 3, pp. 17-28, 1995.

[34] Han S. Son and Poong H. Seong, Quantitative evaluation of safety-critical software at the early development stage: an interposing logic system software example, Reliability Engineering and System Safety, Vol. 50, No. 3, pp. 261-269, 1995.

[35] An entropy-based meaure of software complexity, IEEE Transactions on Software Engineering, Vol.18, No. 11, pp. 1025-1029, 1992.

[36] G. J. Mayers, Reliable software through composite design, Petrocelli, 1975.

[37] Barry W. Bohem, Software engineering economics, Prentice-Hill, 1981.

[38] Wendy W. Peng and Dolores R. Wallace, Software error analysis, NIST Special Publication 500-209, April 1993.

[39] David N. Card and Robert L. Glass, Measuring software design quality, Prentice-Hall, 1990.

[40] A. Mowshowitz, Entropy and the complexity of graphs: I. An index of the relative complexity of a graph, Bulletin of Mathematical Biophysics, Vol. 30, pp. 175-204, 1968.

[41] K. S. Lew et al., Software complexity and its impact on software reliability, IEEE Transactions on Software Engineering, Vol. 14, No. 11, pp. 1645-1655, 1988.

[42] M. R. Woodward et al., A measure of control flow complexity in program text, IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, pp. 45-50, 1979.

[43] J. McCall, et. al, Methodology for software reliability prediction, RADC-TR-87-171, Volume I and II, Rome Air Development Center, 1987.

[44] George F. Watson, MIL Reliability: A New Approach, IEEE Spectrum, Vol. 29, Iss. 8, pp. 46-49, 1992.

[45] Jeffrey M. Voas, PIE: A Dynamic Failure-Based Technique, IEEE Transactions on Software Engineering, Vol. 18, No. 8, pp. 717-727, 1992.

부록 – 소프트웨어 신뢰도 평가 방법론

본 부록에서는 소프트웨어 신뢰도를 평가하기 위해 현재까지 제안된 방법론들을 개략적으로 정리하였다. 이러한 방법론들에 대한 기본적인 참고자료는 [20] 및 [21]이고, 이 외의 자료들은 각 방법론을 설명할 때 따로 표시하였다.

A. 소프트웨어의 구조적 복잡도 평가 방법

A.1 Operational or Functional Complexity

복잡도 (complexity)는 소프트웨어의 신뢰도를 간접적으로 표현하는데 유용할 수 있는 개념이고, 이러한 복잡도는 주로 엔트로피 (entropy)를 통해 정량화될 수 있다. 이때 소프트웨어 복잡도 평가에 사용되는 엔트로피는 Shannon이 제안한 정보이론 (information theory)에 근거를 둔 것으로, 다음과 같이 정의된다.

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

p_i = 시스템이 i 번째 상태(state)에 있을 확률

이러한 엔트로피의 정의를 바탕으로 현재까지 4개의 소프트웨어 복잡도 평가척도(functional complexity, interface complexity, micro complexity 및 operator complexity) 가 제안되어 있고 [34, 35], 이들의 정의는 표 A.1 과 같다.

표 A.1 엔트로피를 사용한 복잡도 평가 방법

Functional complexity = $-\sum_{i=1}^{k} p_i \log_2(p_i)$	k = 소프트웨어에 포함된 function 수 p_i = i 번째 function 이 수행될 확률 n = 소프트웨어에 포함된 모듈 수 f_i = 서로 coupling 된 모듈 수 s_i = 서로 cohesion 된 모듈 수 m = 사용된 서로 다른 operator 수 N = 소프트웨어에 사용된 총 operator 수 g_i = i 번째 operator 가 수행된 수
Interface complexity = $-\sum_{i=1}^{l} \frac{f_i}{n} \log_2(\frac{f_i}{n})$	
Micro complexity = $-\sum_{i=1}^{l'} \frac{s_i}{n} \log_2(\frac{s_i}{n})$	
Operator complexity = $-\sum_{i=1}^{m} \frac{g_i}{N} \log_2(\frac{g_i}{N})$	

A.2 BVA 모델

이 평가 척도는 소프트웨어에 포함된 서로 다른 모듈들에 대한 coupling을 평가하기 위해 제안된 것으로, 시험을 통한 소프트웨어 신뢰도를 평가할 때 적정 수준의 시험 수를 구하거나 논리적인 소프트웨어 결함 들을 찾기 위해 사용될 수 있다. BVA 방법의 가장 큰 특징은 계산에 사용되는 알고리즘 (algorithm)이 명확하기 때문에 쉽게 사용될 수 있다는 점이다. 즉 각 프로그래밍 언어에 사용된 변수들에 대해 표 A.2 와 같은 값들을 준 후, 이들을 모두 더해서 소프트웨어의 복잡도를 평가하게 된다.

A.2 BVA 모델에 사용되는 값

변수 종류	프로그래밍 언어	BVA 값
integer	C, Ada	5
float	C, Ada	5
double	C, Ada	5
long (integer)	C, Ada	5
Boolean	C, Ada	2
Natural	Ada	3
character	C, Ada	2
unsigned integer	C	3

A.3 Number of Entries and Exits per Module

이 방법은 소프트웨어의 architecture 의 복잡도를 결정하기 위해, 소프트웨어에 포함된 모듈 들의 entry/exit point 들을 고려하는 방법이다. 즉, i 번째 모듈에 대해 entry point 의 수가 e_i 개이고 exit point 의 수가 x_i 개이면 i 번째 모듈의 복잡도 m_i 는

$$m_i = e_i + x_i$$

로 결정된다. 이 방법은 모듈에 포함된 이러한 point 들은 모두 function 들과 관계가 있고, 포함된 function 들의 수가 증가할수록 프로그램의 복잡도도 증가할 것이라는 간단한 가정을 바탕으로 하고 있다. 또한 각 모듈에 대해 포함될 수 있는 최대 function 수는 5 개가 적절하다고 제안하고 있다.

A.4 Design Structure

이 방법은 소프트웨어의 설계에 대한 복잡도를 평가하기 위해, 다음과 같은 계산 방법을 제안하고 있다.

P1 = 프로그램에 포함된 모듈의 총 수

P2 = 프로그램의 입력 및 출력에 관계된 모듈의 총 수

P3 = 프로그램에 의해 수행된 prior processing 에 관계되는 모듈의 총 수

P4 = 데이터베이스에 포함된 element 들의 수

P5 = 데이터베이스 element 들 중 unique 하지 않은 element 들의 수

P6 = 데이터베이스의 segment 수

P7 = 프로그램에 포함된 모듈 중 1 개의 entry/exit point 를 가지지 않는 모듈 들의 수

D1 = 프로그램이 top-down 형태로 잘 구성되었는가? (그렇다면 1, 그렇지 않으면 0 을 할당한다.)

D2 = Module dependence = P2/P1

D3 = Module dependent on prior process = P3/P1

D4 = Database size = P5/P4

D5 = Database compartmentalization = P6/P4

D6 = Module single entry/exit = P7/P1

이러한 값들을 통해 design structure measure (DSM) 은 다음과 같이 표현된다.

$$DSM = \sum_{i=1}^{6} w_i D_i, \quad \sum_{i=1}^{6} w_i = 1.0$$

여기에서 w_i는 가중인자(weighting factor)로서 사용자가 각 값의 중요도에 따라 할당하게 된다.

A.5 Data Flow Complexity

이 평가 척도는 대규모 소프트웨어의 자료구조나 모듈 간의 자료흐름 및 모듈간의 상호작용을 평가하는데 사용할 수 있고, 개략적인 평가 방법은 다음과 같다.

lfi = 모듈로 들어가는 local data flow

datain = 모듈에서 사용되는 자료 (구조)의 수

ifo = 모듈에서 나가는 local data flow

dataout = 모듈에서 바뀌는 자료 (구조)의 수

length = 모듈에 포함되어 있는 코드의 라인 수 (주석문은 제외)

이 값들을 사용한 information flow complexity (IFC)는 다음과 같은 방법으로 계산된다.

$$IFC = \{ fanin \times fanout \}^2 = \{ (lfi + datain)^2 \times (lfo + dataout)^2$$

A.6 Cohesion/Coupling

이 평가 방법은 Mayers[36]에 의해 제안되었고, 정량적인 평가보다는 소프트웨어 설계의 적절성을 정성적으로 평가하는데 목적을 두고 있다. 여기에서 "cohesion"은 하나의 모듈이 어떤 function을 수행하는데 얼마나 기여하는지를 의미하는 용어이고 "coupling"은 모듈간의 상호 작용을 의미하는 용어이다. 따라서 이들이 소프트웨어 설계에 적절치 못하게 반영될 경우 소프트웨어의 신뢰도가 떨어질 수 있다는 점에서 소프트웨어의 초기 설계단계에 적용될 수 있는 방법이고, 이들은 표 A.3과 같은 항목들을 고려하여 결정된다.

표 A.3 Cohesion 및 Coupling 결정 기준

Cohesion	Coupling
• Difficult to describe the module function • Module performs more than one function • Only one function performed per invocation • Each function has an entry point • Module performs related class of functions • Functions are related to problem procedure • All of the functions use the same data	• Direct reference between modules • Modules packed together • Some interface data external or global • Some interface data control information • Some interface data in data structure

A.7 Functional Complexity

이 방법은 Bohem[37]이 제안한 것으로서, 정량적인 평가보다는 소프트웨어 설계의 적절성을 정성적으로 평가하는데 목적을 두고 있다. 표 A.4는 소프트웨어의 functional complexity의 정도를 결정하는데 사용되는 기준을 보여준다.

표 A.4 Functional Complexity 결정 기준 (1/2)

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
Very low	Straight-line code with a few nonnested SP operators: Dos, CASEs, IFTHENELSEs. Simple predicates.	Evaluation of simple expressions, such as $a = b + c \cdot (d - e)$	Simple read, write statements with simple formats.	Simple arrays in main memory.
Low	Straightforward nesting of SP operators. Mostly simple predicates.	Evaluation of moderate-level expressions, such as $d = \sqrt{b^2 - 4ac}$	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET / PUT level. No cognizance of overlap.	Single file subsetting with no data structure changes, no edits, no intermediate files.
Nominal	Mostly simple nesting. Some inter-module control. Decision tables.	Use of standard math and statistical routines. Basic matrix / vector operations.	I/O processing includes device selection, status checking and error processing.	Multi-file input and single file output. Simple structural changes, simple edits.

표 A.4 Functional Complexity 결정 기준 (2/2)

Rating	Control Operations	Computational Operations	Device-dependent Operations	Data Management Operations
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable intermodule control.	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, round off concerns.	Operations at physical I/O level (physical storage address translations; seeks, reads, etc.). Optimized I/O overlap.	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level.
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling.	Difficult but structured numerical analysis: near-singular matrix equations, partial differential equations.	Routines for interrupt diagnosis, servicing, masking. Communication line handling.	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization.
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control.	Difficult and unstructured numerical analysis: highly accurate analysis of noisy, stochastic data.	Device timing-dependent coding, micro-programmed operations.	Highly coupled, dynamic relational structures. Natural language data management.

A.8 Data Structure Metrics [38]

이 방법은 모듈의 복잡도를 평가하는 것으로서, 만일 모듈이 복잡하다면 결함으로 인해 고장이 발생했을 때 이것을 쉽게 감지할 수 없다는 점을 해결하기 위한 방법으로 제안되었다. 모듈을 평가하기 위해서 제안된 평가 척도는 모두 3 개이고 이들은 다음과 같이 결정된다.

L_i = 모듈 안에 있는 i 번째 line

v_j = 모듈에서 사용된 j 번째 변수

N = 모듈에서 사용된 변수들의 총 수

LV_i = i 번째 line 을 수행할 경우 모듈내에서 영향을 받는 변수 (live variable) 들의 수

VS_j = j 번째 변수가 사용된 line 사이에 포함된 code line 수 (the number of lines of code between the first and last reference of the variable)

SP_{jk} = j 번째 변수가 영향을 미치는 code line 수 (the number of lines of code between the k_{th} and $(k+1)_{st}$ references to the variable)

이러한 값들을 사용하여 다음과 같은 복잡도 평가 척도를 계산하게 된다.

$$LV = \frac{1}{LOC}\sum_i LV_i \qquad LOC = code\text{의 } line \text{ 수}$$

$$VS = \frac{1}{N}\sum_j VS_j$$

$$SP = \frac{1}{N}\sum_j \sum_k SP_{jk}$$

A.9 System Design Complexity [39]

모듈의 구성이 복잡하다면 그렇지 않은 것에 비해 상대적으로 많은 결함들을 포함할 가능성이 높아진다고 생각할 수 있다. 따라서 이 평가 척도는 소프트웨어에 포함된 모듈들의 복잡도를 평가하기 위해 제안된 방법으로, 다음과 같이 표현된다.

n = system 에 포함된 모듈들의 수

f(i) = 다른 모듈에 의해 i 번째 모듈이 불려지는 회수 (call 문장을 통해)

v(i) = i 번째 모듈에 포함된 I/O 변수들의 수

S(i): structural complexity of the i_{th} 모듈 → $S(i) = [f(i)]^2$

D(i): data complexity of the i_{th} 모듈 → $D(i) = \frac{v(i)}{f(i)+1}$

St = program structural complexity → $St = \sum_{i=1}^{n} [f(i)]^2$

Dt: program data complexity → $Dt = \frac{1}{n} \sum_{i=1}^{n} \frac{v(i)}{f(i)+1}$

C: program design complexity → $C = St + Dt$

Fr: expected failure rate (1000 line 당) → $Fr = 0.4 \cdot C - 5.2$

A.10 Entropy Measure [40, 41, 42]

일반적으로 엔트로피는 여러 분야에서 복잡도를 평가척도로서 사용되어 왔고, 소프트웨어 공학분야에서도 프로그램 제어 그래프(program control graph)나 프로그램 내에 포함된 자료 구조 그래프(data structure graph)의 복잡도 평가척도로 사용되어 왔다. 즉, 프로그램의 제어 그래프나 자료 구조 그래프가 너무 복잡하다면 어떤 프로그램을 이해하기 위하여 프로그래머나 관리/보수 요원이 투입해야 하는 시간 및 정신적인 노력은 증가하기 때문이다.

그래프에 대한 엔트로피는 다음과 같은 형태로 Mowshowitz에 의해 정의되었고, 이는 크게 그래프의 규칙성(regularity) 또는 대칭성(symmetry)을 평가할 수 있는 "일차(first-order) 엔트로피"와 그래프에 포함된 정보의 양을 평가할 수 있는 "이차(second-order) 엔트로피"로 구분된다.

$$H = -\sum_{i=1}^{h} p(A_i) \cdot \log[p(A_i)]$$

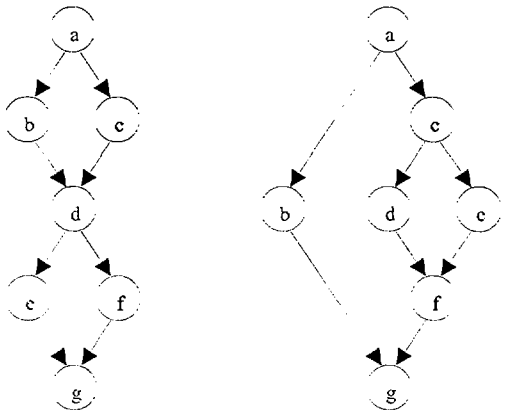
A_i = identified classes in a graph

h = the number of identified classes

$p(A_i)$ = estimated probability of A_i = $\frac{\text{the number of nodes included in } A_i}{\text{the total number of nodes in a graph}}$

일반적으로 일차 엔트로피는 프로그램 제어 그래프를 평가하는데 사용되고, 이차 엔트로피는 자료 구조 그래프를 평가하는데 주로 사용된다. 우선 일차 엔트로피를 계산하기 위해서는, 주어진 그래프의 노드들을 노드로 들어오는 화살표와 나가는 화살표의 수에 따른 종류(class)별로 구분하는 작업이 필요하다. 예를 들어, 그림 A.1과

같은 프로그램 제어 그래프의 일차 엔트로피를 계산하기 위해, 각각의 그래프에 대한 노드의 종류들을 표 A.6와 같이 구분할 수 있다.



G G'

그림 A.1 일차 엔트로피를 계산하기 위한 임의의 프로그램 제어 그래프

표 A.6 프로그램 제어 그래프 G 및 G'에 대한 일차 엔트로피 노드 구별 방법

그래프 G			노드 종류 (node class)	그래프 G'		
노드	in	out		노드	in	out
{a}	0	2	I	{a}	0	2
{b, c, e, f}	1	1	II	{b, d, e}	1	1
{d}	2	2	III	{c}	1	2
{g}	2	0	IV	{f}	2	1
			V	{g}	2	0

즉 노드 a의 경우 들어오는 화살표는 없는 반면 나가는 화살표는 2개이고, 그래프 G의 모든 노드들에 대해 같은 수의 화살표를 가지는 노드들이 없기 때문에, 노드 a는 하나의 특별한 종류(class)로 구분된다. 반면 노드 b, c, e 및 f의 경우 들어오는 화살표 및 나가는 화살표가 각각 1개로 동일하기 때문에, 이들은 모두 같은 노드 종류로 구별된다. 따라서, 표 A.6의 결과를 통해 그래프 G 및 G'에 대한 일차 엔트로피를 표 A.7과 같이 계산할 수 있다.

표 A.7 프로그램 제어 그래프 G 및 G'의 일차 엔트로피

그래프 G	h = the number of identified classes = 4 $p(A_I)$ = estimated probability of A_I = 1/7 $p(A_{II})$ = estimated probability of A_{II} = 4/7 $p(A_{III})$ = estimated probability of A_{III} = 1/7 $p(A_{IV})$ = estimated probability of A_{IV} = 1/7

	$H_G = -\sum_{i=1}^{4} p(A_i) \cdot \log[p(A_i)] = 1.664$
그래프 G'	h = the number of identified classes = 5 $p(A_I)$ = estimated probability of A_I = 1/7 $p(A_{II})$ = estimated probability of A_{II} = 3/7 $p(A_{III})$ = estimated probability of A_{III} = 1/7 $p(A_{IV})$ = estimated probability of A_{IV} = 1/7 $p(A_V)$ = estimated probability of A_V = 1/7 $H_{G'} = -\sum_{i=1}^{5} p(A_i) \cdot \log[p(A_i)] = 2.218$

이러한 일차 엔트로피의 의미는 그림 3.2를 비교해 보면 명확해 질 수 있다. 즉 그림 A.1에서, 그래프 G의 경우 b, c, e 및 f 노드가 같은 종류인 반면 그래프 G'의 경우에는 b, d 및 e 노드만이 같은 종류로 구분되어 있다. 이는 곧 그래프 G가 그래프 G'보다 규칙적기 때문에 덜 복잡한 그래프임을 의미하고, 이에 따른 표 7의 일차 엔트로피 결과는 그래프 G 쪽이 더 작게 계산되었다.

이러한 일차 엔트로피의 의미를 이해하기 위해, 그래프 G와 그래프 G'의 프로그램 제어 구조를 가지는 2개의 원시 프로그램을 생각해 보자. 만일 동일한 사람이 2개의 원시 프로그램을 관리할 때, 프로그램의 내용은 규칙적인 쪽이 그렇지 않은 쪽 보다 훨씬 더 쉽게 이해될 수 있을 것이다. 즉 규칙적인 프로그램 제어 구조는 대칭적인 특성을 갖고 있기 때문에 반복되는 부분만을 이해한다면 나머지 부분도 자연히 이해될 수 있기 때문이다. 그러나 규칙성이 적은 프로그램 제어 구조의 경우에는 반복되지 않는 모든 부분을 다 이해하기 위해 더 많은 노력이 필요하게 된다. 따라서 일차 엔트로피는 "원시 프로그램의 제어구조가 얼마나 쉽게 이해될 수 있는가?"에 대한 평가척도로 사용되고 있다.

이차 엔트로피도, 노드의 종류를 구분하는 방법을 제외하고는, 일차 엔트로피와 유사한 방법으로 계산될 수 있다. 즉 이차 엔트로피의 경우, 어떤 노드를 기준으로 그 노드에 직접 연결된 주위 노드(neighborhood node)들의 수와 종류가 동일할 때만 같은 노드 종류로 구분한다. 예를 들어, 이차 엔트로피에 사용하는 노드 구별 방법을 그림 A.2의 그래프 G 및 그래프 G'에 대해 적용해 본다면, 표 A.8과 같은 결과가 얻어진다.

표 A.8 프로그램 제어 그래프 G 및 G'에 대한 이차 엔트로피 노드 구별 방법

Graph G			Graph G'	
노드	주위 노드		노드	주위 노드
{a}	{b, c}	I	{a}	{b, c}
{b, c}	{a, d}	II	{b}	{a, g}

{d}	{b, c, e, f}	III	{c}	{a, d, e}
{e, f}	{d, g}	IV	{d, e}	{e, f}
{g}	{e, f}	V	{f}	{d, e, g}
		VI	{g}	{b, f}

즉 노드 a의 경우 주위 노드는 b, c이고, 그래프 G의 모든 노드들에 대해 같은 주위 노드를 가지는 노드는 없기 때문에 노드 a는 독립적인 노드 종류로 구별된다. 반면 노드 b는 동일한 주위 노드 a, d를 가지는 노드 c가 그래프 G에 있기 때문에 같은 노드 종류로 구별된다. 따라서, 위의 표 A.8의 결과를 사용한 그래프 G 및 G'의 이차 엔트로피는 표 A.7에서 제시된 방법과 유사하게 계산될 수 있다.

이러한 이차 엔트로피는 그래프를 노드들을 구별하기 위해 필요한 정보의 양이나 그래프의 크기(size)를 의미하게 된다, 즉 어떤 그래프에 포함된 노드들이 모두 각각의 이웃 노드들을 가지거나 그래프에 포함된 노드들의 수가 매우 많은 경우 노드들을 구별하기 위한 정보는 증가할 것이지만, 만일 많은 노드들이 있더라도 공통된 주위 노드들을 가진 노드들이 많다면, 노드들을 구별하기 위한 정보는 줄어들 것이다. 예를 들어, 그래프 G에 포함된 노드 e 및 f의 경우, 노드 e를 구별하기 위한 정보는 주위 노드 d, g이고, 이들은 동일하게 노드 f를 구분하는 데 사용될 수 있다. 따라서 노드 d를 구별할 수 있는 정보를 가지고 있다면, 노드 g에 대한 구별 정보는 추가적인 노력 없이도 쉽게 기억될 수 있다. 이러한 특성 때문에 이차 엔트로피는 소프트웨어 공학 분야에서 "그래프에 포함된 정보의 양" 또는 "그래프의 크기"에 대한 평가척도로 사용되고 있고, 특히 프로그램 내에 포함된 변수들의 자료 구조 그래프(data structure graph)를 평가하기 위해 사용된다. 즉 프로그램 내에 포함된 변수들의 자료 구조 그래프에서, 이차 엔트로피 값이 크다는 것은 그만큼 자료 구조를 이해하기 위해 사람이 구별해야 하는 정보의 양이 많다는 것을 의미하기 때문이다.

B. Requirement Quality 평가 방법

B.1 Cause and Effect Graphing

이 방법은 설계요건에 포함된 모호함이나 부적절성을 평가하기 위해 제안된 방법으로, 일반적으로 표현된 설계요건 문장들을 입력 및 출력조건 (input and output condition)에 대해 그래프를 구성하여 (cause and effect graph) 평가하게 된다. 평가 절차는 표 A.9 와 같다.

표 A.9 Cause and Effect Graphing 평가 절차

단계	내 용
1	각각의 설계요건에 대한 입력 및 출력 조건들에 대한 목록을 만든다.
2	만들어진 목록에 포함된 조건들을 그래프의 node 로 표현한 후 이들을 연결한다. 이때 각 node 에 대한 제한조건 (constraint)들도 함께 표현한다.
3	만들어진 그래프의 분석 (각 node 들이 가질 수 있는 값은 true/false 이고 주어진 제한 조건들을 고려한다)을 통해 애매한 node 들을 찾는다. 여기에서 애매한 node 들은 effect 를 가지지 않는 cause node 나 cause 를 가지지 않는 effect node 등을 의미한다.

B.2 Requirement Traceability

이 방법은 단순히 설계요건이 적절하게 반영되었는가를 평가하기 위해 제안된 것으로, 설정된 설계요건의 수를 R_1이라 하고 현재 반영된 설계요건 수를 R_2라고 할 때 이 둘의 비를 계산하여 평가척도로 사용한다.

B.3 Number of Conflicting Requirements

이 방법은 소프트웨어의 설계요건에 대해 이를 실제적으로 반영하는 function 들과 이 function 에 필요한 입력 및 출력 조건들을 사용하여 서로 다른 설계요건에 대해 같은 function/입력/출력 등이 사용되었는지를 평가하는 방법이다.

B.4 Requirements Compliance

이 방법은 system verification diagram (SVD)를 사용하여 설계요건에 포함된 불일치, 불완전함 및 잘못된 구현 등을 감지하기 위해 제안된 방법이다.

B.5 Completeness

이 방법은 설계요건의 완전성 (completeness)를 평가하기 위해 CM (completeness measure)를 제안하고 있다. 여기에서 CM을 계산하기 위한 절차는 다음과 같다. 우선 표 A.10과 같은 값들을 정한다.

A.10 Completeness 평가를 위한 기본적인 값들

B_1	number of functions not satisfactorily defined
B_2	number of functions
B_3	number of data references not having an origin
B_4	number of data references
B_5	number of defined functions not used
B_6	number of defined functions
B_7	number of referenced functions not defined
B_8	number of referenced functions
B_9	number of decision points not using all conditions or options or both
B_{10}	number of decision points
B_{11}	number of condition options without processing
B_{12}	number of condition options
B_{13}	number of calling routines with parameters not agreeing with defined parameters
B_{14}	number of calling routines
B_{15}	number of condition options not set
B_{16}	number of set condition options having no processing
B_{17}	number of set condition options
B_{18}	number of data references having no destination

이렇게 결정된 값들을 사용하여 표 A.11과 같은 10개의 세부 평가 척도들 및 CM을 계산한다.

표 A.11 세부 평가 척도

	표현식	의미
D_1	$(B_2 - B_1) / B_2$	Functions satisfactorily defined
D_2	$(B_4 - B_3) / B_4$	data references having an origin

D_3	$(B_6 - B_5) / B_6$	Defined functions used
D_4	$(B_8 - B_7) / B_8$	referenced functions defined
D_5	$(B_{10} - B_9) / B_{10}$	all condition options at decision points
D_6	$(B_{12} - B_{11}) / B_{12}$	all condition options with processing at decision points are used
D_7	$(B_{14} - B_{13}) / B_{14}$	calling routine parameters agree with the called routine's defined parameters
D_8	$(B_{12} - B_{15}) / B_{12}$	all condition options that are set
D_9	$(B_{17} - B_{16}) / B_{17}$	processing follows set condition options
D_{10}	$(B_4 - B_{18}) / B_4$	data references have a destination

$$CM = \sum_{i=1}^{10} w_i D_i , \qquad \sum_{i=1}^{10} w_i = 1.0$$

w_i는 가중치(weighting factor)로서 평가자가 각 세부 평가 척도들의 중요성에 따라 값을 할당하게 된다.

B.6 Requirements Specification Change Requests

이 방법은 소프트웨어 개발을 위해 처음 정해진 설계요건들에 대해 소프트웨어 개발 도중 여러 가지 이유로 인해 바뀐 설계요건들의 수를 세어, 제안된 설계요건들의 적절성(즉 처음 정해진 설계요건들이 적절하다면 바뀐 것들이 없을 것이기 때문)을 평가하는 방법이다.

C. 소프트웨어 설계 품질 평가 방법

C.1 Software Process Capability Determination

ISO/IEC Draft International Standard (DIS) 15504에서 제안하고 있는 이 방법은, 소프트웨어 개발 팀의 개발 과정 (process) 및 개발 능력 (capability)를 전문가의 판단에 의해 평가하도록 하고 있다. 이때 개발 과정 및 개발 능력을 평가하기 위해 고려되는 항목들은 표 A.12 와 같다.

표 A.12 소프트웨어 개발팀의 개발과정 및 개발능력 평가 항목

개발 과정 평가 항목	개발 능력 평가 항목
• Acquire software • Manage customer needs • Supply software • Operate software • Provide customer service • Develop system requirements and design • Develop software requirements • Develop software design • Implement software design • Integrate and test software • Integrate and test system • Maintain system and software • Develop documentation • Perform configuration management • Perform quality assurance • Perform work product verification • Perform work product validation • Perform joint reviews • Perform audits • Perform problem resolution • Manage the project • Manage quality • Manage risks • Manage subcontractors • Engineer the business • Define the process • Improve the process • Provide skilled human resources • Provide software engineering infrastructure	• Process performance attribute • Performance management attribute • Work product management attribute • Process definition attribute • Process resource attribute • Process measurement attribute • Process control attribute • Process change attribute • Continuous improvement attribute

이러한 항목들은 전문가들에 의해 N (not achieved), P (partially achieved), L (largely achieved) 및 F (fully achieved)로 평가된다.

C.2 Cost and Schedule

이 방법은 소프트웨어 개발팀에 대해 이 팀이 주어진 기간 및 예산으로 필요한 소프트웨어를 개발할 수 있는지를 평가하기 위해 제안되었다. 이를 위해 CI (cost index)와 SI (schedule index)와 같은 두개의 평가 척도를 제안하고 있는데, 이들의 계산에는 표 A.13 과 같은 값들이 사용된다.

표 A.13 Cost 및 Schedule 계산을 위한 기본 값

E_i	Estimated costs for life cycle phase I
C_i	actual cost of life cycle phase I
T_i	Estimated elapsed time to be spent on life cycle phase i
S_i	actual elapsed time for life cycle phase I
w_i	Weighting factor for life cycle i costs, $\sum_{i=1}^{10} w_i = 1.0$
v_i	Weighting factor for life cycle i elapsed time
$CI = \sum_i w_i \frac{C_i}{E_i}, \; SI = \sum_i v_i \frac{S_i}{T_i}$	

C.3 Reliability Prediction as a function of Development Environment [43, 44]

이 방법은 소프트웨어의 개발 환경 및 사용 환경등을 고려하여 소프트웨어의 신뢰도를 예측하기 위해 RADC (Rome air development center)에서 제안하였다. 즉 소프트웨어의 신뢰도 예측값 (R_p)은 $R_p = A \cdot D$ 형태로 표현된다. 이때 A 는 소프트웨어의 사용환경에 의해 결정되는 값이고 D 는 소프트웨어의 개발환경에 따라 결정되는 값이다. A 및 D 값은 RADC 의 경험에 따라 표 A.14 및 A.15 와 같은 형태로 주어진다.

표 A.14 A 값 할당표

사용 분야	A
Airborne	0.0128
Strategic	0.0092
Tactical	0.0078
Process Control	0.0018
Production Center	0.0085
Developmental	0.0123
Total/Average	0.0094

표 A.15 D 값 할당표

개발 환경		D
Organic	Software that is being developed by a group that is responsible for the overall application (for example, reactor protection system software that is being developed by a nuclear reactor vendor)	0.76
Semi-detached	The software developer has specialized knowledge of the application area, but is not part of the sponsoring organization (e.g., network control software that is being developed by a communications company that does not operate the target network)	1.0
Embedded	Software that frequently has very tight performance constraints being developed by a specialist software organization that is directly connected with the application (e.g., surveillance radar software being developed by a group within the radar manufacturer, but not organizationally tied to the user of the surveillance information).	1.3

이 방법 외에, "Reliability prediction for the Operational Environment" 및 "Reliability Prediction as a Function of Software Characteristics" 방법도 위에서 설명한 방법과 유사한 접근을 통해 소프트웨어의 신뢰도를 예측하고 있다.

D. 소프트웨어 시험에 대한 Coverage 평가 방법

D.1 Functional Test Coverage

이 방법은 소프트웨어에 포함된 기능 또는 모듈들에 대해 얼마나 많은 부분이 실제로 시험 되었는지를 평가하는 방법으로 다음과 같은 값들을 사용한다.

FE = 시험을 성공적으로 마친 소프트웨어에 포함된 기능 또는 모듈들의 수
FT = 소프트웨어에 포함된 기능 또는 모듈들의 총 수

$$\text{Functional Test Coverage Index} = \frac{FE}{FT}$$

D.2 Minimal Unit Case Determination

이 방법은 주어진 소프트웨어에 포함된 모든 모듈들에 대해, 최소한으로 필요한 수의 시험이 수행되었는지를 평가하기 위해 제안된 방법이다. 즉 소프트웨어에 포함된 모든 모듈들과 그들의 연관관계를 그래프로 표현한 후 (이 그래프는 cyclomatic complexity 를 계산하기 위해 구성되는 그래프와 동일한 성질을 가지고 있다), 소프트웨어가 주어진 기능을 수행할 수 있는 최소 독립경로 (minimal independent path)들을 찾아 이들이 모두 시험 되었는지를 평가하는 방법이다. 이때 최소 독립경로의 수는 cyclomatic complexity 값으로 결정되고, 이렇게 결정된 최소 독립경로들의 수에 대해 현재까지 시험 된 독립경로들의 수의 비율을 계산하여 소프트웨어 시험의 적절성을 결정하게 된다.

D.3 Test Coverage

이 방법은 소프트웨어 개발자 및 사용자의 관점에서 본 소프트웨어 시험의 완전성 (completeness)을 평가하기 위해 제안된 방법으로 다음과 같은 값들을 필요로 한다.

Requirement Capabilities (RC) – 소프트웨어에 대한 설계요건 들의 수
Implemented Capabilities (IC) – 제안된 설계요건들에 대해 실제로 구현된 것들의 수
Total Program Primitives (TPP) – 소프트웨어에 포함된 모듈들의 수와 이 모듈들에 사용되는 자료들의 종류
Program Primitives Tested (PPT) – 실제로 시험이 수행된 primitive 들의 수

$$TC\ (\%) = \frac{IC}{RC} \times \frac{PPT}{TPP} \times 100$$

D.4 Test Sufficiency

이 방법은 주어진 소프트웨어에 대해 예측된 결함들의 수와 소프트웨어의 통합 시험 (integrated test) 도중 발견된 결함들의 수의 비율을 계산하여 소프트웨어의 통합 시험이 적절하게 수행되었는지를 결정하기 위한 방법으로서, 다음과 같은 값들을 통해 계산된다.

NF = 소프트웨어에 포함되었다고 예측되는 결함들의 수

F_{it} = 소프트웨어 시험 도중 발견된 결함들의 수

F_{pit} = 소프트웨어 시험 전에 발견된 결함들의 수

M_{it} = 소프트웨어에 통합된 모듈들의 수

M_{tot} = 소프트웨어에 최종적으로 통합되는 모듈들의 수

NF_{rem} = 시험 후 소프트웨어에 남아있을 것으로 예상되는 결함들의 수

$$NF_{rem} = \frac{M_{it}}{M_{tot}} \cdot (NF - F_{pit})$$

D.5 Testability Analysis [45]

PIE (Propagation, Infection and Execution) 분석이라고도 불리는 이 방법은 주어진 소프트웨어에 대해 결함들이 포함되어 있을 가능성이 높은 부분을 찾은 후 이들을 모두 시험하기 위해 필요한 시험 회수를 구하기 위해 제안되었고, 이름에서 알 수 있는 바와 같이 주어진 소프트웨어에 대해 propagation, infection 및 execution 분석이 필요하다. 이들에 대한 간략한 설명은 표 A.16 과 같다.

표 A.16 PIE 분석 방법

Execution 분석	n 번 소프트웨어를 실행한 후, 소프트웨어의 l 번째 line 이 얼마나 수행되는지를 세어 c_l 값으로 할당한다.
Infection 분석	n 번 소프트웨어를 실행한 후, 소프트웨어의 l 번째 line 이 소프트웨어의 내부 자료 상태 (internal data state)를 몇 번이나 변화시키는지 세어 c_{ml} 값으로 할당한다.
Propagation	n 번 소프트웨어를 실행한 후, 소프트웨어의 l 번째 line 이

분석	소프트웨어의 외부 자료 상태 (즉 소프트웨어의 출력 또는 결과 값)를 몇 번이나 변화시키는지 세어 c_{vl}값으로 할당한다.

이렇게 얻어진 값들을 사용하여 필요한 시험 회수 (T)는 다음과 같이 계산된다.

$$\varepsilon_l = \frac{c_l}{n},\ \lambda_{ml} = \frac{c_{ml}}{n},\ \Psi_{vl} = \frac{c_{vl}}{n}$$

$$\beta_l = \varepsilon_l \cdot \left[\sigma\left\{ \min_m(\lambda_{ml}), \min_v(\Psi_{vl}) \right\} \right],\ \sigma(a,b) = \max(0, a-(1-b))$$

$$T = \frac{\ln(1-c)}{\ln(1-\min_l(\beta_l))}$$

E. 소프트웨어의 Test Effort 평가 방법

E.1 Fault Number Days

일반적으로 소프트웨어 시험에 투입된 노력 (effort)이 클수록 소프트웨어 신뢰도가 좋아질 것이라는 생각할 수 있다. 따라서 이 방법은 어떤 결함의 발견에서부터 수정까지 걸린 시간들을 모두 합한 값을 소프트웨어 시험을 위해 투입된 노력으로 제안하고 있다. 즉 FD_i를 "i 번째 결함의 발견에서 수정까지 걸린 시간"이라고 할 경우 소프트웨어에 투입된 총 노력 FD (fault days number)는 다음과 같다.

$$FD = \sum_i FD_i$$

E.2 Man-hours per Major Defect Detected

이 방법은 앞에서 설명한 방법에 비해 소프트웨어에 투입된 노력을 세분한 점 외에는 동일한 개념의 접근을 시도하고 있고, 다음과 같은 값들을 통해 소프트웨어에 투입한 인력 (man-hours)을 계산하게 된다.

T_1 = 소프트웨어를 검사 (inspection)하기 위한 전담 팀 (team)의 구성에 소요된 시간
T_2 = 소프트웨어 검사 팀에서 소프트웨어를 분석/이해하기 위해 소요된 시간
I = 총 소프트웨어 검사 회수
S_i = i 번째 검사에서 발견된 중요한 결함 (non-trival fault) 들의 수

$$\text{Man-hours per major Faults} = M = \frac{\sum_{i=1}^{I} (T_1 + T_2)}{\sum_{i=1}^{I} S_i}$$

F. 소프트웨어의 Test Team Availability/Capability 평가 방법

F.1 Number of Faults Remaining

이 방법은 어떤 소프트웨어를 시험하기 위해 구성된 팀의 능력을 평가한 후, 이들의 능력에 따라 시험된 소프트웨어에 남아있는 결함들의 수를 예측하는 접근 (sampling technique)을 취하고 있다. 예를 들어, 시험 하고자 하는 소프트웨어에 N_s개의 알려진 결함들을 집어넣은 후 (fault seeding), 시험 팀에게 찾도록 한다. 이때 시험 팀이 심어진 결함들 중에서 n_s개의 결함들을 찾았고 동시에 소프트웨어에 있었던 (심어지지 않은) 결함들 중 n_F개를 찾았다고 가정할 경우, 소프트웨어에 남아있다고 예상되는 결함들의 수 NF 는 다음과 같이 계산된다.

$$\text{Total number of "real" faults (estimated)} = \text{NF} = \frac{n_F}{n_s} \cdot N_s$$

따라서 소프트웨어에 남아있는 결함들의 수 NF_{rem}은 다음과 같이 주어진다.

$$NF_{rem} = NF - n_F$$

이 외에, mutation testing 및 test accuracy 방법 역시 number of faults remaining 과 동일한 접근을 통해 소프트웨어에 포함되어 있을 것이라고 예상되는 결함들의 총 수를 예측하고 있기 때문에, 본 보고서에서 자세한 언급은 생략하였다.

서 지 정 보 양 식

수행기관보고서번호	위탁기관보고서번호	표준보고서번호	INIS 주제코드
KAERI/AR-565/00			

제목 / 부제	소프트웨어 신뢰도의 정량적 평가 기법에 대한 고유 현안 분석

연구책임자 및 부서명 (주저자)	박진균 (종합안전평가팀)
연 구 자 및 부 서 명	성태용 (종합안전평가팀), 엄홍섭 (종합안전평가팀), 정환성 (하나로운영팀), 박진희 (종합안전평가팀), 강현국 (종합안전평가팀), 이기영 (동력로기술개발팀), 박종균 (동력로기술개발팀)

출 판 지	대전	발행기관	KAERI	발행년	2000. 4.
페 이 지	67 p.	도 표	있음(○), 없음( )	크 기	21×29.7cm
참고사항					
비밀여부	공개(○), 대외비( ), __ 급비밀		보고서종류	기술현황분석보고서	
연구위탁기관			계약 번호		

초록

디지털 계측제어 계통에 포함되어 있는 소프트웨어의 신뢰도를 평가하기 위한 방법론을 개발하기 위한 기초단계로서, 현재까지 제안된 소프트웨어 신뢰도 평가방법들을 조사하고 이들의 장·단점을 분석하였다.

분석 결과, 현재까지 제시된 여러 가지 방법론들 중에서 디지털 계측제어 계통에 포함되는 소프트웨어의 신뢰도를 적절히 평가할 수 있는 것은 아직 없는 것으로 판단되므로, 현재 제안된 여러 가지 방법론들 중 가장 적절한 것들을 조합하여 소프트웨어의 신뢰도를 평가할 수 있도록 하는 연구가 진행되어야 할 것으로 생각된다.

주제명키워드 (10단어내외)	디지털 계측제어 계통, 확률론적 안전성 평가, 소프트웨어, 신뢰도, 정량적 평가

BIBLIOGRAPHIC INFORMATION SHEET

Performing Org. Report No.	Sponsoring Org. Report No.	Standard Report No.	INIS Subject Code
KAERI/AR-565/00			

Title / Subtitle	A technical survey on issues of the quantitative evaluation of software reliability
Project Manager and Department	J.K. Park (Integrated Safety Assessment team)
Researcher and Department	T.Y. Sung (ISA team), H.S. Eom (ISA team), H.S. Jeong (Hanaro) J.H. Park (ISA team) and H.G. Kang (ISA team), K.Y. Lee (ARTD team), J.K Park (ARTD team)

Publication Place	Taejon	Publisher	KAERI	Publication Date	2000. 4.
Page	67 p.	Ill. & Tab.	Yes(○), No ( )	Size	21× 29.7cm
Note					

Classified	Open(○), Restricted( ), ___ Class Document	Report Type	Analysis Report
Sponsoring Org.		Contract No.	

Abstract (15-20 Lines)

To develope the methodology for evaluating the software reliability included in digital instrumentation and control system (I&C), many kinds of methodologies/techniques that have been proposed from the software reliability engineering field are analyzed to identify the strong and week points of them.

According to analysis results, methodologies/techniques that can be directly applied for the evaluation of the software reliability are not exist. Thus additional researches to combine the most appropriate methodologies/techniques from existing ones would be needed to evaluate the software reliability.

Subject Keywords (About 10 words)	digital I&C system, probabilistic safety assessment, software, reliability, quantitative evaluation