확률적 신뢰도 성장

최규식°

건양대학교 의공학과

° Che Gyu Shik

Stochastic Software Reliability Growth

요약

프로그램의 고장율이 잔여 결함의 (미지)수에 대한 일정한 배수라고 한 것은 모든 결함이 프로그램의 고장율에 동일한 양으로 기여한다는 것을 의미한다. 우리는 이 가정에 대해서 도전을 하고자 하며, 대안을 제시한다. 이 모델은 다루기 쉬워서 계산할 다양한 신뢰도 척도를 허용한다. 목표 신뢰도를 얻기 위한 전체 수행시간과 목표 신뢰도를 얻기 위한 총 결함의 수를 예측할 수 있다. 이 모델은 설계오류를 줄여서 신뢰도 성장을 가져오는 하드웨어에도 적용될 수 있을 것으로 기대된다.

1. 서론

소프트웨어 신뢰도에서 모든 결함이 제거된다면, 프로그램이 완벽하여 영원히 고장을 일으키지 않고 작동할 것이다. 이것을 고전적인 하드웨어 신뢰도 이론과 대비해보자. 하드웨어에서는 전체 시스템의 고장이 부품 고장에 의해서 발생한다. 이것을 볼 때 어느 시스템이 되었건 구성 부품 전체가 완벽할 수는 없기 때문에 시스템이 언제나 고장을 일으킨다는 것을 확신할 수 있다. 명백히, 고장공정이 잔여결함의 수에 의존한다. 그리고, 신뢰도 성장이 결함제거와 관련이 있다. 초기 연구[3]에서는 동일크기의 일련의 단계에서 디버깅이 고장율 향상을 이루게 하기 위해서 각각의 결함이 동일한 양으로 전체 고장율에 기여를 하는 것으로 가정했다.[3]

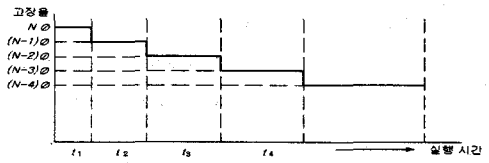

그림 1. 고장율 그림

이 상태에서 (i-1)개의 결함이 제거되었으므로, 확률은

$$pdf(t_i)=\lambda_i \exp(-\lambda_i t_i) \quad (1)$$

이고,

고장율은

$$\lambda_i=(N-i+1)\phi \quad (2)$$

라고 한다.

본 논문에서는 기본적인 모델 문제를 근본적으로 고찰한다. (1)과 (2)에 의해서 발생되는 기본 모델조차도 때때로 어려움에 봉착하게 된다. Sukert[8]는 이러한 연구를 수행한 첫 번째 사람으로 보인다. 이들은 F&S[9], L&V[10]에 의해서 연구되어왔다. 데이터 집합이 명백하게 신뢰도 성장을 보여주지 않는 한 이 모델이 잘못 유도된 해법을 제시한다. 프로그램 p_i를 프로그램 p_{i+1}로 바꾸는 결함제거 작동이 λ_i와 λ_{i+1} 사이의 관계에 대해서 불확실성을 가지고 있다. 이는 이 단계에서 제거되는 결함의 빈도(λ_i를 λ_{i+1}로 향상시키면서)가 알려지지 않아서 예측할 수 없기 때문이다. 사실상, 비록 프로그램의 운영 프로파일에서 결함발생율의 대략적인 추정이 가능하다고는 하나, 결함이 제거된 후에라도 이를 측정하는 것이 보통은 가능하지 않다. 디버깅이 진행됨에 따라 최대발생빈도를 가지는 결함에 대해서 조기에 고장이 제거되는 경향이 있으며, 따라서, 맨 최초에 제거된다. 그러므로, 디버깅에서 프로그램의 고장율이 조기에 신뢰도 향상을 크게 나타내는 경향이 있다는 것을 의미한다. 프로그램은 결함 반을 제거한 후 고장율이 반 이하일 수도 있다. 디버깅 후반부에는 프로그램이 매우 높은 신뢰도를 성취하나, 아직도 많은 결함을 포함하고 있다. 이러한 확률적 디버깅의 결과는 (2)에서 의미하는 것보다 경험에 의한 것에 좀더 가까운 것처럼 보인다. 소프트웨어 신뢰도 모델링에 대한 초기의 연구[4]는 초기의 포괄적인 하드웨어 이론으로부터 유도된 학습(lesson)을 강조하는 경향이 있었다. 좀더 근래에는 단순한 평행상태(parallel)가 잘못 유도된다는 것을 감지하는 면이 점점 더 증가하고 있다고 할 수 있다.[1] 현재 대부분의 연구자들은 하드웨어와 소프트웨어 사이에 부합되는 신뢰도 척도가 사용되어야 한다는 것에 동의하고 있으며, 그러므로, 복잡한 하드웨

어/소프트웨어 시스템의 신뢰도 연구가 타당성이 있게 해야 진행되어야 한다는 것이다. 하드웨어 신뢰도 이론은 시스템 거동에서 부품고장의 원인에 초점을 맞추는 경향이 강했다. 그러한 부품 고장은 근본적으로 반복적이다. 부품이 고장나면 유사한, 작동 가능한 부품으로 교체하게 되며, 이것 또한 결국은 고장 나게 된다. 이것은 한 번 프로그램을 수정하면 계속하여 수정된 상태를 유지하여 다시는 동일한 고장을 일으키지 않는 소프트웨어의 경우와 대비가 된다.

2. 모델

우리의 관심사는 시간-고장 분포에 있으며, 데이터는 고장 사이의 연속실행시간 t_1, t_2, t_3, …, t_i, …의 시퀀스가 된다. 단순하게 하기 위해서 결함제거가 순간적이고, 결함이 발견될 때마다 확실하게 제거되는 것으로 가정한다. 후자의 가정은 불완전 디버깅인 경우를 모델링하기 위해서 생략할 수도 있으나, 여기에서는 이러한 복잡성을 도입하지 않는다.

불행히도, 고장율이 미지수이므로, 미지의 프로그램 초기 결함 총수 N과 이미 수정된 기지의 결함수를 통하여 모델링해야 한다. 그 다음, 확률변수 T를 고려해보기로 한다. 총 경과시간이 τ이고 i개의 결함을 수정한 경우를 생각해보자.(그림 2) 잔여결함수는 $N-i$ 개다. 결정적인 점은 이 $N-i$ 개의 결함들이 각각 다른 발생율 ϕ_1, ϕ_2, …, ϕ_{N-i}를 가질 것이란 사실이다. 프로그램의 고장율은

$$\lambda = \phi_1 + \phi_2 + \cdots + \phi_{N-i} \tag{4}$$

이다.

그림 2. 고장율과의 관계

각 결함에 대한 고장의 시간분포가 지수분포라 가정하면, ϕ가 기지수일 때 프로그램의 현재 신뢰도를 완벽하게 기술할 수 있다. 그러나, ϕ는 미지수이다. 사실상, 잔여결함에 대해서 ϕ를 추정하기 위한 어떠한 가용 데이터도 없다는 것이다, 그러므로, 우리는 각각의 고장발생율을 어떤 분포를 가진 확률변수 Φ로 취급하여 ϕ값에 대한 불확실성을 모델링해야만 했다. 프로그램의 고장율은 가정 3에 의해서 전에서와 같이

$$pdf(t|\Lambda=\lambda) = \lambda \exp(-\lambda t) \tag{5}$$

로 한다.

이제 우리는 결함 발생율(및 그에 따른 프로그램의 고장율)에 대한 불확실성이 시간에 따라 어떻게 변화하는가를 설명하는데 있어서 베이시안 이론을 사용해야 할 위치에 와 있다. 그림 2의 "현재" 시간에서 $N-i$개의 잔여 결함중의 하나에 대한 발생율 Φ을 생각해보기로 한다.

$$pdf(\phi|\textit{this fault not fixed in } (0,\tau)) \tag{6}$$

$$= c \cdot e^{-\phi\tau} \cdot \beta^{\alpha}\phi^{\alpha-1}e^{-\beta\phi}/\Gamma(\alpha) \tag{7}$$

$$= (\beta+\tau) \cdot gamd([\beta+\tau]\phi;\alpha) \tag{8}$$

(3)으로부터 프로그램의 고장율 Λ는 $(N-i)$ i.i.d. gamf(α, $\beta+\tau$) 확률변수의 합이므로, 아래와 같은 pdf를 가진다.

$$(\beta+\tau) \cdot gamd([\beta+\tau]\lambda;[N-i]\alpha) \tag{9}$$

마지막으로, T를 얻을 수 있다.(그림 2)

$$pdf(t) = \int_0^\infty pdf(t|\Lambda=\lambda)d\lambda = \int_0^\infty \lambda e^{-\lambda t} \cdot pdf(\lambda)d\lambda \tag{10}$$

$$= \frac{(N-i)\alpha(\beta+\tau)^{(N-i)\alpha}}{(\beta+\tau+t)^{(N-i)\alpha+1}} \tag{11}$$

이는 $(\beta+\tau)^{-1} \cdot pard(t/[\beta+\tau];[N-i]\alpha)$이다. 이것이 연구논문의 나머지에 있는 모든 일상적인 스칼라 척도이다. 그러므로, 신뢰도 및 고장율 함수는

$$S(t) = \left[\frac{\beta+\tau}{\beta+\tau+t}\right]^{(N-i)\alpha} \tag{12}$$

$$\lambda(t) = (N-i)\alpha/(\beta+\tau+t) \tag{13}$$

이고, t에서 감소한다. 사실상, 이러한 감소고정율(DFR) 특성은 Φ에서 선정된 분포와 무관하다. 이는 혼합작동(10)의 일반적인 결과이다.[13 및 4장 참조] 결함발생율에 대한 불확실성을 나타내기 위해서 우리가 채택한 감마분포는 수학적인 추적성(traceability)에 의해서 지시된다. 그러나, 이러한 선택이 상당한 융통성을 제공하는 것은 아니므로, 나는 가장 필요로 할 때 적절할 것으로 기대한다.

그림 2의 화살표 시점에 있는 고장평균시간은 T의 s-예측치이다. (11)로부터

$$E(T) = (\beta+\tau)/[(N-i)\alpha-1] \tag{14}$$

이며, 이는 $(N-i)\alpha>1$, $\alpha>1$인 한 존재한다.

그림 4의 화살표 시점에 있는 고장율을 고려해보자. (13)으로부터

$$\lambda(0) = (N-i)/(\beta+\tau) \tag{15}$$

이며, 이는 총 실행시간 τ에서 i번 고장 나서 i번 수정한 프로그램의 고장율이다. 이것이 조건확률로 정의되므로 조건 고장율이라 부른다. 디버깅이 진행되면서 이것이 어떻게 변화하는지 주의해보기로 하자. 고장 없이 작동되는 기간동안에는 i가 일정한 상태로 남아 있고 τ는 증

가한다. 이것이 고장율을 쌍곡선으로 감소되도록 한다. 고장이 발생하여 결함이 고쳐지면, 고장율이 $a/(\beta+\tau)$만큼 줄어든다. 작은 τ에서 일찍이 결함을 수정하면, 후자에 비하여 프로그램의 조건고장율을 크게 감소시킬 수 있다. 따라서, 이 모델은 그림 3에서 보인 바와 같이 시간 대비 조건고장율 그림을 그리게 된다.

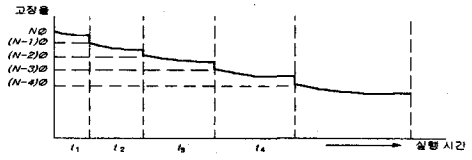

그림 3. 버그 수정이 빠르면 점프를 크게 나타냄 : 버그 수정 사이의 DFR

하드웨어 신뢰도성장에 관한 문헌에 익숙한 독자들은 본 모델과 Duane 형태의 고장율 모델링 사이의 차이를 알 수 있을 것이다. 이는 연속감소함수를 가진 NHPP의 변량인 것처럼 보인다. Crow[16]는 웨이블 분포의 위험도와 동일한 수학적 형태를 갖춘 고장율 함수를 가진 특수한 경우(웨이블 공정으로 잘못 명명됨)에 대해서 고려하였다. 지금까지 연구된 것들에 비하여 현재의 모델 장점은 고장 사이의 DFR 특성 외에도 고장율이 결함 제거로 해서 이산적으로 향상될 때 직접적으로 모델링할 수 있다는 것이다.

3. 미래 신뢰도 예측

미래의 신뢰도 예측은 두 가지 방법으로 접근 가능하다. 한 쪽은 어떤 규정된 실행시간이 경과된 후에 프로그램이 얼마나 신뢰성 있게 작동하는가 하는 것을 알기 원할 때가 있다. 또 다른 한편으로 얼마만큼의 규정된 결함수를 수정한 후에 그 신뢰도가 얼마나 될까 하는 것에 좀더 관심을 가질 수 있다. 실행시간과 역일 시간이 거의 동일하다면 첫 번째 접근방법이 좀더 유용할 수 있다. 두 번째 접근법은 실행시간이 "수리"시간에 비하여 작을 때에 더 유용하다. 이 문제는 단지 이 모델(앞의 [3-7]과 공동으로)이 "수리시간"을 무시하기 때문에 발생된다. 불행히도, 전체 역일 시간모델에 합쳐질 수밖에 없는 수리시간분포에 대해서는 합의된 바가 없다고 볼 수 있을 정도이다. 이 분야는 앞으로 많은 연구를 필요로 한다.

3.1 앞쪽 신뢰도 k-고장

그림 4와 같은 상황을 고려해보자. 전체 경과시간이 τ이고 이 기간동안 i개의 결함을 발견하여 수정하였다. 우리는 지금 k개의 고장이 더 발생한 후 프로그램의 신뢰도가 어떻게 될 것인가에 관심이 있다. 그림 4의 A라고 표시된 시점을 참조하자. 우리는 $T(\equiv T_{i+k+1})$의 무조건적인 분포를 요한다.

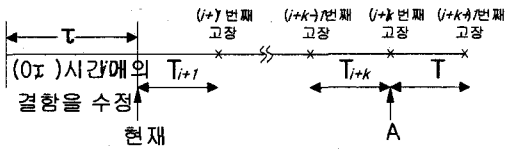

그림 4. 신뢰도 예측도

3.2 미래의 실행시간 τ'가 경과한 후의 신뢰도

이제 그림5의 B 시점에 있는 프로그램의 신뢰도를 고려해보자. 전과 같이 전체 실행시간 τ동안 i개의 고장(그리고 따라서 i개의 결함을 수정)을 관찰하였다. 미래의 디버깅 시간(실행시간) τ'가 경과한 후 프로그램이 얼마나 신뢰성 있게 작동하냐 하는 것에 관심이 있다. 본장의 결론은 하드웨어 설계오류 제거에 대한 3.1보다 더욱 더 관심이 갈 것이다. 그러한 상황에서는 역일시간이 제약요소가 될 것으로 본다.

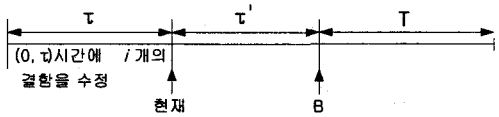

그림 5. 미래의 신뢰도 예측

3.3 마지막 결함을 제거할 때까지의 시간

"완벽한" 프로그램을 얻을 때까지 필요한 디버깅시간(실행시간)의 분포를 아는 것이 큰 관심사일 수가 있다. 이 확률변수를 T^*라고 하자. i개의 고장이 (0, τ)기간 동안 이미 제거되었으므로, 프로그램에는 $(N-i)$개만 남아있다. 이러한 잔여 결함 중에서 시간 X_j가 지난 후에 j번째 결함이 고장을 일으켰고 또 그 결함이 제거되었다고 하자. 그러면

$$T^* = \max_{j=1,\cdots,(N-i)} \{X_j\} \qquad (16)$$

이다.

3.4 주어진 시간 동안의 고장

이 결과는 그러한 상황에서 프로그램 사용환경(즉, 고객이 수령한 후)에서 디버깅이 계속되는 값일 것이다. 고

장에 의해서 정지되는 정지시간에 관한 것과 결합하여 관심사가 되는 기간 예를 들면 수행시간 또는 전 수명사이클에 걸친 시스템의 이용율에 관한 정보를 제공할 수 있다.

3.5 k개의 고장에 대한 실행시간

그림 5에서 k개의 고장이 발견될 때까지 지금부터 전체 시간의 cdf를 고찰해보기로 한다. 이는

$$\Pr\{T_{i+1}+T_{i+2}+\cdots+T_{i+k}<t\}$$

$$=\Pr\left\{Z>\left[\frac{\beta+\tau}{\beta+\tau+t}\right]^{\alpha}\right\} \quad (17)$$

이다.

4. 모델파라미터 추정

일반적인 데이터 집합은 고장 사이의 연속된 실행시간열 $t_1, t_2, \cdots, t_i$이다. 추정하는데에 3개의 파라미터 N, α, β가 있으며, 이는 J&M모델에서 두 개의 파라미터 N, ϕ인 것과 비교가 된다. (11)로부터

$$pdf(t_m|t_1, t_2, \cdots, t_{m-1})$$

$$=pdf\left\{t_m|\text{전체 실행시간 }\sum_{1}^{m-1}t_j\text{에서 }(m-1)\text{개의고장}\right\} \quad (18)$$

이므로, 평균 최소 평가치는

$$L(N, \alpha, \beta)=\prod_{m=1}^{i} pdf(t_m|t_1, t_2, \cdots, t_{m-1})$$

$$=\prod_{m=1}^{i}\frac{(N-m+1)\alpha(\beta+\sum_{1}^{m-1}t_j)^{(N-m+1)\alpha}}{(\beta+\sum_{1}^{m-1}t_j+t_m)^{(N-m+1)\alpha+1}} \quad (19)$$

이고, 평균최소방정식은 N, α, β에 대해서 해석적으로 구할 수가 없다. 그러나, 다음과 같은 식에 유의하면 단순화시킬 수는 있다.

$$\hat{\alpha}=\frac{i}{\sum_{m=1}^{i-1}\log\left[\frac{\hat{\beta}+\sum_{1}^{m}t_j}{\hat{\beta}+\sum_{1}^{i}t_j}\right]+\hat{N}\log\left[\frac{\hat{\beta}+\sum_{1}^{i}t_j}{\hat{\beta}}\right]} \quad (20)$$

이는 L의 최대치가 (N, β)의 이차원공간에서 생기도록 한다.

5. 결론

여기에 제시된 모델은 이미 존재하고 있던 기존의 결함계수 모델을 비판하는 데에서 시작되었다. 그 의도는 기존모델의 심각한 반대 현상으로 나타나는 것을 극복하기 위한 것이다. 즉, 프로그램내의 모든 결함에 대해서 프로그램의 고장을 입장에서 동일한 심각성을 가진 것으로 가정한 것을 말한다. 사실상 결함의 심각성에 있어서 여러 다양성이 있으며, 이것이 결함제거로부터 연유되는 신뢰도 성장모델에 반영되어야만 하는 것이다.

어떤 고장들은 다른 것(발생율을 차별화하는 문제와 확연히 구분)보다 훨씬 더 심각한 결과를 가져오기도 한다. 그래서 우리는 이러한 비용(또는 결과) 공정을 모델링해야 했다. 이러한 관찰은 동일한 힘으로 고전적인 연구에 적용한다. 그러나, 설계오류를 연구할 때는 특별한 힘을 가지고 있다. 치명적인 설계의 고장영향으로부터 우리를 보호하는 능력은 극히 제한되어 있다. 이는 기존의 신뢰도 연구와 대비가 되며, 특별히 치명적인 부품중에 여유다중을 가지는 것이 가능한 경우에서 대비된다. 우리는 소프트웨어 및 설계 신뢰도 분야에서 유사한 기법을 요한다. 일부 공정을 여기에서 진행하였다. 그러나, 수명주기 비용(고장의 결과로)을 모델링하는데 있어서 지금까지 데이터 부족이 전진을 방해해왔다.

참고문헌

[1] B. Littlewood, "How to measure software reliability and how not to" , IEEE Trans.
Reliability, vol R-28, 1979 Jun, pp103-110.

[2] B. Littlewood, "Theories of software reliability" , IEEE Trans. Software Engineering,
vol SE-6, 1980 Sep, pp465-484.

[3] Jelinski, P.B.Moranda, "Software Reliability Research" , in statistical computer
performance evaluation New York, Academic Press, 1972 pp465-484.