中国计算机学会学术著作丛书

软件可靠性模型及应用

徐仁佐　谢　　郑人杰

清华大学出版社
广西科学技术出版社

SOFTWARE RELIABILITY MODELS AND APPLICATIONS

XU REN-ZUO
XIE MIN
ZHENG REN-JIE

内 容 提 要

本书介绍了软件可靠性研究的现状与发展，主要讨论了随机过程类和非随机过程类模型，以及其它一些实用的模型，并介绍了各种不同的建模方法以及模型的分类与比较技术。此外，对软件可靠性技术应用于软件开发的成本和软件的最佳投放时间，也作了详细的介绍。书中既结合了作者的研究工作，对非齐次泊松过程模型进行了深入的探讨，也针对软件可靠性研究中目前存在的问题以及发展的几个主要方向，进行了深入的论述。

本书适于软件工程及软件可靠性分析专业的教师、高年级本科生、研究生阅读；对于广大软件工程界负责软件项目开发、分析的人员，软件质量高级管理人员，以及希望了解软件可靠性的现状与发展方向的研究人员和实际工作者，本书更是必备的。

(京)新登字 158 号
(桂)新登字 06 号

软件可靠性模型及应用
徐仁佐　谢　砾　郑人杰
☆
清华大学出版社出版
北京　清华园
广西科学技术出版社出版
南宁市河堤路 14 号
顺义振华印刷厂印刷
新华书店总店科技发行所发行
☆
开本：787×1092 1/16　印张：17.25 字数：401 千字
1994 年 1 月第 1 版　1994 年 1 月第 1 次印刷
印数：0001—6000
ISBN 7-302-01332-2/TP·516
单价：12.80 元

SUMMARY

Software Reliability Models and Applications introduces the art of the state and the development in the research field of software reliability. In this book, authors briefly discuss random and non-random process classes of software reliability models, some modelling methods for analyses and predictions of software reliability, and the techniques of classification and comparison for existing software reliability models. In addition, the authors introduce the applications to the control of software development cost and the prediction of the optimal release time of software products in detail, as well. In this book, the authors not only investigate the Non-homogeneous Poisson Process models deeply, specially combining with their research works, but also discuss the problems existing in recent investigations and some main developing directions in the field of software reliability. This book was written to give practical help to the graduate students and higher ranks of developers, analyzers, managers who want to do a better job of managing the developments for the software engineering, and software reliability engineering. Especially, this book is indispensable one to who hope to understand the present state and developing direction in the field of software reliability.

清华大学出版社　广西科学技术出版社
计算机学术著作出版基金

评 审 委 员 会

主 任 委 员　张效祥

副主任委员　周远清　汪成为

委　　员　王鼎兴　杨芙清　李三立　施伯乐　徐家福
夏培肃　董韫美　张兴强　徐培忠

出　版　说　明

近年来，随着微电子和计算机技术渗透到各个技术领域，人类正在步入一个技术迅猛发展的新时期。这个新时期的主要标志是计算机和信息处理的广泛应用。计算机在改造传统产业，实现管理自动化，促进新兴产业的发展等方面都起着重要作用，它在现代化建设中的战略地位愈来愈明显。计算机科学与其它学科的交叉又产生了许多新学科，推动着科学技术向更广阔的领域发展，正在对人类社会产生深远的影响。

科学技术是第一生产力。计算机科学技术是我国高科技领域的一个重要方面。为了推动我国计算机科学及产业的发展，促进学术交流，使科研成果尽快转化为生产力，清华大学出版社与广西科学技术出版社联合设立了“计算机学术著作基金”，旨在支持和鼓励科技人员，撰写高水平的学术著作，以反映和推广我国在这一领域的最新成果。

计算机学术著作出版基金资助出版的著作范围包括：有重要理论价值或重要应用价值的学术专著；计算机学科前沿探索的论著；推动计算机技术及产业发展的专著；与计算机有关的交叉学科的论著；有较大应用价值的工具书；世界名著的优秀翻译作品。凡经作者本人申请，计算机学术著作出版基金评审委员会评审通过的著作，将由该基金资助出版，出版社将努力做好出版工作。

基金还支持两社列选的国家高科技重点图书和国家教委重点图书规划中计算机学科领域的学术著作的出版。为了做好选题工作，出版社特邀请“中国计算机学会”、“中国中文信息学会”帮助做好组织有关学术著作丛书的列选工作。

热诚希望得到广大计算机界同仁的支持和帮助。

清华大学出版社
广西科学技术出版社
计算机学术著作出版基金办公室

1992 年 4 月

序　　言

计算机是当代发展最为迅猛的科学技术，其应用几乎已深入到人类社会活动和生活的一切领域，大大提高了社会生产力，引起了经济结构、社会结构和生活方式的深刻变化和变革，是最为活跃的生产力之一。计算机本身在国际范围内已成为年产值达 2500 亿美元的巨大产业，国际竞争异常剧烈，预计到本世纪末将发展为世界第一大产业。计算机科技具有极大的综合性质，与众多科学技术相交叉而反过来又渗入更多的科学技术，促进它们的发展。计算机科技内容十分丰富，学科分支生长尤为迅速，日新月异，层出不穷。因此在我国计算机科技尚比较落后的情况下，加强计算机科技的传播实为当务之急。

中国计算机学会一直把出版图书刊物作为学术活动的重要内容之一。我国计算机专家学者通过科学实践，做出了大量成果，积累了丰富经验与学识。他们有撰写著作的很大积极性，但相当时期以来计算机学术著作由于印数不多，出版往往遇到不少困难，专业性越强越有深度的著作，出版难度越大。最近清华大学出版社与广西科学技术出版社为促进我国计算机科学技术及产业的发展，推动计算机科技著作的出版工作，特设立“计算机学术著作出版基金”，以支持我国计算机科技工作者撰写高水平的学术著作，并将资助出版的著作列为中国计算机学会的学术著作丛书。我们十分重视这件事，并已把它列为学会本届理事会的工作要点之一。我们希望这一系列丛书能对传播学术成果、交流学术思想、促进科技转化为生产力起到良好作用，能对我国计算机科技发展具有有益的导向意义，也希望我国广大学会会员和计算机科技工作者，包括海外工作和学习的神州学人积极投稿，出好这一系列丛书。

中国计算机学会

1992 年 4 月 20 日

目　　录

前　言

在过去的十多年里，软件可靠性理论从创立到发展，直至实际应用，经历了一个颇为有声有色的过程。关于这方面的论文，在国外已发表了许多，散见于许多杂志、会议录和专题讨论会的文集之中。近年来，国内理论界及软件工程界的不少有识之士，已经认识到在中国开展软件可靠性研究的意义及重要性、迫切性。目前，已有人正在进行或计划着手进行这方面的研究工作，这是一种可喜的局面。我们祝愿软件可靠性理论的研究及应用在国内取得蓬勃的发展。

在软件可靠性研究方面，我们与谢砜博士早在 1986 年夏就建立了学术上的联系。而与清华大学郑人杰教授的合作，则从 1984 年邀请他来武汉大学讲学时就已开始了。我应 Bo Bergman 教授和谢砜博士的邀请，1989 年 5 月至 7 月，在瑞典的林雪平大学(LINKÖPING UNIVERSITY)，开展合作研究，并完成了本书初稿的撰写工作。其中，自然也包括了郑人杰教授的许多工作。

为了让国内广大读者能尽快进入这一领域，我们将平日收集的资料及研究的心得体会，特别是在我从瑞典回国以后所从事的科研工作，以及近期内国外取得的新进展，汇总起来，我们的主旨是以实用为目的，无论是软件可靠性理论方面的内容，还是概率论与数理统计方面的知识，一律以实用为主来决定取舍。编写本书的目的在于抛砖引玉，以期推动国内关于软件可靠性的研究工作。

本书第一章为概论，主要讨论软件可靠性研究的意义及基本概念、软件可靠性模型的作用及意义，并概要地介绍了软件可靠性模型的发展概况。第二章从软件工程的角度介绍了软件的开发过程，它与软件可靠性研究的关系，以及软件可靠性与硬件可靠性的共性和各自的特点。第三章为可靠性分析数学的基础知识。我们仅以提纲式的介绍，为读者提供一个可供参考的简略材料。

第四章对软件可靠性模型进行概述，介绍了模型的特点、目前采用的几种模型分类方法。主要讨论了目前软件可靠性模型所使用的各种假设的局限性，并顺便介绍了 Musa 的执行时间概念。第五章介绍随机过程类的软件可靠性模型。它们包括：马尔可夫过程模型，主要介绍 Jelinski-Moranda 模型及其推广形式；非齐次泊松过程模型，以介绍 Goel-Okumoto 模型及其推广形式为主，兼及其它的非齐次泊松过程模型。特别，结合我们的研究工作，对非齐次泊松过程模型的参数估计、三参数模型的奇异性及改进方法、应用 EM 算法对模型参数进行调整，以及对非齐次泊松过程模型拟合质量的改进技术，都作了深入的讨论。另外，这一章还介绍了 Musa 的执行时间模型及对数泊松执行时间模型。对于它们，我们还尽量注意到给出应用的实例。第六章主要介绍非随机过程类模型，它们包括运用贝叶斯估计的贝叶斯模型，其中以 Littlewood-Verrall 模型为主要代表，也涉及贝叶斯理论应用于 JM 模型的许多推广；对 Seeding 模型、基于输入域的模型也作了相应的介绍。在这一章里还介绍了许多其它的方法，如：非参数分析、结构化模型、Cox 比例风险函

数模型以及 Singpurwalla 等人的时间序列分析方法。

第七章讨论了对于各种模型进行比较的标准和技术。主要介绍结合测试策略对模型进行选择的技术、预分析技术、以及 U-图、Y-图、PL 检验等一系列实用的比较技术。第八章讨论软件可靠性在软件规划管理中的作用。它的作用主要通过软件测试终止时间和最佳投放时间的决策,发挥它在软件开发中的经济效益。最后,第九章介绍了软件可靠性研究的现状与发展。对软件可靠性分配模型、X-件(软、硬件结合系统)的可靠性分析方法、恢复块结构技术,对软件测试过程的直接建模方法、测试中观察到的故障为零时的故障概率估计方法等一系列新的方法,特别,对于具有广泛应用前景的神经网络方法,作了详细的探讨。

为了使感兴趣的读者今后能进行更深入的了解和研究,书后列出了详细分类的参考文献,以使读者能按此线索深入下去。

本书适合于软件工程及软件可靠性分析专业(特别是软件可靠性工程专业)高年级本科生或研究生阅读,也适合广大软件工程界负责软件项目开发、分析的人员、软件质量高级管理人员、各类研究工作者及实际工作者阅读。

软件可靠性研究课题,武汉大学得到了中国国家自然科学基金委员会(The National Natural Science Foundation of China)资助,瑞典林雪平大学得到了瑞典技术发展基金会(STU)资助。乘本书出版机会,分别向这两个委员会表示感谢。

瑞典 Bo Bergman 教授、美国 J. D. Musa 先生、日本 S. Yamada 博士等,对我们的工作给予了可贵的帮助。特别,林雪平大学的 Bo Bergman 教授为我们的合作创造了条件。林雪平大学图书馆工作人员协助我们用计算机进行检索,为我们提供了不少最新资料,才使我们这本书得以反映最新的研究成果。在本书中,也包括了武汉大学软件工程研究所吴新玲、袁砾,以及武汉大学数学系柳春雷等同志的部分工作。在我们的研究工作中,还得到武汉大学张尧庭教授、胡泽民副教授的悉心指导。在此,一并向他们表示衷心的感谢。另外,我们还要感谢清华大学出版社的编辑同志,正是由于他们的辛勤努力,才使本书得以以现在的风貌呈现在读者面前。在本书最后定稿的过程中,得到航空航天部质量司何国伟总师、航空航天部二院二零四所王纬总师,以及刘忠信同志的关心和支持,特此向他们表示衷心的感谢。

本书错误及不足之处在所难免,敬请广大读者批评指正。

徐仁佐

1992.10

于清华园

数 学 符 号[①]

一、用于错误计数的记号

N_0:	在时刻 0(即测试一开始)的初始错误数。
$N_r(t)$:	在时刻 t 的剩余错误数。
$N_c(t)$:	到时刻 t 为止已改正的错误数。
$N_g(t)$:	到时刻 t 为止产生出的错误数。
$N_d(t)$, $N(t)$:	到时刻 t 为止已发现的错误数。
n_i:	第 i 个时间间隔内已发现的错误数。
N_i:	第 i 个时间间隔的开始时刻,具有风险的错误个数。对于某些特定模型,它的数值可能是常数或 0。

二、用于表示时间的记号

t:	时间的一般记法。
t_i:	第 i 个错误的出现时刻,或每个相连续的测试时间段的开始时刻。注意,对错误出现时刻作如下规定: $i > j \quad t_i > t_j$。
T_i:	第 i 个错误的出现时刻或删改时刻。
τ:	两个错误出现的间隔时间,有 $\tau = t_i - t_{i-1}$。

三、用于描述软件的记号

M:	模块数。
J_i:	在第 i 次测试时,被测模块的集合。
l_j:	模块 j 的代码行数,则全系统的代码行数为 $\sum_{j=1}^{M} l_j$

四、用于表示测试有关信息的记号

S:	测试运行次数。
S_i:	第 i 次测试中的测试运行次数。
m:	因出错而终止的测试运行次数。
m_i:	在第 i 次测试中,因出错而终止的测试运行次数。
r_j:	错误 j 出现的次数。

① 未列入的记号,凡正文中第一次出现一律予以说明。

r_{ji}: 在第 i 次测试中, 错误 j 出现的次数。

五、统计记号

$\{F_t\}$: 完全且右连续的不减 σ 代数族, 具体地说, F_t 是在时刻 t 的逻辑值假定已知的事件族。

L(·): 似然函数。

lnL(·): 自然对数似然函数。

$\Lambda(t)$: 在时刻 t 时的错误出现率。

Z(t): 风险函数。

Λ_0: 初始错误出现率(即 $\Lambda(0)$)。

m(t): 均值函数: $m(t)=\int_0^t \Lambda(s)ds$.

R(t|s): 对给定的时刻 t, 最后被观察到的故障是在时刻 s 时的系统可靠性。

α, β: 模型参数。

六、用于表示概率分布的记号

Exp( ): $f(x)=\alpha e^{-\alpha x}$

U(a, b): $f(x)=\begin{cases}\dfrac{1}{a-b}, & x\in[a,b]\\ 0, & \text{否则}\end{cases}$

Bin(a, b): $f(x)=\binom{a}{x}b^x(1-b)^{a-x}$

Γ(a, b): $f(x)=\dfrac{a^b x^{b-1}}{\Gamma(b)}e^{-ax}$

P(a): $f(x)=\dfrac{a^x}{x!}e^{-a}$

β(a, b): $f(x)=\dfrac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}(1-x)^{a-1}x^{b-1}$

β(a, b|k_1, k_2): $f(x)=\dfrac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\cdot\dfrac{(k_2-x)^{a-1}(x-k_1)^{b-1}}{(k_2-k_1)^{a+b-1}}$

反 Γ(a, b): $f(x)=\dfrac{a^b}{x^{b+1}}\cdot\dfrac{e^{-\frac{a}{x}}}{\Gamma(b)}$

如果一个或几个参数并不重要, 或目前对它们并不感兴趣, 则它们可以用圆点来代替, 如:

$x\sim U(\cdot,\cdot)$ 表示 x 具有正态分布, 但它的参数在目前并不重要。

七、关于马尔可夫过程的记号

p_{ij}: p(系统从状态 i 转向状态 j)。

$p_i(t)$: p(在时刻 t 处于状态 i)。

$\lambda(i,t)$: 在时刻 t, 状态 i 的死亡率。

$V(i,t)$: 在时刻 t, 状态 i 的诞生率。

当参数 i 被认为并不必要时, 可以省去。如: 当死亡率并不依赖于状态时, $\lambda(i,t)$ 可以写成 $\lambda(t)$。

$Q(t)$: 在区间 $[0,t]$ 内, 死亡的期待纯值, $Q(t) = \int_0^s [\lambda(s) - V(s)]ds$。

第一章　概　　论

计算机系统工程分为硬件和软件两大范畴。

计算机硬件的工程技术由电子设计发展而来,并且在过去的 30 多年中已经达到了相当成熟的状态。硬件设计技术已经很好地建立起来,硬件的制造方法一直在不断地改进,可靠性已是一种现实的要求,而不再是一种朴素的愿望。

而在软件工程技术方面,在以计算机为基础的系统中,软件已成为最难设计、最少可能成功(指在时间上以及成本方面),并且管理起来最危险的系统成分。随着以计算机为基础的系统在数量、复杂程度和应用方面的激增,对软件的需要却在不断增加,因此促使供求矛盾日趋激化。

世界上第一台电子计算机诞生以后,手工的程序设计方法就受到了严厉的指责,许多人都认为它严重地限制和妨碍了计算机的应用。从那时以来,计算机硬件经过了从第一代至第四代的发展,在硬件制造技术方面,人类已经获得了巨大的进展。现在,又在进行新一代计算机的研制工作。随着计算机系统越来越庞大,它的应用范围越来越广泛,计算机的软件系统也变得越来越复杂,随之而来的问题就是:大的软件系统成本高、可靠性差,甚至有时人的大脑已无法理解、无法驾驭人类本身所创造出来的复杂逻辑系统。例如:在 60 年代,美国 IBM 公司生产的 OS 360 花费了 5000 人-年的巨大代价,但是,由于它太庞大,OS 360 变得相当不可靠,平均每次修改以后的新版本都大约存在 1000 个左右的错误,并且有理由认为这是一个常数。另外,美国空军的范登堡中心在 60 年代后期发生过多次导弹试射失败的事故,事后检查几乎都是由于计算机的软件有错误而造成的。因此,由于人类大脑对复杂逻辑系统理解与控制的局限性,对先进的计算机硬设备的应用施加了一种无形的束缚。人类智力的不完善和不一致,在软件的开发过程中,势必会产生大量的错误。每一个这样的错误都被看作是与整个系统无关的单个逻辑缺陷,但是,经验证明:如果将它们看作一个整体的话,那么,它们就都涉及到一个共同的问题,即所谓的“软件危机”。许多软件系统的复杂程度已使它们变得无法管理,其自然的结果就是它们的开发周期延长,成本增加,可靠性降低。

软件危机是指在计算机软件开发中所遇到的一系列问题,而不仅限于指“不能正确地工作”的软件。确切地说,软件危机包含和下列问题有关的问题:我们怎样开发软件?我们怎样维护现有的、容量又在不断增加的软件?我们怎样做才能满足对软件不断增长的需求?

软件危机以许多问题为表征,而负责软件开发的管理人员则往往将注意力集中于“底线”问题:进度和成本的估计往往很不精确。实际中不乏这样的例子:成本超过计划的一个数量级;进度一拖就是几个月甚至几年。许多软件开发的困难明显表现在以下方面:

- 人们还没有时间去收集关于软件开发过程的数据。由于缺乏历史数据作为指南，所有关于进度和成本的估算都十分粗略，因为没有切实的生产率指标，我们无法精确评价新工具、新技术或新标准的效能。
- 经常发生用户不满足于“完全的”软件系统。软件开发项目往往在只有模糊的用户要求指示情况下就着手进行。用户与开发者之间的沟通常常是很糟的。
- 软件质量常常是很可疑的。人们只是最近才开始理解系统的、技术上完全的测试的重要性。软件可靠性和质量保证的切实定量概念还刚开始出现。
- 现有的软件可能很难维护。软件维护任务耗费了软件经费的主要部分。人们还没有真正把软件的可维护性作为验收的一个重要准则。

与软件系统的不可靠性密切联系的经济上的含意也是十分明显的：大约 1/3～1/2 的开发和维护软件系统的努力，都花费在测试和排错上面。因为在大计算机系统的运行过程中，绝大部分资源都是分配给软件的，而花在克服软件系统的不可靠性上面的直接开销，其总数也就大体上是计算机的全部开销了。进而言之，如果花在软件错误上的非直接开销(如整个计算机系统的收益上的损失等)也计算在内的话，那么，系统的不可靠性所带来的经济上的损失就变得更加惊人。

与软件危机有关的许多问题都起源于软件本身的特点、担负软件开发职责的人员的弱点、以及在软件开发的早期人们对于软件开发实质的种种不切实际的误解。软件危机不会很快消失，认识问题的本质所在及其产生的根源，是走向解决问题的第一步。

计算机软件最终可能成为以计算机为基础的系统发展中的限制因素。由于软件开发、管理和技术问题表现为软件危机，由于对软件和开发软件所需方法的诸多误解，管理问题将继续存在。而由于有了新开发的工具和技术，技术方面的问题可能解决。为了更多、更快、更好地开发计算机软件，在 1968 年前后产生了一种工程的方法——一种集成了具有最好技术的、经过考验的、关于控制的管理技术的方法，即软件工程的方法。

由于软件开发过程是知识密集型的过程，在软件开发过程之前、之中、之后，都要运用大量的知识，它们主要包括：

- 有关软件工程技巧的知识
- 有关程序设计技巧的知识
- 有关程序设计语言的知识
- 有关计算机硬件的知识(即机器知识)
- 有关应用领域的知识
- 有关软件开发历史的知识

但是，只靠现有的软件工程方法并不可能解决软件危机的根本问题。于是，有人又提出了基于知识的软件工程方法的设想，力求将软件工程与知识工程、人工智能技术结合起来，以构造基于知识的软件开发环境。

§ 1.1 软件可靠性研究的意义和基本概念

计算机硬件的性能价格比以每十年大约1000的速率在增加。信息处理系统的应用日趋广泛,由此刺激了软件工程技术的进步,以应付日益膨胀的软件需求。具体地说,影响到当今软件工程发展的因素主要有:

1. 国际上商业竞争的势态越来越激烈。
2. 信息处理系统的开发成本,以及由于它们发生故障而造成的经济损失也正在增加。
3. 计算技术的发展速度越来越快。
4. 对于信息系统开发的管理变得更加复杂。

绝大多数信息系统被用于商业,而今天商业上竞争的激烈程度使许多信息系统的顾客真正认识到他们对于软件产品的需要。而存在于软件产业界的激烈竞争,又使他们进一步认识到软件产品连同一起提供的有效服务对于他们自身利益的意义。他们曾经很天真地完全信赖提供软件产品的公司或厂家,但现在他们对于软件产品的质量以及厂家提供的服务,也变得越来越苛求和挑剔。软件开发者们应充分而清楚地认识到用户的需要主要集中于这样三个方面:质量要求、交货时间和开发成本。与此同时,软件的开发和运行成本实际上也在增加,且软件产品的大小、复杂性和分布式的程度也都在不断增加。目前已有越来越多的计算机系统由网络连接起来,有许多不同种类的软件同时在运行并相互发生作用,其直接的结果就是使开发成本变得更大。信息系统应用的激增更增加了对上述系统的依赖性,这一依赖性已发展到要依赖于更小、更低级的组织结构,并且以实时方式工作的系统的比例也在增加。于是,发生故障以后造成的损失更大、更具关键性。这样的事例很多,如:飞机公司预订票系统的失败,银行计算机系统的故障,自动飞行控制系统的失灵,导弹控制系统的失误,核电站安全系统的故障,等等,它们造成的经济损失是巨大的,有的甚至是后果不堪设想的。成本还不仅只包含那些直接的开销,而且还包括对产品开发债务风险所承担的经济赔偿以及对公司声誉的赔偿金。而后一种赔偿往往对公司占领市场及所得利润产生戏剧性的影响。

计算技术的飞快变化使得许多信息系统很快就变得过时了,这样淘汰旧产品也会使软件产业承受很大的经济损失。使软件产品很快变得落后的威胁主要来自两个方面:硬件技术的发展和软件技术的进步。因此软件产品的交货时间就变得十分重要,谁也不希望在软件产品还未交付使用之前,运行该软件系统的硬件环境就已被淘汰。人们必须充分认识到将一个新的软件产品打入市场的有效时限变得越来越短暂了。

基于上述原因,一个软件开发公司如果企图同时提供开发周期短而且成本低廉的高质量软件产品,几乎是不可能的。要认识到,在当前科学技术飞速进步的情况下,这种想法

早已陈旧不堪。

随着软件系统的大小及复杂性的增加,对于信息系统开发的管理也变得更加复杂。现在许多软件系统都被划分成许多部件,由不同的公司来进行开发,这样就要求对系统以及它的部件的特性有清晰的了解,并且对于开发期间的进度要有明确的表示。从管理、签约及法律的观点来看,对于所有特性的了解都是重要的。而且其结果是对软件产品特性以及开发期间对那些特性的当前状态的测量和预测,都变得更重要了。

上面已叙述过软件产品最重要的三个特性是质量、成本和进度,它们基本上是面向用户的而不是面向开发者的属性。其中后两个特性比较容易把握,而软件质量的度量则要困难得多。鉴于目前仍然缺乏度量软件质量的有效手段,人们自然把注意力集中于此。事实上,这种对软件产品质量的定量测量手段的缺乏,正是许多软件产品存在重要质量问题的基本原因。

在软件质量的范畴内,或许最重要的固有特性之一就是系统的可靠性,它关心的问题主要是存在于软件产品中的缺陷,而正如 Jones 在 1986 年指出的那样,软件系统中的缺陷代表了程序设计过程中最大的成本因素。软件可靠性关心的是软件本身的功能如何才能最大限度地满足软件用户的要求。

现在有不少人都在热切地追求软件可靠性的目标,不过大多都只停留在理论的抽象上,却很少有人将它表示成为一个程序本身所具有的属性。理论上的抽象是可以做得十分完美的,但是在软件产品开发过程中,实际上为获取更高的软件可靠性目标,其作法通常是以软件产品本身的某些其它特性(如程序大小、程序的运行时间或程序的响应时间、软件产品本身的可维护性,等等)作为代价来换取的;或者是在软件产品的开发过程中,采取诸如:增加开发成本、增加对资源的要求、推迟生产的进度等措施来换取的。B oehm 等人已将人们希望予以考虑的某些一般的软件生产的特性进行了分类。人们在软件产品的生产过程中,一旦想要对这些因素权衡取舍,以取得一个折衷的方案时,软件可靠性就是一个首先必须加以考虑的因素。在某些情形下,软件可靠性甚至是更一般的特性之一。比如说,考虑实时系统的工作情况,要使用户的要求得到满足或提高用户的工作效率,软件可靠性就是一个很重要的因素。

为达到人们期望的软件可靠性指标,必须在软件产品的特性中,以及在软件产品的生产过程的特性中进行适当的取舍。而这样的取舍,首先得由系统设计师们来作。所以关于软件可靠性的定量评估技术,在他们看来就具有特别重要的意义。当然,这些技术对于软件工程师和软件生产的管理人员而言,也是同样重要的。

软件系统在特定的环境(条件)下,在给定的时间内,不发生故障地工作的概率,称为该软件系统的可靠度。而这种性质,就称为软件系统的可靠性,亦简称为软件可靠性。

以 E 表示特定的环境,t 表示给定的时间,设系统从时间 0 开始运行,直到 T 时发生故障,则:

$$R(E,t) = P_r\{T > t\}$$

就表示软件系统在特定的环境 E 下的、正常工作到时刻 t 时的概率,其中,T 是从时刻 0 开始,软件系统运行到发生故障时的时间。

R(E,t)具有下列的性质:

$R(E,0)=1$; 即在 0 时刻, 系统绝对不发生故障。

$R(E,+\infty)=0$; 即在无限远的时刻, 系统必定失败。

在时间区间$(0,+\infty)$上, 函数 $R(E,t)$ 是单调下降的。

按照概率论的观点,“软件系统正常工作”是一事件, 则它的对立事件“软件系统运行出错”定义了一个故障概率函数: $Q(E,t)$, 因而显然有:

$$R(E,t)+Q(E,t)=1.$$

通常, 特定的环境 E 要从描述测量的上下文去了解, 而毋须确切地给出, 因此, 可以把可靠度函数 $R(E,t)$ 简写为 $R(t)$, 而把故障概率函数 $Q(E,t)$ 简写为 $Q(t)$。因此, 有:

$$R(t)+Q(t)=1,$$

$$R(t)=1-Q(t)=P_r\{T>t\},$$

其中 T 的意义同上。

硬件可靠性的概念是十分明显的, 它充分体现在对元器件的产品质量检查的过程之中。但是, 软件产品可靠性定义中的概率性质体现在什么地方呢? 不可能说我们在一批生产出来的相同的软件产品中, 随机地抽取出若干个来进行测试。一般可以这样说, 软件产品可靠性中的概率性质主要体现在输入的选取上。

如果把程序看作输入空间到输出空间的映射, 那么程序运行出错就是由于程序没有将某些输入映射到为人们所期望的输出上去。设: 输入空间一共有 I 个点, 若点 i 为输入时程序运行正确, 引入一个程序执行变量 Y_i 如下:

$$Y(i)=\begin{cases}1, & \text{输入点 } i \text{ 时, 程序运行正确;}\\ 0, & \text{否则。}\end{cases}$$

在特定的应用中, 设 $P(i)$ 为输入点 i 的概率, 则在这一特定的应用中的一次输入导致程序正确运行的概率为:

$$\sum_{i=1}^{I}P(i)Y(i).$$

因此就有:

$$R(t)=\left[\sum_{i=1}^{I}P(i)Y(i)\right]^n,$$

其中, n 是在时间区间$(0,t)$内程序总共运行的次数。

值得注意的是: 上面我们提及的“一次运行”, 也是一个有时很难以确定或量化的含糊概念。

上面所述描述软件可靠度的方法, 的确从本质上反映了软件可靠度定义的概率性质, 但是它对实际确定软件的可靠度并无任何意义。这主要是因为: 第一, 输入空间的大小 I 即使不是无穷大, 通常也是十分大的数字; 第二, 也是更重要的一点就是, 在某一特定的应用中如何来确定 $P(i)$ 的大小, 在上面所叙述的方法中并未明确指出, 而且, 事实上要真正确定 $P(i)$ 的大小, 也是一件十分困难的工作。因此一种更为实际的, 基于运行的软件可靠度定义可以这样来描述:

设 n 表示在一特定应用中程序实际运行的次数, c_n 表示在这 n 次运行中正确运行的次数, 则:

$\lim_{n\to\infty}\frac{C_n}{n}$就表示一次运行正确的概率。于是有:

$$R(t) = \lim_{n\to\infty}\left(\frac{C_n}{n}\right)^r,$$

其中,r 是在时间区间$(0, t)$内,程序运行的总次数。

故障率,也有人称它为风险函数(hazard function),也是一个直接源于硬件可靠性的术语。

一般地说,我们用 $\lambda(t)$ 来表示风险函数,$\lambda(t)$ 就是程序正确地运行到时刻 t 时,单位时间内程序发生故障的概率(实际上,$\lambda(t)$ 应该是概率密度,真正的概率应该是 $\lambda(t)\cdot\Delta t$)。

如果以 T 表示从 0 开始运行一程序,到程序发生故障为止所经过的时间。则对于不同的运行,T 的值显然是不同的。因而可以断定,T 是一个连续型的随机变量。于是有:

$$\lambda(t)\Delta t = P_r\{t < T \leqslant t + \Delta t \mid T > t\},$$

因此,我们又可以写:

$$\begin{aligned} R(t+\Delta t) &= P_r\{T > (t+\Delta t)\} \\ &= P_r\{(T > t) \wedge \text{在区间}[t, t+\Delta t]\ \text{内程序不发生故障}\} \\ &= R(t)[1 - \lambda(t)\Delta t] \end{aligned} \tag{1.1.1}$$

对等式(1.1.1)关于 t 求微分,我们得到:

$$dR = -\lambda(t)R(t)\,dt.$$

从而可以导出下面的微分方程:

$$\frac{dR}{R} = -\lambda(t)\,dt \tag{1.1.2}$$

解之,得:

$$R(t) = \exp\left[-\int_0^t \lambda(x)\,dx\right] \tag{1.1.3}$$

上式描述了风险函数 $\lambda(t)$ 与可靠度函数 $R(t)$ 的关系。

风险函数 $\lambda(t)$ 在软件可靠性的研究中,受到了广泛的关注。因为从式(1.1.3)我们可以看出,一旦知道了 $\lambda(t)$,则 $R(t)$即可计算出来。但是在实际的工作中,却十分令人失望,直到目前为止,人们还只能对它作出形形色色的假设。

有的研究者认为,存在于软件中的错误,它们引起软件在运行过程中发生故障的可能性都是相同的,因此有理由认为 $\lambda(t)$ 是一个常数。但是,在我们的实践中,经常可以观察到这样笼统地假设 $\lambda(t)$ 为一个常数的作法与事实不符。不过可以认为 $\lambda(t)$ 在分段的时间内是一个常数。

联系到排错的实际过程来看,软件中的错误随着排错的进展,不断地有错误相继地被发现、被纠正、被排除,因此可以乐观地认为,存在于软件中的错误是越来越少了。与之相适应的一个事实就是:发生故障的频率也会随之下降。因此,有理由说:$\lambda(t)$ 本质上是一个减函数。

然而,无论是常数论者也好,还是减函数论者也好,他们的共同点在于,不会认为 $\lambda(t)$ 是一个增函数,并且也没有人能够具体地给出一个实实在在的风险函数的表达式。这并不是说我们的数学家们以及他们所掌握的数学工具的无能,而是因为软件在运行的过程中发生故障的诱因多种多样,影响到风险函数的因素过于复杂的缘故。

正是由于这样一个原因,才决定了目前在软件可靠性评估的研究工作中,一个普遍的作法就是:每一个估测模型的作者在提出模型之前,都不得不首先提出一系列的假设,然后在这些假设的基础上来开展工作。当然,他们作出的假设必须具有一定的事实作为基础。然而有的假设在实际的测试、排错过程中与事实有些出入的现象也时有发生,但为了在数学上处理起来容易,暂时认为它们还是成立的。这似乎已成为一种默契。

§ 1.2 软件可靠性模型的作用及意义

软件可靠性模型是随机过程的一种表示,通过这一表示,可以将软件可靠性或与软件可靠性直接有关的量,如:平均无故障时间或故障率等,表示成时间以及软件产品的特性,或者开发过程的函数。软件可靠性模型通常描述了软件可靠性对上述各变量的一种依赖关系。其描述的形式则根据通常由已知的故障数据出发所作的统计推断过程而定。对于软件的模块、子系统、系统都可以应用软件可靠性模型。

为了给软件可靠性的估测建立数学模型,应首先考虑影响到估测的基本要素。它们是:错误引入软件的方式、排错的实际过程以及程序运行的环境。错误的引入主要依赖于已开发出的程序代码的特性(程序代码开发出来主要是为了实际的应用,以及对程序的修改过程),以及开发过程的特性。最明显的代码特性就是程序代码的长度。开发过程特性包括:软件工程技术,使用的工具,以及开发者个人的业务经历。有一点必须注意的是:程序既可以开发出来用以增加软件的功能,也可以开发出来用于排错。排错依赖于用于排错的时间、排错时的运行环境、用于排错的输入数据、以及修复行为的质量。环境则直接依赖于操作剖面。操作剖面(Operational Profile)主要指各种类型的运行出现的概率,而它们则由各自的输入状态描述其各种运行的性质。因为上述各基本要素大多具有随机性且均与时间有关,所以软件可靠性估测模型大多都处理成随机过程的形式。模型与模型之间,大多根据故障时间或已发生的故障次数的概率分布来加以区分,也可以根据与时间有关的随机过程的不同处理方式来加以区分。

一个好的软件可靠性模型还应描述出上述诸基本要素间的总的故障过程的依赖关系。根据定义,我们已假定模型以时间为基础(这并不等于说不以时间为基础的模型就不能用)。用不同的数学公式描述故障过程的可能性是几乎没有什么限制的。对于各种不同的数学表达式,我们可以通过建立模型参数的有效数据来加以判定。这样就有下面两种方式可以采用:

(1) 估计模型参数——将统计推断过程应用于程序运行过程中产生的故障数据;
(2) 预测程序将来的故障行为——由软件产品的特性和它的开发过程来进行判断(这可以在程序的任何执行之前进行)。

在对数学表达式进行判定时,总存在着某种不确定性。可以考虑采用置信区间的概念

来予以解决。置信区间就是指对于给定的一个确定的置信度,待估计的参数值确切地可以落在某个范围之内。

一旦确立了数学表达式,对许多不同的故障特性都能加以判定。对于各种不同的模型,它们分别采用了许多不同的表达式:

(1) 在任一时刻所发生的故障的平均数;
(2) 在一时间区间内发生的故障的平均数;
(3) 在任一时刻的故障密度;
(4) 故障间隔的概率分布。

一个好的软件可靠性模型应有一系列重要的性质:

(1) 对于将来的故障行为能给出好的预测;
(2) 对有用的量能进行计算;
(3) 简单明了;
(4) 具有广泛的应用;
(5) 它应在合理的、与实际情况完全吻合的或十分接近的假设基础上作出。

在对将来的故障行为进行预测时,应保证模型参数的值不发生变化。如果在进行预测时发现引入了新的错误,或修复行为使新的故障不断发生,就应停止预测,并等至足够多的故障出现以后,再重新进行模型参数的估计。否则,这样的变化会因为增加问题的复杂程度而使模型的实用性降低。

一般说来,软件可靠性模型是以在固定不变的运行环境中运行的不变的程序作为估测实体的。这也就是说,程序的代码和操作剖面都不发生变化。但它们往往总要发生变化的,于是在这种情况之下,就应采取分段处理的方式来进行工作。因此,模型主要地就要集中注意力于排错。但是,也有的模型具有能处理缓慢地引进错误的情况的能力。

对于一个已发行并正在运行的程序,应暂缓安装新的功能和对下一次发行的版本的修复。如果能保持一个不变的操作剖面,则程序的故障密度将显示为一个常数。

一般说来,一个好的软件可靠性模型增加了关于开发项目的通讯,并对了解软件开发过程提供了一个共同的工作基础。它也增加了管理的透明度和其它令人感兴趣的东西。即使在特殊的情况之下,通过模型作出的预测并不是很精确的话,上面的这些优点也仍然是明显而有价值的。

实际建立一个有用的软件可靠性模型,一般包括:坚实的理论研究工作、有关工具的建造、实际工作经验的积累。通常这些工作要求许多人一年的工作量。相反,要应用一个好的软件可靠性模型,则要求极少的项目资源就可以在实际工作中产生好的效益。

§ 1.3 软件可靠性模型发展简述

最早的模型出现在 1956 年, 由 H. K. Weiss 提出了一系列的公式。但由于它们太复杂了, 这些公式对以后软件可靠性模型的建立几乎没有产生什么影响。

1967 年, Hudson 观察到软件的开发过程是一个生灭过程, 一个典型的马尔可夫过程。其中错误的产生是诞生期, 错误的改正是死亡期, 在任一时刻存在于软件中的错误个数可用来定义过程的状态, 状态的转移概率则与生灭函数有关。为了在数学上容易处理, 他的工作只限于研究纯死亡过程方面。他假设: 错误改正率与剩余错误个数成正比, 还与时间的某个正次幂成正比, 即, 错误改正率随时间的增加而增加。由此, 他得出故障间隔 (Interval Between Failures) 的 Weibull 分布。同时他还发表了由系统测试阶段收集的数据, 并声称: 如果将系统测试分为三个既互相衔接又适当分离的子阶段, 就可获得模型与数据之间合理的一致性。

对于软件可靠性模型发展首次起到较重要作用的两个模型, 发表于 1971 年。Shooman 模型由 M. L. Shooman 发表, J-M 模型由 Z. Jelinski 和 P. B. Moranda 发表。它们有着惊人的相似之处, 且 Shooman 在他稍晚的一篇文章中指出, 存在着一个简单的参数转换集可以将 J-M 模型转换成 Shooman 模型。它们都假设:

- 软件中的初始错误数为 N(N≥0)。
- 故障率与软件中的剩余错误个数成正比。
- 一个错误一旦被发现, 立即排除且排错不引入新的错误。

这些假设都被随后许多软件可靠性模型以各种方式所采用。

另外, J-M 模型还假设故障的风险率为分段常数, 它在每次改错过程中由一个常量来改变, 但在两次改错过程之间保持常数。Jelinski 与 Moranda 应用最大似然估计来估计软件中的总错误个数、剩余错误数与风险率之间的比例常数。

1972 年, B. Littlewood 和 J. L. Verrall 发表了第一个贝叶斯模型, 他假设: 故障间隔时间服从含参数 λ 的指数分布, 且 λ 服从先验的 Γ-分布。据此由标准的贝叶斯过程可以获得 t_{n+1}©{$t_1, \cdots, t_n$}。同年, G. J. Schick 与 R. W. Wolverton 提出的模型与其它模型的主要区别就在于: 他们假定连续的故障之间的排错时间服从 Rayleigh 分布。1973, W. L. Wagoner 发表的模型与此相类似, 但假设了风险函数服从 Weibull 分布。它的特例就是 Schick-Wolverton 模型。

J. D. Musa 在 1975 年发表了执行时间模型, 引入了一系列新的显式参数:

- 测试压缩因子, 以表述这样一种思想: 使用测试条件和数据(Test Cases), 比使用用户的输入数据有着更高引起故障发生的概率。
- 关于软件系统的初始 MTTF。
- 与日历时间相对的执行时间(即 CPU 时间)。

同年, P. B. Moranda 发表了几何泊松模型。他认为, 故障率随时间的增加以几何级数

下降,并且下降的过程出现在每次故障的纠正期间。因此,他假设在第 i 个时间段内的错误数满足含参数 λK^{i-1} 的泊松分布。

同年,K. Trividi 和 M. L. Shooman 还发表了第一个马尔可夫方式的模型。它将系统区分为"Up"状态与"Down"状态:工作正常与需要修复。所有的 Up 状态与 Down 状态间的转换概率都被假定为相同。模型的输出结果是一个概率的集合,其中每个的值都是:

P(在[0,t] 内找出 k 个错误)。

还是在这一年,N. F. Schneidewind 发表了第一个非齐次泊松过程(NHPP)模型。他建议对不同的可靠性函数,如:指数函数、正态函数、Γ-函数、Weibull 函数等进行研究,并针对开发的具体软件项目,选用最适合于实际问题要求的可靠性函数,以估测软件系统的可靠度。他建议在可靠性估测过程中,可从实际的数据出发,用经常校正时标的方式来判定查明故障到纠正故障之间的时延。他使用离散的时间步以构造模型。

1976 年,M. L. Shooman 和 S. Natarajan 首次发表了考虑到排错时引入新错的模型。他们使用查错率、改错率以及引入新错率来处理新错的引入问题。

1979 年,A. Goel 和 K. Okumoto 关于连续时间的 NHPP 模型,对软件界产生了持续的影响。

B. Littlewood 采用贝叶斯方法以研究软件可靠性建模。他认为,在具有常数风险率和程序操作期间,故障的发生都是随机的,但却将风险率当作一个已经发生的故障的随机过程。因此,在 Littlewood 的模型中,风险率是一个有条件的概念。1980 年,在他发表的微分模型中,假设对于程序的风险率取不同的基值,因此不同的错误将以不同的频率出现。

1983 年,由 Yamada, Ohba 和 Osaki 发表的一个 NHPP 模型,给出呈 S-形的可靠性增长曲线的均值函数。由 K. Okumoto 和 J. D. Musa 提出的对数泊松过程模型,具有一个初始错误率 λ 以及对应于 λ 的一个递减率。模型的日历时间部分则与执行时间模型的日历时间部分相同。

T. Bendell 将试探性数据分析法(EDA)引入软件可靠性评估,且使用时间序列分析法以及成比例的风险函数建立软件可靠性模型。

1989 年,Y. Tohma, K. Tokunaga, S. Nagase 和 Y. Murata 提出一个新的、以超几何分布为基础的模型,用于估计软件中的剩余错误个数。

Tsu-Feng Ho, Wah-Chun Chan 和 Chyan-Goei Chung 提出一个模块结构模型,通过每个模块的可靠性来估计软件系统的可靠性。另外他们还提出了结合信息论的方法,来建立软件可靠性模型的方法。

Y. Masuda 等人认为,在软件执行期间的任一时刻,只有 k 个模块(k 为常数,即一软件中含有的模块总数)中的一个在执行。他们据此开发出根据软件的模块结构判定最佳投放时间的模型。而 H. Ohtera 和 S. Yamada 则不仅考虑测试,而且也考虑到在运行期间的查错过程中,根据软件可靠性目标和平均无故障时间指标,来判定软件的最佳投放时间。N. D. Singpurwalla 以如何在不确定的情况下进行决策的思想为基础,提出两个判定在软件投放前应测试和排错多长时间,且使软件的实用性达到最大的实用性函数(utility functions),而它们分别以成本指标和软件可靠性指标作为控制。他指出,根据软件可靠性的概率模型使用于软件故障数据的结果,关于单个状态测试的情况,该最优化问题可以用

非数值化的技术求解。

硬件可靠性分配技术已成熟,但软件可靠性的分配则缺乏成熟的技术。F. Zahedi 和 N. Ashrafi 采用了对确立的系统划分层次的方法,应用分析分层过程(AHP: Analytic Hierarchy Process)以推定需要的模型参数。她们的模型具有非线性规划的形式,在模块和程序级别上考虑软件可靠性的各种技术和经济上的约束条件的同时,使软件系统的实用性达到最大。求解这一非线性规划问题,则可确定在模块和程序级别上的软件可靠性的分配方案。

现代的电子设备系统,既有硬件,也有软件,如何评估它们的可靠性,问题难度更大。J. C. Laprie 和 K. Kanoun 在这一方面有杰出的工作。

G. Pucci 应用恢复块结构技术,对软件可靠性等软件质量进行估计。而且根据错误对软件系统所产生的可观察到的后果进行分类,使得将测试数据应用于模型参数的估计成为可能。T. Downs 针对现有大多数软件可靠性模型在测试阶段将软件处理成黑盒子的作法,考虑对测试过程直接建模。

K. W. Miller 等人,开发出以软件的黑盒子模型为基础的理论分析方法,解决:(1) 在随机测试过程中,当观察到的故障次数为零时,如何估计当前版本的故障概率;(2) 在使用分布(use distribution)与测试分布(test distribution)不匹配时,对估计的故障概率进行调整;(3) 在估计故障概率时,将随机测试的结果与其它的信息结合起来进行分析。

N. Karunanith, Y. Malaiya 和 D. Whitley 应用神经网络系统理论预测软件可靠性。他们利用前向神经网络(a feed-forward neural network)对三个数据集合进行软件可靠性估测,结果发现:(1) 神经网络方法在软件可靠性估测中显示出良好的一致性,这是一般现有软件可靠性模型所无法做到的。因此,神经网络方法对于提高估测精度有着极大的贡献;(2) 由神经网络方法估计出的软件中的错误个数,始终较使用现有其它软件可靠性模型所估计出的结果要少;(3) 神经网络方法的估测能力,可通过由进行分析的数据,与其它的模型进行比较获知。

第二章　软件开发与软件可靠性

本章将简要介绍软件产品的一般开发过程,对软件中的错误做一些粗略的分析并进而对它们分类,概要地列举一些常用的软件度量,最后讨论一下软件可靠性与硬件可靠性的区别。

§ 2.1　软件的开发过程

传统的软件寿命周期,是包含了软件开始开发以前和它进入实际使用以后所出现的各种活动的一个长远观点。概略地讲,软件生存周期主要包括:软件计划阶段、软件开发阶段和软件维护阶段。

一般地,软件总是一个更大的以计算机为基础的系统的一个部分,所以,系统分析和定义必须先于(或至少是同时开始)软件计划。必须把系统功能分配给软件。

软件计划阶段以软件计划开始,对软件的作用范围必须作出描述,预测开发这一软件所要求的必不可少的资源,并且要作出代价和进度的初步估算。

软件计划的下一步任务是软件要求分析和定义,它要达到的目标主要有:

- 通过揭露信息流程和结构以提供软件开发的基础;
- 通过标识接口细节,提出深入的功能说明书以描述软件,确定设计约束和定义软件有效性要求;
- 建立和保持与用户及要求者的通讯,以便达到上述两个目标。

软件要求分析可以分成四个方面的工作:问题识别、评价和综合、写出规格说明书、复审。软件要求分析是软件计划阶段的关键一步,在这一步要完成的任务是将用户和要求者关于软件的模糊概念变换成一个具体的、有条理的、严谨的、无二义性的规格说明,这个规格说明乃是以后软件开发的基础,随后的一切开发活动皆以此为依据而展开。因此要求系统分析员具有敏锐的观察和综合问题的能力、由个别到一般的逻辑思维能力、由具体到抽象的分析归纳能力,以及用准确的语言与用户和要求者通讯并将这一切写下来的能力。

软件开发阶段是软件寿命周期中的中心阶段,关于软件的规格说明书一经确定,这一阶段的工作就应立即开始。

软件开发阶段的任务主要分为:初步设计、详细设计、编码、测试四大部分。

在软件的设计阶段,人们开发出各种行之有效的技术和方法,如:自顶向下、逐步精化、结构化设计方法,等等。许多研究者也提出了一些把数据流或数据结构翻译成设计定义的方法。

初步设计建立软件的模块结构,详细设计则完成所有必要的过程细节,给出设计表示,为下一步的编码打下基础,使之能据此直接而简单地导出源代码。

详细设计阶段可以利用诸如:图形工具、表格工具、语言工具等不同类型的设计工具,以完成设计描述。

编码阶段是一个翻译过程,将详细设计翻译成程序设计语言,最终由计算机自动地(利用与选用的程序设计语言对应的编译程序)转换成机器可执行的指令。风格是源代码的一个重要特性,简明性和清晰性是关键,即使牺牲一些执行效率和存储空间也是值得的。

软件测试是软件质量保证的关键一步,它的重要性及其与可靠性的密切联系,怎么强调也不过分。开发软件系统涉及到一系列的生产活动,人们在每一步中犯错误的机会多得举不胜举,人们无法完美无缺地行动和彼此通讯,因此,如何保证软件的质量是一个十分紧迫而重大的任务。软件测试则包括了对规格说明书、设计表示以及编码的最终复审。由测试所提供的有关软件系统的故障数据,则构成了进行软件可靠性分析的基础。

软件维护被有的人形容为漂浮在海洋中的一座冰山,大量潜在的问题和代价被隐藏着。变化是引起一切问题的根本。计算机程序总是在变动的。要纠错,要增补新的功能,要优化原有的功能,这一切都要求变动,而变动本身又引起一些新的问题。软件维护一般分为:校正性维护,适应性维护,完善性维护,预防性维护四种方式。为了纠正在软件使用以后所暴露的错误,就需要进行校正性维护;当外部环境的改变促使对软件进行修改时,就应进行适应性维护;完善性维护主要指由于用户集团所要求的对软件拥有功能的增补;为改进软件将来的可维护性及可靠性,并为将来的功能的进一步增补提供一个基础,我们必须采取预防性维护措施。

软件的开发除了传统的寿命周期方法以外,人们又提出并研究了诸如:快速原型法、面向对象的程序设计方法,以及计算机程序的自动生成等开发方法。有的已取得重要进展,有的则代表了新的方向。

关于软件开发过程的详细叙述,读者可参阅 R. S. Pressman: Software Engineering: A Practitioner's Approach.

§ 2.2 软件中的错误及其分类

对可能出现在一软件系统中的错误进行分类,有利于软件可靠性分析工作的进行。出现于各类参考文献中的用词不统一,在语义上间或也会产生一些混乱。在此列举一些术语并加以解释,同时力图在本书中采取前后一致的用法。

- 错误(error):在某系统中,人们原来所期望的以及系统实际具有的状态或行为之间的偏差。
- 故障(failure):在软件运行期间出现的错误,它是因软件中存在的错误引起的、在执行期间的动态表现。
- 缺陷(fault): 泛指系统中的所有错误。
- 过失(mistake): 人类在设计、运行系统过程中所犯的错误。在软件中一切错误皆因此而产生。

我们如果要区分一个错误的影响和一个错误的起因之间的区别,我们就要讨论错误的征兆和错误的诱因。

仅仅当状态与预先定义的标准发生了偏差时,人们才能辨认出一个错误的存在,因此错误也只是一个相对性的概念,特别在缺少明确而严格的、统一的标准时,它的意义就会显得十分含糊不清。在对给定的状态进行评价期间,当应用那些事先制定的标准的时候,将这些状态本身进行正确的分类,或者错误地依据制定的标准而行事,都是可能的。

在开发软件系统的过程中,一个可供参考的标准可以来源于系统说明,在这里当然指的是功能要求标准和系统的认可、接收标准。如果不存在明晰而不含糊的系统说明,那么,在开发软件系统的接收过程中间,关于系统功能的实现,人们就可能以一个模糊不清的假设去取代一个客观的标准。然而,这样做的结果,在许多情况之下往往都会使人们对具体的问题产生严重的误解。

软件错误和硬件错误

首先,最普通的分类方式就是将错误划分成软件错误和硬件错误。软件错误包括由系统分析或者程序设计而产生的全部错误;硬件错误则是由于机器的失灵而产生的错误。虽然乍看起来这一区分是明确的、毫不含糊的,但是,许多实际存在的难以区分的模棱两可的情况,增加了将错误按此法分类的难度。随着技术的进步,硬软件之间的界线变得越来越不清楚了,人们在构造计算机系统时,越来越多地采用了“软件硬化”以及“硬件软化”的技术。如:中央处理机中只读存储器中的微程序,它的一个错误在许多情况下,都会引起一条硬件指令的错误执行。存在于微程序中的错误,一般都是在进行微程序设计时产生的,照理说,它应该划入软件错误的范畴,但是,它引起的都是一条硬件指令的错误执行,是否又该将它划归硬件错误的范畴呢?

根据错误的征兆来进行分类

根据错误直接显示出来的征兆对错误进行分类,是特别有意义的。因为这样一种分类方法可以直截了当地进行,而无须事先进行那些不得不进行的、而且既困难又使得问题更趋复杂化的错误诊断。当我们把对错误分类的问题与软件系统的输入空间联系起来考虑时,我们可以根据错误的征兆,按表 2.1 所示的方式对错误进行分类。在这种分类的方式中,某种类型的错误所引起的实际后果,与特殊的应用项目的要求是密切相关的,也就是说,实际后果紧密地依赖于该应用的具体要求。一般地说,第Ⅰ种类型的错误(即因系统本

身辨认出不能够处理其正常的、但超出预先规定的输入范围的输入，因而对该输入加以拒绝)是并不如其他各种类型(第Ⅱ—Ⅴ种)的错误那么严重的。但是，这一点也有一个例外的情况，我们可以在实时系统中找到。在一个预先确切规定的时间范围之内，系统必须对任何的输入做出反应。因为在实时系统中，系统对某个输入的拒绝反应将可能导致一个灾难性的后果。

表 2.1　根据查错并联系到输入范围的输入对错误进行分类

输入 / 系统反应	超出预先规定的输入范围(输入错误)	在预先规定的输入范围之内
对输入的拒绝(系统查错)	正确	第Ⅰ种类型的错误
得出错误结果(系统外查错)	第Ⅱ种类型的错误(严重)	第Ⅲ种类型的错误(严重)
系统崩溃	第Ⅳ种类型的错误(严重)	第Ⅴ种类型的错误(严重)

根据错误的起因对错误进行分类

在计算机系统中的每一个错误，究其产生的原因，大都可以归纳为下列几种情况之一：在硬件或软件中的错误设计；由于环境或部件的老化所引起的硬件恶化；错误的输入数据(包括操作人员的失误或者实际的输入数据是错误的)。设计错误、硬件的恶化和数据错误是构成错误空间的三个互相垂直的轴，如图 2.1 所示。

将错误划分成这三个部分的错误分类法，是以错误的起因作为基点的，而并非以错误所产生的影响作为基点。因此，这一分类方法必须以一个完全的、成功的错误诊断作为它的先决条件。

图 2.1　计算机系统的错误空间

在一个设计错误出现的时候，任凭所有的硬部件的操作都是正确的(这只要参照它们的说明，是很容易判别出来的)，任凭输入数据都是正确的，但是系统总不可能产生出人们所企望 的正确计算结果来。

在一个系统部件由于恶化而出现错误时(主要由于元部件的老化或者环境因素的影响)，系统所表现出来的状态，在说明中是不会有相应的叙述的。设计错误和元部件的老化，在硬件中也都会出现，但是，在软件中，仅仅会出现的只有设计错误这一项。

起源于系统的不正确操作而发生的一个输入错误，也即是说，由于在相关的操作手册中没有给出正确的操作命令，以至产生了输入错误；或者是实际的输入数据确实是错误的；这样的两种情况都会导致把错误的数据输入到系统中去。因此，正确设计出在发生输入数据错误时系统本身应该采取的应急处理措施，是系统设计的任务之一。在表 2.2 中概括地表示出这三种类型的错误的特性。

表 2.2　各种类型错误的发生情况

	硬件错误(恶化)	软件错误(设计)	输入错误
错误起因	部件的老化故障 环境	设计的复杂性 输入分布	人类的过失
作为时间的函数的错误率的变化情况	初期阶段减少,然后为一常数,在生命周期的后期又增加。	常数(假定程序和输入分布都不变)	初期阶段减少(一个学习曲线)然后是一常数。
这一类型的错误能否完全地从系统中排除掉(理论上/实际)	不可/不可	可[①](?)/不可	不可/不可

根据这一对错误进行分类的方法,存在于数据库中的一个错误的数据元,它或者是一个软件错误,或者它将引起一个错误的输入。如果我们采取这样一个观点:数据库是软件系统的一个部分,则存在于数据库中的一个错误的数据元,也是一个软件错误。但是,如果数据库被认为是独立于软件系统之外的,则存在于该数据库中的一个错误的数据元,将导致一个输入错误的发生。

根据错误发生的持续时间来分类

对于硬件错误和输入错误,我们可以根据它们发生时的持续时间来进一步进行分类。如果一个错误从某一特定的时间开始,往后它总不被打断,则可以将它称为永久的错误。可是,当某个错误发生时,它只使得系统特性产生一个十分短暂的变化,然后系统又可能完全恢复正常状态,这样的错误可以被称为短暂的错误或瞬时错误。此时虽然错误的持续时间是短暂的,但它产生的后果却可以是十分严重的,特别是对于程序往后的进一步执行要受到严重的干扰。由于短暂的错误不可能用系统的方法予以重现,因而使得对它们的诊断和排除成为十分困难的工作。

在硬件中发生的短暂错误,往往很难与软件错误所造成的影响区分开来。例如:一个硬件的缺陷,它产生的影响是将某位二进制数位取反,则这一影响可能导致一个错误数据的存储,或者一条错误指令的执行,由此而引起的征兆就非常难得与软件错误区分开来。

内部错误与外部错误

一个错误的直接后果,有时常常是不能够被直接观察到的,除非只有在诸如一个“内部错误”的影响传播到一个影响到输出的点上时,该错误才能够被从系统的外部观察出来。内部错误与外部错误之间的区分,就是以可观察性作为依据的。如果一个系统采用了

① 关于在软件中的设计错误,在理论上能否从系统中完全排除掉的问题,至今仍有两种截然不同的意见,一种认为可以——如图上所示;另一种则认为人类的过失在设计过程中无法完全避免,所以是不能够完全从系统中排除掉的。

冗余技术,并非每一个内部错误都将必定会导致一个外部错误。内部错误和外部错误的细分,在极大的程度上要依赖于在每一个特殊情况中的、被选择用来查错的查错界面。例如,如果一个查错界面具有分层次的结构,则同一个错误既可以被看作一个内部错误,也可以被看作一个外部错误,它的区分则依据于作为划分标准的层次的级别。而且,深入研究内部错误与外部错误之间的区别,是进一步分析可靠性的基础之一。为了给从系统中排错打下一个基础,就要求对在某些特定的查错界面上查出所有内部错误的起因的能力进行量测;或者,要求在采用冗余部件以后,对于随后的外部错误的预防能力进行量测。如果一个错误在系统内部查不出来,也就是说,查错技巧只能在系统以外来实施的话,那么,该错误就可以引起一个严重的系统错误,也可能就是引起系统崩溃的根源。

根据应用的结果对错误进行分类

关于复杂系统的说明,除了包含有全部的、以经济的观点来解释的系统执行的基本功能以外,往往还要包含对那些有助于增强整个系统的有效性的次要一些的功能的解释。假如这些次要功能失效的话,对于用户来说,结果也许并不会是灾难性的。但是,如果一个错误导致系统基本功能执行的中断,那么这一错误就是关键性的。如果一个错误仅仅只影响到系统的那些次要功能,那么,这个错误就是非关键性的。只有关于系统的说明中能对全部的基本功能和次要功能进行清晰的划分并加以区分的时候,将错误划分成关键性的和非关键性的这一分类方法才是行之有效的。对于可靠的系统的开发来说,这样的区分是十分重要的。而且十分明显,对于关键性错误,设计者们必须要给予足够的重视。在确定性的情况之下,为了纠正一个已经出现的关键性错误而故意中断一个次要功能的执行,即使不是十分必要的,那也是完全可行的。Fragola 和 Spahn 于 1973 年甚至提出比仅仅区分关键性错误和非关键性错误更强的区分法,以便对一个错误的后果亦可进行分类。在他们提出的方法当中,对每一个错误都标以一个数字,以便指示出它们的严重性的程度。

根据开发阶段的分类法

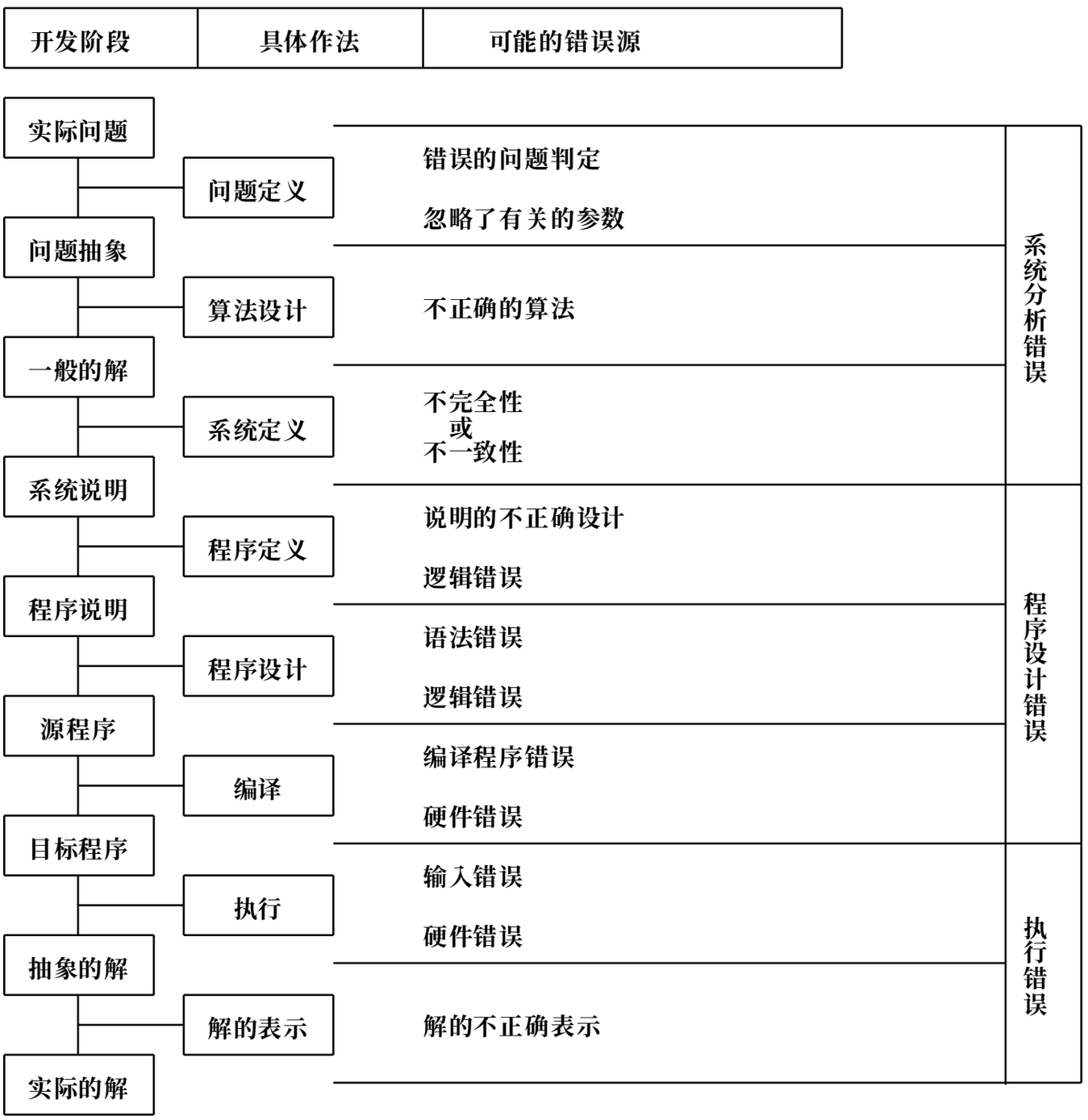
一个软件系统的开发与使用可以分成许多阶段,而其中的每一个阶段又是下面一个阶段的先导。因此,出现于开发过程中的错误,也可以根据开发阶段来予以分类(如表 2.3 所示)。对于一个具体的问题都能得出一个具体的解,换句话说,当在软件开发周期中的每一阶段都是正确的时候,也只有在这时,才可以认为一个给定的结果是正确无误的。因此,对于开发过程中的每一阶段,我们都必须给以足够的重视。只注意于开发的某一个环节而忽视其它的环节,这样做的结果只能对整个系统的可靠性起到相反的作用。

系统分析中的错误

我们周围的大千世界,包罗了万事万物,有许多的问题也并不总是以数学公式的形式出现的。因此,通过一个长期的、反复的抽象过程,以便从实际的问题中提取出对该问题的描述,写出对该问题的抽象化表示,往往是十分必要的。在技术上来讲,这一点对于上面所叙述的开发阶段,也往往是很困难的一个环节。它要求从大量的信息中抽取出与具体问题有关的每一个要素,去掉那些并不涉及到问题本质的次要因素,去掉许多根本与问题无关

的、但又往往会掩盖问题本质的那些因素。为了完成这一极具挑战性的任务，也要求设计者应具有丰富的专业知识和高度的抽象能力。在系统分析期间出现的错误往往总是在整个系统完成(指抽象出来的对具体问题的表述系统)之后才被发现出来的，这是因为，在将具体问题进行抽象处理的整个阶段，不存在什么正规的参照标准可资查考。据 Endres 等人在 1975 年的研究报告称：在分析了 DOS/VS 操作系统以后发现，在该系统中所碰到的全部问题中的半数左右，都是由于对问题的误解所造成的，而不是由于程序设计的原因而引起的。一般说来，程序设计的错误比系统分析的错误更易于辨认出来。

表 2.3　在开发过程中的各阶段及其可能的错误源的例子(Elspas, 1971)



另外，与具体问题表述的抽象化比较，为解决该问题而作的算法设计和对一个确定的程序说明进行定义，通常要容易些。

程序设计的错误

一旦源程序偏离了程序说明,就容易产生程序设计的错误。不论是在逻辑、编码的具体实现过程中还是在代码的转换过程中所出现的全部错误,都可以归于程序设计的错误一类。

下面给出一部分程序设计错误的例子。

(1) 在逻辑的具体实现期间:
程序说明的错误中断;
不完全的逻辑;
特殊情况的忽略;
缺乏对错误处理能力的考虑;
在时间考虑方面的疏忽;等等。
(2) 在编码期间:
语法错误;
初始化的错误;
参数的混淆;
循环计数错误;
对判定结果的不正确处理;
变量的重复或者未定义就加以使用;
变量名的书写错误;
类型和维数的不正确说明;等等。
(3) 在代码的转换期间:
编译程序的错误;
外部符号的错误解析;
库程序名的混淆;等等。

对于程序设计错误的出现频率这样一个问题,存在着不同的分歧意见,这些意见都部分地与对程序员的不平等限制有关,也部分地与在程序设计期间所遭遇到的复杂性的变化程度有关。很明显,程序设计方法和设计人员所选择的程序设计语言一样,它也会强烈地影响到程序设计阶段所产生的错误的数目。

在由 B vehm(1974)所编译的一个实时系统中,从对所查出的软件错误数目的情况分析看来,我们可以注意到这样一种情况:在三个子系统的每一个当中所查出的逻辑错误的数目是与起先花费在系统分析和系统设计阶段的时间成反比的(见表 2.4)。

数据准备过程的错误

实际上,几乎每一个数据准备方法的错误易发性都是很值得考虑的问题。在穿孔卡片机上或者在其它的数据准备系统上面准备数据的时候,人类的过失几乎是不可避免的。

表 2.4　根据错误源对错误的分类(B$\overset{iqo}{V}$ehm, 1974)

错　误　源		硬件诊断程序(%)	系统软件(%)	用户软件(%)
在一个变更之后的意外副作用		5	25	10
设计中的逻辑错误	原先的	5	10	2
	变更之后的	5	15	8
设计与执行之间的冲突		5	30	10
小计:	逻辑错误	20	80	30
	硬件错误	40	—	20
	书写错误	40	20	50
总计:		100	100	100
错误的总数(在 3 年中查出的)		36	108	18
指令条数		4K	10K	10K
分析和设计占全部努力的百分比		59%	37%	54%

Gilb (1973) 详细地分析了一个典型的商业应用系统(记帐系统)中的错误统计资料,并且发现:

几乎占全部输入记录的 3～5% 中存在着错误;
全部错误的 30～40% 出现在记载现金款项的数据区,因此很难进行校验;
仅仅只有一小部分计数错误能经校验码查出。

经由模拟的方法得到的数据或者从数字源所获得的数据,也能够导致错误,并且对于由它们所产生的错误的出错概率还不能低估。在远距离处理系统中,数据的传输也是应引起我们进一步加以注意的错误源(Martin, 1972)。

执行错误

在讨论执行期间出现的错误时,我们总是假定:一个正确的程序,加上正确的输入,就一定能得出正确的最后结果,也就是说,假定在程序的执行期间是不会发生错误的(当然,前提是这个程序是正确的)。这种可能性不但是可能的,而且是应该的,否则,我们将对计算机系统得出的计算结果都要持疑问的态度。

然而,除此之外,在某个程序的执行期间,诸如操作失误等情况的发生,也是不乏机会的。另外,比如错误的磁盘、磁带的使用、不正确的操作系统的初始化、对于在系统中所遇到的困难问题的不正确反应,等等,也都是些典型的例子。

此外还有:人类的过失、机器的错误(即指硬件的缺陷)、电源系统的故障、空调设备的故障、数据媒介设备的永久性损坏(如:磁盘机上磁头的损坏等)、计算机本身的问题、瞬时故障(如:计算机硬件的暂时失灵)等一系列的问题,也都是很恼人的。

在用户软件的执行期间,一定要要求操作系统的服务,因此,操作系统本身的错误也要引起用户任务的故障概率的增加。而且事实上,操作系统本身内部也确实同样存在着大量的错误。正如 Hopkins(1970)所说的:“我们在开发大的系统过程中,面临着一个稀奇古怪的问题。例如,在我们的 OS 360 中,几乎每个版本都有上千个错误存在,而且有理由认为这是一个常数。”

象操作系统一样,诸如溢出一类的算法错误也只有在运行过程中出现,这当然也是应该予以重视的一类问题。

瞬时的硬件故障

瞬时的硬件故障往往总是被误认为是软件错误,所以,对瞬时硬件故障的特别处理就显得十分重要。虽然根据 Ball 和 Hardie 的研究[8]结果可知:在许多技术方面,瞬时硬件故障出现的概率比固定的硬件故障出现的概率要大,但是,几乎没有任何其它的研究报告可资比较。究其原因,可能在于:这一类的故障出现的时间十分短暂,因而,企图指望通过测试程序的执行来清楚地认识它们是十分困难的工作。

瞬时硬件故障的诱因,一方面可能是由于电源系统或者计算机的物理环境中有间歇性的干扰(比如微小的冲击震动等)出现,另一方面可能是由于某硬部件的缓慢而逐渐的崩溃。在 Ball 和 Hardie 于 1969 年对这一类型的故障所进行的广泛模拟研究中,揭示出如下的主要事实:

(1) 控制部件对瞬时硬件故障的反应比算术部件的反应更敏感;
(2) 在一条指令的执行期间出现在控制部件中的故障,被立即查出的概率,其数量级要小于出现在算术部件中被查出的概率的数量级;
(3) 瞬时硬件故障仅仅只有在它们持续的时间大于一个周期时,才是关系重大的。

瞬时硬件故障所引起的后果是产生不正确的数据或者是产生不正确的控制流。由于这些征兆与软件错误的后果十分相似,它们往往会诱使人们作出错误的结论——瞬时硬件故障是一个系统故障的诱因,特别当通过诊断程序的方法检查硬件并且显示出硬件功能完全正常时,更是如此。

在硬件本身并不具备查错逻辑功能(例如内存的奇偶校验功能)的部件中,瞬时硬件故障相比较而言,可能会引起更多的困难。对于许多微型计算机系统,这一点更是特别明显。

虽然,统计在“无法解释的错误”这一类中的一部分错误,必定属于瞬时硬件故障,这样的观点已经为人们所接受,但并不等于说这一类型的错误就与无法解释的系统行为是一码事,特别是在系统软件的开发和委托期间,更不能将它们等同起来。在对硬件经过一段长时间的测试以后,要估计这些瞬时硬件故障出现的概率,应该是可能的。然而,在高可靠性要求的系统开发的过程中,对瞬时硬件故障出现的可能性不应该忽视,因为对于这些高可靠性要求的系统来说,瞬时硬件故障也可以变成系统故障的主要诱因。

数据库错误

如果一个系统要处理大批的数据项,就必须对下面两种情况加以区分:一是这些数据是被作为系统的外部对象来加以处理的;一是这些数据构成系统内部的一个完整的部分。在前一情况中,输入数据和输出数据之间存在着一个明显的界限。在处理的过程中,假如发生一个错误,则重新启动该计算并不会是特别困难的事情,也即是说,重新对该系统进行初始化的工作,是很容易做到的。在处理过程的末尾,还可以对输出数据是否正确进行校验,当然,这些校验工作可以在将输出的数据作为下一步计算的输入数据之前进行。由于应用了对一系列的数据的发生进行存档的技术,就使得对追溯一系列的处理步骤成为可能。

在数据必须作为系统的一个完整部分来进行处理时,情况则要复杂得多。在这种情况中,为了排除一个在系统中出现的错误,就必须从一个没有错误的、并且是可以明确地加以定义的系统状态开始进行处理。然而,这一状态的定义代表一个由存储在内存中的数据的拟连续变化性而引起的、应加以考虑的问题。明显地,自从该系统状态被定义以后,对已经作出的全部记录,都不得不加以复制。在这样的系统中,数据准备可以更好地由对输入的记录与预先存储的数据进行直接的比较来更好地加以校验。但是,由于存储于内存中的数据的连续变化的性质,对于文件中的内容进行的是否正确的测试,或者和数校验,都比在上面讨论的情况中所讨论的校验要更加困难。除此之外,还有一个并行处理的复杂性增加的问题。

因此,对于错的数据元的存储问题,所产生的影响不可低估。假如这一错误的数据元被用来作为下一个记录的输入,则这一记录的结果也必将是错误的。

这些错误的缓慢传递,可能导致对一个数据库的侵蚀,它可能要在经过了许多个月之后才被查出来。到那时,对于重新生成数据所要求的必要数据和记录将可能再也不是可资利用的了。因此,对一个数据库的内容所作的特殊检查就是十分基本的要求了。

总之,概括起来讲,在一个错误出现之后,要重新存放系统本身的、被定义的状态,是更复杂了,它势必要花费更高的代价以排除错误和克服它所产生的影响,并且因此,对于系统所应具有的可靠性要求,必定也要作相应的增加。因而我们应该经常意识到,由一个简单的初始化过程,对完全的系统进行处理,必将会带来巨大的好处。

根据以上讨论的各种分类方法,并结合作者开发软件的实践,现将开发过程中所发生的错误,分类归并,特列出如下:

一、用户要求的变化

简化接口/使系统使用更加方便

要求全新的功能/增加新的功能

对 CPU 的要求

对磁盘的要求

对输入/输出的要求

对内存的要求
对安全性的要求
要求新的硬件/对操作系统的功能上的要求
对其它外设的要求
对容量的要求
对数据库管理以及完整性的要求

二、文档错误

程序限制
操作过程
流程图(或 PAD 图)与编码之间的差别
错误信息
程序应有功能的说明
输出形式的要求
文档不清楚/不完全
test case 文档
操作系统文档

三、逻辑错误

对问题进行抽象时的错误
不正确的逻辑
无效逻辑
本来简单的逻辑给复杂化了
采用了错误的逻辑分枝
不完全的处理
死循环
逻辑或条件不完全的测试
对上、下限的判断错误
下标未经校验
特征值或特殊数据值未经测试
重复步长的不正确判定

四、结构上的错误

编译错误
程序的不正确分段
非法指令
不可解释的停机(包括硬件的瞬时故障)

五、数据处理的错误

向错的磁盘区读/写数据
数据被丢失/不能存储
数据、下标、特征值未加设置
数据、下标、特征值的不正确设置
数据、下标、特征值的不正确初始化
数据、下标、特征值的错误更改
额外项(表、数组)的产生
二进位信息处理错误
错误变址
数组类型转换错误
内部变量的错误定义、设置或使用
数据存/取错误
程序对并不存在的记录进行数据查找
越限
数据链接错误
溢出或对溢出的错误处理
读出错误
读不出数据
错误的分类
错误的覆盖

六、操作错误

测试执行错误
使用错误的主结构
盘用错
带用错
数据准备的错误

七、在要求满足方面的问题

超过规定的运行时间
要求的能力被忽略/对响应要求方面考虑不周
未达到所要求的功能
在规定的时间之内不释放资源

八、计算错误

使用不正确的表达式与习惯表示法
数学模型的问题
不精确的运算结果
算法选择的问题

非期望的运算结果
下标计算不对
混合运算次序不对
向量运算错误
符号习惯表示法的错误运用

九、用户接口的错误

程序对输入数据的不正确解释
程序拒绝接收有效的输入数据
程序无法使用有效的输入数据
本来已加以拒绝的输入数据又为程序所用
不使用读出的输入数据
接受并处理加工非法的输入数据
对合法的输入数据作不正确的处理
操作接口设计不完善
不适当的中断与再启动功能

十、程序/ 程序接口方面的问题

程序要求错误的参数
程序无法使用可用的数据
调用次序紊乱
程序初始化错误
程序之间通过错误的数据区进行通讯
程序的不兼容性

十一、程序/ 系统软件接口方面的问题

操作系统接口错误
程序对系统支持软件的不正确使用
操作系统本身的问题限制程序功能的发挥

十二、输入/ 输出错误

输出信息的丢失
输出时丢失数据项
输出与设计文档不一致
输出形式不正确
重复输出
输出场大小不适当
排错时的输出问题(与设计文档有关)
打印机控制中的错误

行/页计数错误
设计中未定义必要的输出形式
应输出的出错信息被删掉了
标题输出的问题

§ 2.3 软件中的一些重要测度

为了对软件系统进行一些定量的研究,也为了便于对一些同类型的软件进行比较,人们陆续规定了一些软件的测度。下面,择其主要者略述之。

描述一个软件大小,人们常用的是它的语句的行数,也即通常所说的"某个程序有多少代码行。"另外,也有人用一个程序中所含字符个数的多少来衡量一个程序的大小。除此之外,人们也经常用一个程序所占内存空间的多少,来衡量它的大小。

进而,有人将程序中用到的操作符及操作数也作了量化(见 Halstead, 1977):

- 程序中使用的操作符的种类,用 n_1 表示;
- 程序中使用的操作数的种类,用 n_2 表示;
- 程序中出现的操作符的总次数,用 N_1 表示;
- 程序中出现的操作数的总次数,用 N_2 表示。

并以此为基础,规定了:

- 程序所用词汇量的多少,用 n 表示,则:

$$n = n_1 + n_2.$$

- 程序长度的观测值,用 N 表示,则:

$$N = N_1 + N_2.$$

- 程序长度的估算值,用 N 表示,则:

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2.$$

为了使 N 和 N 的值相符,必须将由编译程序检查出来的程序"杂质"(impurity)全部清除干净。

- 程序实现的容量(即指必要的 bit 数),用 V 表示,则:

$$V = N \log_2 n.$$

- 程序的抽象级别被定义为程序的潜在容量与实现容量的比值,如用 L 表示程序的抽象级别,则:

$$L = V^* / V,$$

其中,V^* 表示程序的紧凑形式所占的容量,即:

$$V^* = L \log_2 V.$$

如果有:

$$V^* = V,$$

则:

$$L = 1.$$

通常为节省存储空间,总是使 $V^* \leq V$,于是一般总有关系:

$$L \leq 1.$$

· 程序抽象级别的估算值,用 L 表示,则:

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}.$$

用程序设计语言写的程序的抽象级别,要受到某常数值的限制。根据不同的程序设计语言对程序的抽象级别所施加的限制不同的事实,估计出的语言不变量,用 λ表示,则:

$$\lambda = LV^* = LV^2.$$

一般地,程序设计的难度(指程序设计时的复杂程度)随 V 值的增大而增加,但随 L 值的增大而下降。于是又定义了为制作程序而须付出的努力的测度(指全力以赴的优秀程序员完成任务时的努力)是:

$$E = \frac{V}{L} = \frac{V^*}{L^2} = \frac{(V^*)^3}{(LV^*)^2} = \frac{(V^*)^3}{\lambda^2},$$

E 在规模为 100 人-月的软件研制项目的设计中,与报告的出错数密切相关,相关系数约为 0.982。而且,这一指标也是一般程序员所能达得到的,但 E 中不包括"徒劳的努力"在内。

以 E 为基础,进一步可以估计程序研制所需要的时间 T 为:

$$T = \frac{E}{S},$$

这里,S 是程序员的工作速度,根据经验,一般取 S= 18。

另外,还可以估算完成一段程序所需工作量的人-月数,估算公式如下:

$$MM = \frac{2.1433 \times 10^{-7} (N \ln n)^{1.5}}{\lambda^{0.5}},$$

其中,

MM: 表示工作量,单位为人-月,

N: 2667× I(指用汇编语言写的程序),

1900× I(指用 FORTRAN 写的程序),

I: 指令条数,

λ 其数值因程序设计语言而异,一般,汇编语言: λ= 0.88; FORTRAN 语言: λ= 1.14。λ和 N 的数值均由经验上的统计数据得出。

关于复杂度,主要分为两类:一是用于测量模块复杂度的测度,一是测量模块间的中间连接复杂度的测度。

第一类测度主要有:

- 模块的代码行数: 虽然它有着明显的不足, 但仍被广泛使用, 并构成成本模型的一个基本参数。
- Halstead 的 N、V: 它也有不足之处, 但比代码行却又具有更多意义。
- MaCabe 测度: 它表示控制流图的复杂度, 虽然它的实际意义是明显的, 但似乎用处不大。
- 数据流量测: 可实际用于对数据流图复杂性的测量, 但由于缺乏特殊性质, 常指示错误的程序。
- 凝聚度: 它对一个模块内的不同部分在逻辑上的共同归属进行划分和归类。
- 功能点: 这里的“点”对已知的复杂性根源进行定位(例如: 输入、输出、管理文件等), 并给出一个加权的和数。特别当开发大量十分相类似的程序时(例如, 顾客应用工程), 它是很有用的测度。

第二类测度主要有:

- 连接: 它对不同的模块逻辑上的共同归属进行归类和划分。
- 控制流复杂度: 通常用于保证模块调用图的构造完好(例如: 树型结构)。
- 数据流复杂度: 保证数据调用是构造完好的(特别指外部数据调用)。
- Chapin 的 Q 测度: 它对不同人对模块实现功能的理解程度进行测量。
- Haney 的“概率联结矩阵”Þ: 由于对一个模块的改动和变化, 势必引起其它相关模块的变化。Haney 研究了在这种变化引起的波动的情况下, 系统的稳定性以及对维护策略的影响。

关于软件被测程度的量度, 大多数测度都着重指出给定测试集已覆盖程序控制流图的程度。它看起来是合理的, 因为对于测试条件和数据应起码达到一个规定的覆盖度, 但不应过分相信它。这是因为: 一, 它只是指出了测试集对数据流图和要求的覆盖; 二、虽然由它可推出许多关于测试覆盖的直接标准, 但如何得到这样的测试集仍是不清楚的, 因此它们的性能/价格比是有问题的。

关于软件可靠性方面的量度, 主要有:

- 软件中的初始错误个数;
- 软件经过测试后, 通过查错、改错, 在软件中的剩余错误个数;
- 平均无故障时间;
- 故障间隔的时间长度;
- 故障发生率;
- 软件的可靠度;
- 经预测, 下一次故障的发生时间, 等等。

§ 2.4 软件可靠性的特点及其与硬件可靠性的区别

硬件可靠性理论的建立已经有相当的历史了,在建立软件可靠性理论时,人们也在很大程度上借鉴了硬件可靠性理论的许多成功经验。

软件中的故障源是在设计阶段的人为因素所产生的缺陷,而硬件中的故障源则是因为物理性能的恶化所造成的。为研究软件可靠性而开发出来的概念、理论,可以实际地应用到任何的设计活动中去,其中当然也包括对硬件的设计。但是,软件可靠性的概念毕竟不同于传统的硬件可靠性概念,其具体的体现之一就在于,软件中出现的故障不能归因于硬件所特有的"运行损耗"。一旦软件(设计)上的缺陷需要修改,一般地说,凡在它出现的所有地方都必须进行相应的修复。也即是说,对软件的修复,直接的结果就是要改变软件本身的结构,而且不论局部的或全局的影响都要考虑到。但对于硬件而言,即使是同种类的元器件,发生故障的地方也只是局部性的。从时空概念上来看,软件无论放多长时间,它是什么样还是什么样,无论拷贝多少份,只要进行拷贝的硬件环境不发生故障,它们都是一个样。但硬件一方面存在"自然老化"现象,一方面在批量生产时,因生产过程中众多环节上某些变动,它们的质量是存在一定的离散性的。

通常软件的故障只有当程序所面临的环境,并非是为设计或测试该程序而设的特定环境时,才会出现。这里当然不是指硬件环境和系统软件环境,比如:操作系统等。这些环境不同了,软件根本就无法运行。这里环境是指:软件的使用条件、承担的任务、输入数据的范围等等。在实践当中,大量可能出现的状态和大量可能的输入(甚至是无限的输入空间)使得对于程序要求的完全理解、完美的实现,以及完全的测试,都成为理论上的空话,不可能真正做到。从而,使得软件可靠度基本上只是我们对于软件在设计、生产的过程中,以及在它所预定的环境中具有的能力的置信度的一个测度罢了。

虽然发生于软件系统的故障过程与发生于硬件系统的故障过程不同,但是它们对于系统所造成的结果是相同的,这就是为什么软件可靠性理论能够以类似于硬件可靠性理论的方式来建立的原因之所在。

下面两点是值得注意的:

1. 软件可靠性理论是可靠性理论在软件系统上的特殊应用,它们在许多地方有着相通之处,但也各有其特殊性的一面。我们在研究软件可靠性理论时,一方面要应用许多可靠性理论的知识,但应时刻记住软件可靠性的特殊性。
2. 在从硬件可靠性理论向软件可靠性理论的类推过程中,一定要强调它们不同的地方,切莫太注重它们的共同点,以致产生"一切都可以借鉴硬件可靠性理论"的误解。

概括起来,软件可靠性与硬件可靠性的区别主要有:

- 软件可靠性在意义上区别于硬件可靠性,它主要关心的实际上是软件的设计质量。
- 硬件失效是物理变化的结果,即故障的物理过程;软件失效则是因为设计过程中引入程序中的错误所产生的结果。
- 硬件有损耗,软件没有,不存在关于软件设备的浴盆曲线。
- 关于硬件可靠性已有完整的数学理论加以描述,但对于软件可靠性的数学理论尚待建立。
- 对于硬件可靠性的有关测度,如故障率、MTBF、MTTF 等,对于软件是没有意义的。
- 硬件有风险函数,软件相对的有错误函数。
- 对于估测目的,可靠性增长是一种设计现象,关于许多技术,硬件和软件都可应用。
- 事先估计可靠性、测试和可靠性的逐步增长等技术,对于硬件和软件在意义上是不同的。
- 大多数用于硬件可靠性的冗余技术,并不能用以改进软件可靠性。
- 用于硬件防止发生故障的预防性维护技术,对于软件并不适用,因为软件并不会产生物理失效。
- 硬件的纠错维护是通过修复失效的系统以重新恢复系统功能,而软件只有通过再设计。
- MTTR 对于软件是没有意义的。
- 对数正态分布是用于描述硬件修复时间的,而对软件是不适用的。
- 软件故障不能象硬件那样采用断开失效部件的办法来克服。

图 2.2 示出硬件故障率与软件故障率行为的区别。图 2.3 则示出软件累积故障数与时间的关系。

图 2.2　硬件与软件故障率的行为区别

图 2.3　软件累积故障数与时间的关系

§ 2.5 软件可靠性数据的收集方法

作为软件可靠性估测基础的软件故障数据,在整个软件可靠性工程的研究中,占据重要的地位。目前,软件可靠性研究工作者,普遍感到由于缺乏数据,严重地影响到研究工作的进展。同时,关于数据的收集工作,又存在着许多问题有待解决:

(1) 收集到的数据大多总是不完全的,而遗漏掉的又总是最重要的;

(2) 进行可靠性估测是要花时间的,而进行数据收集则要花更多的时间;

(3) 进行数据收集同样需要方便实用的工具,这方面的工作目前又还十分缺乏;

(4) 由于所使用的数据而产生的可靠性估计误差比由于使用软件可靠性模型而产生的误差要大一个数量级,这说明数据质量改进的重要性。

为了顺利地收集软件故障数据,有几个重要的概念必须先讨论。

错误(error)一般是指软件开发者在开发过程中所犯的人为错误,正是由于这些人为的错误,才使开发出的软件中存在问题,也才使它在其应用环境中运行时表现出故障现象。人们于是又将软件中客观存在的、静态的错误存在形式称为缺陷(fault)。因为缺陷的客观存在,在它们存在其中的软件运行过程中,软件系统的实际行为就必定会偏离事先规定的行为要求,这种现象就称为故障(failure)。因此,软件故障是一个相对于缺陷而言的一种动态的概念,它们只有当软件在其应用环境中运行时才能被观察到。细心的读者将会发现,这里我们把§ 2.2 中的错误与过失合并为一个概念,统称为错误。

基于以上的讨论,我们可以看到,要收集数据,必须要在软件的实际运行过程中去进行,也即在软件的测试阶段或投放以后的维护阶段收集。而收集到的数据,一般就称为软件故障数据。上面已提及,故障可以形式化地定义为软件在其执行期间的表现偏离了事先规定的行为要求。因此,如果规格说明书错了,尽管软件的实现与规格说明书的要求是符合的,但它与用户的要求不吻合,从用户立场上来看,这也是对事先规定行为的偏离,它将直接影响到用户的使用。软件中的缺陷大致可粗略地分为代码中的缺陷、文档中的缺陷和可使用性的缺陷三类。我们可以注意到,这里故障的概念是涵盖了文档中的可使用性方面的缺陷的明显表现的(在运行过程中的表现)。更一般地说,就是故障总是软件代码中的缺陷在运行过程中的明显表现形式。因此,只要用户一有抱怨,就可以说,软件已出现了故障。

另外,在故障数据的收集问题中,还有一个重要的概念,即时间,也必须作一番研究。在对软件可靠性的估测工作中,一般都将时间粗分为两大类:日历时间与执行时间。Musa 坚持认为,执行时间才是唯一有效而实际的量测。但根据我们的实际工作经验,对于软件运行时间的量测,并不一定非执行时间不可。执行时间要由操作系统给出,增加了故障数据收集的工作量和成本。而使用一般的日历时间,在对软件的故障数据收集过程中,以及对软件可靠性估测的过程中,它所表现出的灵活性及优势,要比使用执行时间更大,而且也并不影响到最后的分析结果。只是一般我们在研究过程中,将日历时间中的节假日的时

间去掉了。之所以这样做的原因是显然的,而且从以后的估测工作结果来看,也是必要的。

软件的运行时间这一概念是十分重要的,但往往不容易被真实完整地记录下来,因此不能真正地反映出故障的实际出现情况。因此,我们还是要进一步分析一下各种各样的时间概念。存在许多测量时间的方法,它们包括:

(1) 逝去的时间:对于经常要运行的软件而言,它可能在每一周的工作时间内,都运行同样长的一段时间,如系统操作等。

(2) 开机时间:对于计算机一开始启动就要进入运行状态的软件,如实时系统软件、嵌入式的操作系统软件等,只要系统开始运行,它们就要始终一直不停地运行。

(3) 正规机器时间:如上述(2)中所指出的,由不同供电系统的机器集中共享数据,且应用一个校正因子来计算的开机时间。

(4) 程序加载时间:对于一多道程序设计环境中的批处理程序,它的加载、运行以及删去所花的时间。

(5) 处理机时间:对于上述(2)、(3)、(4)中任一时间内所花费的处理机时间。一般处理机都有时钟。

(6) 会话时间:对于一交互式程序除开闲置时间外的时间,或仅计算其可使用性问题所花费时间。

对于以上各种时间的记录难易程度以及精确程度各不相同。但是,自动记录器,例如系统会计(system journal),是有用处的。有些时间的记录只能靠自动记录,如果由用户填表,绝对不可能记录完全,如处理机时间等。

要保证全部、精确的数据收集是十分困难的,特别要保证出现故障时的报告做到完全而精确则更难。其中,最经常也是最严重的问题是故障时间的丢失。解决的办法,一是靠教育管理,二靠及时的验证。这一点,我们下面还要讨论。

在我们的实际研究工作中,将软件故障数据分为完全数据和不完全数据两大类。下面给出它们的形式定义:

定义 2.1 数据集合$\{y(i) \mid y(0)=0, i=1,2,\cdots,n\}$称为完全数据集合,如果:

$$\forall i(i \in \{1,2,\cdots,n\} \rightarrow y(i)-y(i-1)=1),$$

其中 $y(i)$ 为直到时间 $t(i)$ 时的累积故障数。

定义 2.2 数据集合$\{y(i) \mid y(0)=0, i=1,2,\cdots,p\}$称为不完全数据集合,如果:

$$\forall i(i \in \{1,2,\cdots,p\} \rightarrow y(i)-y(i-1)>1).$$

完全数据的例子见§ 5.2 中,表 5.4 的 NTDS 数据;不完全数据的例子有 D.inc 和 Martini.d, D.inc 是我们在开发软件项目 WPADT 时收集的,其中的时间已将节假日去掉了;而 Martini.d 则没有去掉节假日。D.inc 示于表 2.5, Martini.d 示于表 2.6。显见,完全数据就是故障时间间隔数据,不完全数据就是每个时间间隔内的累积故障数据。

下面我们讨论影响数据收集的因素及解决办法。

如果说,模型的假设是全部软件可靠性分析和估测的理论基础,那么,软件故障数据就是整个软件可靠性分析和估测过程的工作基础。软件可靠性估测模型的假设是否合理,应由软件故障数据来检验;模型是否精确、估测的结果是否令人满意,要靠软件故障数据来验证。没有数据作为坚实后盾的软件可靠性估测模型,是没有说服力的。

表 2.5　D.inc:不完全数据

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
t(i)	0	2	3	7	8	9	10	11	18	21	33	35	37	44	45	47
y(i)	0	4	5	7	8	14	17	28	29	30	31	33	41	46	48	50
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29		
t(i)	48	49	50	51	52	53	55	56	57	63	76	83	91	106		
y(i)	53	56	59	64	67	68	69	71	74	76	78	79	80	81		

表 2.6　Martini.d:不完全数据

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
t(i)	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
y(i)	0	7	8	36	45	60	74	82	98	106	115	120	134	139	142	145	153	167	174	183	196
i	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38			
t(i)	210	220	230	240	250	260	270	280	290	300	310	320	330	340	350	360	370	380			
y(i)	200	214	223	246	257	277	283	286	292	297	301	302	310	317	319	323	324	338			
i	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56			
t(i)	390	400	410	420	430	440	460	470	480	490	500	540	550	560	570	580	590	600			
y(i)	342	345	350	352	356	367	373	378	381	383	384	387	388	393	398	400	407	413			
i	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71						
t(i)	610	620	630	640	650	660	670	680	690	720	730	740	770	800	810						
y(i)	414	417	419	420	429	440	443	448	454	456	457	458	459	460	461						

软件产业是一门典型的知识密集型产业,存在于软件开发过程中的特殊复杂性问题,大多来源于人类脑力劳动的社会化,对它的管理要复杂、困难得多。由于受到许多潜在因素的影响,数据收集工作变得极为困难。这些因素归纳起来大致有:

(1) 对软件进行度量的尺度定义混乱不清。如对时间、故障、错误类型、模型结构等的定义,就相当含糊,缺乏统一的标准。这样就使得在进行软件故障数据的收集时,目标不明确,严重时甚至无从下手。

(2) 对软件产品的管理问题。软件产品的随意复制,可使它们在不同的系统上运行;同一产品的不同版本,又可以不受限制地同时被使用。于是,其后果就可能使收集的软件故障数据含混不清。

(3) 不完全的排错及诊断,使收集的数据中含有不少的虚假成分,它们不能正确反映软件的真实状况。

(4) 收集技术本身需要许多方便、实用的工具,以及结构精良、定义严谨的数据库。但是,目前由于这些工具及数据库的制作、设计及应用并未受到应有的重视,以至严重妨碍

了对软件故障数据的收集。特别是在“自动错误数据收集”的问题,因为有关错误信息与自动诊断难以定义,存在着许多有待解决的难题。

(5) 心理因素的障碍。在软件开发过程中,自始至终存在着进度压力。争取将软件早日投放市场的激烈竞争,使进度成为首先要考虑的问题。不但高层管理人员、项目主持人要考虑,就是具体承担开发任务的人员,也要时时刻刻考虑到如何加快进度。这时,如果缺乏严格而科学的管理,数据收集的任务就会被当作令人厌烦的“额外负担”而得不到应有的重视,从而无法完成。

为了克服上述不利因素,可以采取如下几项措施:

第一,要解决认识上的问题,使高层管理人员、项目主持人及开发人员相信,他们开发的项目会因所收集的数据而受益,从而最大限度地发挥他们积极主动的、充满活力的“参与意识”。要向他们解释清楚收集数据的原因及要收集哪些数据。在讨论数据收集方案时要请他们参加,听取他们的意见、建议和要求。特别在制定计划时,要充分考虑到他们的困难和要求。

第二,对所要收集的数据,要尽量建立统一的定义标准。虽然在软件工程标准化规范中,偶尔也提及了数据、收集数据的表格等等,但离实际工作的要求仍相距甚远。故应通过广泛的实践活动,最终产生出一个为大家都接受的标准规范。

第三,要保证使收集的数据是“干净”的。这就要求判定每次出现的故障是否是“第一次出现的”,因为对于那些因不完全排错或暂时尚未进行排错的软件,重复发生同样的故障是经常的。另外,特别在多道程序中,要注意区别哪些故障是哪一个程序中的故障。同样,对于软件产品的版本管理,也是在进行数据收集时要加以注意的。不要将前一版本的故障计入后一版本中去,反之亦然。

第四,应使软件故障数据的收集工作尽可能地方便有效,要尽可能地减轻从事软件故障收集工作人员的劳动强度。例如:给他们提供简单明晰的表格、方便实用的工具、便于操作管理的数据库等。要尽量降低数据收集的频度,尽量减少数据收集活动本身所涉及的人数。还必须要求他们及时收集、记录数据,以免为追记过去的信息而浪费时间。要建立核对数据记录的制度。出现问题时,应及时召开有关人员参加的小规模会议,消除误解,修改计划,解决问题。

第五,在收集到一定数量的数据以后,就可以进行数据分析,并对软件可靠性指标进行估测。对数据所反映出来的问题进行分析,如某程序应延长测试时间、某人突然出错率上升、软件的某处部件可能成为不能按时完成的关键部位、某些部门(或小组)在一定时间内需要加班等等。凡此种种信息,都应及时地反馈到有关人员,以便及时采取相应的措施。

最后,为保证上述工作有条不紊地顺利开展,严格的科学管理手段是不可缺少的。否则,软件故障的收集工作就不可能圆满完成。负责收集软件故障数据的人员,必须具有细心、严谨、坚韧不拔的工作作风。当然,在收集软件故障数据时,如果成本、进度方面的矛盾变得十分尖锐,就应该采取灵活的折衷方案。

第三章　可靠性分析的数学基础

这一章的内容是阅读后面几章的准备知识,在内容的选择上以实用性为主,因此,也就不能完全顾及严密性、完备性了。好在讲授概率论和数理统计的书很多,读者如果要进行这方面的深入研究,是有很大选择余地的。

这一章要介绍的内容,在对软件可靠性的分析和模型的应用上都会用到。它们包括:随机变量及其分布、常用的随机过程简介、常用的参数估计方法三个主要方面。读者在读完全书之后,会体会到这里介绍的切实是在研究软件可靠性时常常用到的知识。

§ 3.1　随机变量及其分布

用 T 表示一部件的寿命,由于各种随机因素的作用,T 往往是一随机变量。设 T 为非负连续,即 T 可在区间[0, ∞)中取值。用 f(x)表示 T 的概率密度函数,则 T 的分布函数,又称寿命分布,为:

$$F(t) = P(T \leq t) = \int_0^t f(x)dx.$$

部件的可靠性,即寿命大于给定的 t 的概率为:

$$R(t) = P(T > t) = \int_t^{\infty} f(x)dx = 1 - F(t).$$

可见,可靠性函数 R(t)满足:

$$R(0) = 1,$$

$$R(\infty) = \lim_{t\to\infty} R(t) = 0.$$

在可靠性理论研究中,常定义寿命分布的风险率(又称故障率),即,

$$r(t) = \frac{f(t)}{R(t)}.$$

它表示在时刻 t 的故障发生率。实际上,$r(t) \cdot \Delta t$ 可以用于近似表示在寿命达到 t 时,在未来时刻 $t + \Delta t$ 时,出现故障的概率。

为了方便读者,在表 3.1 中,我们给出函数 f(t), F(t), R(t)及 r(t)之间的关系。

3.1.1　常用离散型分布

若 T 仅在一非连续空间取值,即 T 可取值为 $t_1, t_2, \cdots$,且 $0 \leq t_1 < t_2 < \cdots$,定义 T 的概

表 3.1　f(t), F(t), R(t), r(t)之间的关系

	f(t)	F(t)	R(t)	r(t)
f(t)=		$F'(t)$	$-R'(t)$	$r(t)e^{-\int_0^t r(x)dx}$
F(t)=	$\int_0^t f(x)dx$		$1-R(t)$	$1-e^{-\int_0^t r(x)dx}$
R(t)=	$\int_t^\infty f(x)dx$	$1-F(t)$		$e^{-\int_0^t r(x)dx}$
r(t)=	$f(t)\Big/\int_t^\infty f(x)dx$	$F'(t)\Big/[1-F(t)]$	$-R'(t)/R(t)$	

率分布为:

$$p(t_i) = P(T = t_i), \quad i = 1, 2, \cdots,$$

即 T 的分布函数是:

$$R(t) = P(T \geq t) = \sum_{i:\, t_i \leq t} p(t_i).$$

离散型分布为数理统计理论的基础,下面我们简单介绍几个在软件可靠性研究中常用的离散型分布。

1. 两点分布与二项分布

在许多问题中,我们感兴趣的是某事件 A 发生与否,并称 A 的发生为成功,A 的不发生为失败。这种只有两种结果的试验,称为贝努里试验,记

$$X = \begin{cases} 1, & \text{若事件 A 发生;} \\ 0, & \text{若事件 A 不发生。} \end{cases}$$

我们称 X 服从两点分布,并记

$$p = P(X = 0),$$

$$q = P(X = 1) = 1 - p.$$

两点分布是各种离散型分布的基础,最简单的推广是二项分布。

若随机变量 X 具有:

$$P(X = k) = \binom{n}{k} p^k q^{n-k}, \quad k = 1, 2, \cdots, n,$$

则称 X 服从二项分布。记作 $X \sim B(n, p)$。

易证,若 X_i 服从两点分布,则随机变量: $Y = \sum_{i=1}^{n} X_i$ 服从二项分布。二项分布的数学期望为 np,方差为 npq。

二项分布在概率统计中占有重要的位置。在许多实际问题中出现二项分布,且需计算其数值。关于二项分布的计算,有现成的表可供使用。

2．泊松分布

若随机变量可取一切非负整数值，且有：

$$P(X=k)=\frac{\lambda^k}{k!}e^{-\lambda},\quad k=0,1,\cdots,$$

其中，$\lambda>0$，则称 X 服从泊松分布。记作 $X\sim P(\lambda)$。

易证，泊松分布的参数 λ 就是它的期望值，即：

$$E(X)=\lambda,\qquad Var(X)=\lambda.$$

今后我们将会看到，泊松分布在软件可靠性研究中，有着十分广泛的应用。

3．几何分布

若 X 具有：

$$P(X=k)=pq^{k-1},$$

其中，$p>0,q>0,p+q=1$，则称 X 服从几何分布。记作 $X\sim G(p)$。显然

$$E(X)=\frac{1}{p},\qquad Var(X)=\frac{q}{p^2}.$$

几何分布是离散型分布中具有“无记忆性”的分布，即：

$$P(X>n+m\mid X>n)=P(X>m),\quad m>0,n>0.$$

3.1.2　常用连续型分布

这里我们介绍指数分布、Weibull 分布、极值分布，Γ-分布、正态分布及其变化。

1．指数分布

指数分布函数为可靠性研究中最常见的分布，它的概率密度为：

$$f(t)=\lambda e^{-\lambda t},\quad t\geq 0,$$

它相应的分布函数为：

$$F(t)=1-e^{-\lambda t},\quad t\geq 0.$$

可靠性函数为：

$$R(t)=1-F(t)=e^{-\lambda t},\quad t\geq 0.$$

指数分布的最大特点是它的故障率函数为一常数，即：

$$r(t)=f(t)/R(t)=\lambda e^{-\lambda t}/e^{-\lambda t}=\lambda,\quad t\geq 0.$$

若随机变量 X 具有上述概率密度函数，则称 X 服从指数分布。易证：

$$E(X)=\frac{1}{\lambda},\qquad Var(X)=\frac{1}{\lambda^2}.$$

指数分布图形示于图 3.1 中。

2．Weibull 分布

Weibull 分布是在 1939 年由瑞典科学家 Waloddi Weibull 首先提出的，目前已成为

$f(t)=\lambda e^{-\lambda t},\quad t>0.\quad E(\xi)=1/\lambda\quad R(t)=e^{-\lambda t}.\quad V(\xi)=1/\lambda^2.\quad r(t)=\lambda$

图 3.1　指数分布图

可靠性研究中十分常见的寿命分布,它有效地描述了许多产品的可靠性。

Weibull 分布函数主要由它的故障率函数特征化表示,

$$r(t)=\lambda\beta(\lambda t)^{\beta-1},$$

其密度函数为

$$f(t)=\lambda\beta(\lambda t)^{\beta-1}e^{-(\lambda t)^{\beta}}.$$

它的可靠性函数表达式为:

$$R(t)=e^{-(\lambda t)^{\beta}}.$$

上面所述的 λ是标度参数, β为形状参数。可证, 它的 r 阶矩是:

$$E(X^r)=\lambda^{-r}\Gamma\left(1+\frac{r}{\beta}\right),$$

其中,

$$\Gamma(k)=\int_0^{\infty}u^{k-1}e^{-u}du$$

为常见的 Γ-函数。

Weibull 分布具有单调故障率, 若 β> 1, 则故障率递增; 若 0< β< 1, 则故障率递减。所以, Weibull 分布较指数分布更为灵活。图 3.2 示出 Weibull 分布的 f(t), R(t), r(t) 对 t 的图形。

$$f(t)=\frac{\beta}{\alpha}t^{\beta-1}\exp\left(-\frac{t^{\beta}}{\alpha}\right),\quad t>0.\qquad E(\xi)=\alpha^{1/\beta}\Gamma\left(\frac{1}{\beta}+1\right).\qquad R(t)=\exp\left(-\frac{t^{\beta}}{\alpha}\right).$$

$$V(\xi)=\alpha^{2/\beta}\left[\Gamma\left(\frac{2}{\beta}+1\right)-\Gamma^2\left(\frac{1}{\beta}+1\right)\right].\qquad r(t)=\frac{\beta}{\alpha}t^{\beta-1}.$$

图 3.2　Weibull 分布图

3. 极值分布

极值分布的可靠性函数为:

$$R(t) = \exp - \exp \frac{t - \mu}{b} .$$

相应的密度函数是:

$$f(t) = b^{-1} \exp \frac{t - \mu}{b} - \exp \frac{t - \mu}{b} .$$

极值分布与 Weibull 分布有着密切的关系, 若 X 服从 Weibull 分布, 则 T = lnX 一定服从极值分布。

有时在分析可靠性数据时, 分析对数数据更容易, 故极值数据在可靠性中有一定作用。

4. Γ分布

Γ-分布的密度函数为:

$$f(t) = \frac{\lambda(\lambda t)^{k-1} \cdot e^{-\lambda t}}{\Gamma(k)}, \quad t > 0,$$

其中, λ为标参, k 为形参, 它们的取值范围均为$(0, \infty)$。当 k = 1 时, Γ-分布变为指数分布, 因而可以认为, 指数分布为 Γ-分布的一种特殊情况。

可以证明, 如 k > 1, , Γ-分布具有递增的故障率函数; 如 k < 1, Γ-分布的故障率函数为递减函数。

若随机变量 X 具有上述密度函数, 则称 X 服从参数为(λ, k)的 Γ-分布, 记作 $X \sim \Gamma(\lambda, k)$。易证,

$$E(X) = \frac{k}{\lambda}, \qquad Var(X) = \frac{k}{\lambda^2}.$$

图 3.3 示出 Γ-分布的 f(t), R(t), r(t)对 t 的图形。

$$f(t) = \frac{t^{\beta - 1}}{\Gamma(\beta)\lambda^{\beta}} e^{-t/\lambda}, \quad t > 0. \quad E(\xi) = \lambda\beta$$

$$R(t) = \int_t^{\infty} f(x)dx. \qquad V(\xi) = \lambda^2\beta. \qquad r(t) = \frac{f(t)}{R(t)}.$$

图 3.3 Γ-分布图

5. 正态分布及其变化

当随机变量 X 具有密度函数:

$$f(t)=\frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{(t-\mu)^2}{2\sigma^2}},\quad \sigma>0$$

时,我们称 X 服从正态分布,记作 $X\sim N(\mu,\sigma^2)$。

正态分布常以一些其它的形式,出现在软件可靠性的研究中。若 $X\sim N(\mu,\sigma^2)$,则称 $Y=e^X$ 服从对数正态分布,易证 Y 的密度函数为:

$$f(y)=\frac{1}{\sqrt{2\pi}\sigma y}e^{-\frac{1}{2\sigma^2}(\ln y-\mu)^2},\quad y>0,\quad \sigma>0,$$

且有,

$$E(Y)=e^{\mu+\frac{1}{2}\sigma^2},\qquad Var(Y)=e^{2\mu+\sigma^2}\cdot e^{\sigma^2}-1.$$

容易证明,存在 t_0,有 $t_0\in(0,\infty)$,使得对数正态分布的失效率函数在 $(0,t_0)$ 内单调递增,在 (t_0,∞) 内单调递减。

若非负随机变量 X 具有密度函数:

$$f(t)=\frac{1}{a\sigma\sqrt{2\pi}}e^{-\frac{1}{2\sigma^2}(t-\mu)^2},\quad t\geq 0,\quad \sigma>0,\ -\infty<\mu<\infty,$$

则称 X 服从截尾正态分布,其中 $a>0$ 是常数,它保证:

$$\int_0^{\infty} f(t)dt=1.$$

可算出:

$$E(x)=\mu+\frac{\sigma}{a}\phi\frac{\mu}{\sigma},$$

$$Var(x)=\sigma^2\left[1-\frac{\mu}{a\sigma}\phi\frac{\mu}{\sigma}-\frac{1}{a^2}\phi^2\frac{\mu}{\sigma}\right],$$

其中,

$$\phi(x)=\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}.$$

截尾正态分布的失效率函数 r(t) 单调递增。图 3.4 示出正态分布的 f(t), R(t), r(t)

$f(t)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left[-\frac{(t-\mu)^2}{2\sigma^2}\right]$. $E(\xi)=\mu$ $R(t)=\int_t^{\infty} f(x)dx$. $V(\xi)=\sigma^2$. $r(t)=\frac{f(t)}{R(t)}$.

图 3.4 正态分布图

对 t的图形。

§ 3.2 常用随机过程简介

对于某时间 t, 有随机变量 X(t)与之对应, 则称{X(t), t≥0}为一随机过程。对于任意的 $t_0 < t_1 < \cdots < t_n$, 随机变量:

$$X(t_0), X(t_1) - X(t_0), X(t_2) - X(t_1), \cdots, X(t_n) - X(t_{n-1})$$

若相互独立, 则称{X(t), t≥0}为独立增量过程。

本节我们将集中介绍几个常用的随机过程, 它们包括: 泊松过程、非齐次泊松过程、更新过程以及马尔可夫过程。

1. 泊松过程

如{X(t), t≥0}为独立增量过程, 且对于 t> s, 增量 X(t)- X(s)服从参数为 λ(t- s)的泊松分布, 且 X(0)= 0, 则称 X(t)是具有参数 λ的泊松过程。

泊松过程为最简单的计数过程, 计数过程在可靠性理论中的应用极广。如果 N(t)表示 在区间[0, t]中的累积故障数, m(t)表示 N(t)的数学期望, 那么如何描述计数过程{N(t), t≥0}, 以及如何估计 m(t), 是常被讨论的问题。

2. 非齐次泊松过程

若计数过程{N(t), t≥0}满足以下条件:

i) {N(t), t≥0}具有不相关增量,

ii) P(N(t+ h)- N(t)≥2)= o(h),

iii) P(N(t+ h)- N(t)≥1)= λ(t)·h+ o(h),

iv) N(0)= 0.

则称, {N(t), t≥0}为具有强度 λ(t)的非齐次泊松过程。函数:

$$m(t) = \int_0^t \lambda(x)\,dx$$

称为非齐次泊松过程的均值函数。显见, 如果 λ(t)= c, c 为一常数, 那么该非齐次泊松过程就变为一般的泊松过程。

3. 更新过程

设 $X_1, X_2, \cdots$是独立同分布的非负随机变量序列, 它们的分布函数为 F(t), 密度为 f(t), 均值为 μ 且满足 P(Xi= 0)< 1, 令:

$$S_0 = 0,$$

$$S_n = \sum_{i=1}^{n} X_i, \quad n = 1, 2, \cdots,$$

则计数过程$\{N(t)=\sup\{n: S_n\le t\}, t\ge 0\}$称为由随机序列 $X_1, X_2, \cdots$所产生的更新过程。易证:

$$P(N(t)=k)=F^{(k)}(t)-F^{(k+1)}(t),$$

这里,$F^{(k)}(t)$为 $F(t)$的 k 重卷积,定义为:

$$F^{(k)}(t)=\int_0^t F^{(k-1)}(t-s)\,dF(s), \qquad k>1;$$

$$F^{(1)}(t)=F(t).$$

$N(t)$的平均值 $M(t)$称为更新函数,它可表示为:

$$M(t)=E[N(t)]=\sum_{k=1}^{\infty} kP(N(t)=k)=\sum_{k=1}^{\infty} F^{(k)}(t).$$

可证,$M(t)$满足:

$$M(t)=F(t)+\int_0^t M(t-s)\,dF(s).$$

这一方程称为更新方程。

如果 $F(t)$的密度函数 $f(t)$存在,则 $M(t)$可微,且有:

$$m(t)=\frac{d}{dt}M(t)=\sum_{k=1}^{\infty} f^{(k)}(t).$$

$m(t)$称为更新密度,并且有 $m(t)$满足:

$$m(t)=f(t)+\int_0^t m(t-s)\,dF(s),$$

亦称为更新方程。

关于更新方程,存在多种解法。另外,更新函数也满足一些极限定理,例如:

$$\lim_{t\to\infty}\frac{M(t)}{t}=\frac{1}{\mu}.$$

4. 马尔可夫过程

设$\{X(t), t\ge 0\}$为一组非负、且只取整数值的随机变量。若对于任意 $k\ge 1$, $t_1<t_2<\cdots t_{k+1}$,以及非负整数: $i_1, i_2, \cdots i_{k+1}$,有:

$$\begin{aligned}P(X(t_{k+1})=i_{k+1}\mid X(t_1)=i_1,\cdots,X(t_k)=i_k)\\ =P(X(t_{k+1})=i_{k+1}\mid X(t_k)=i_k),\end{aligned}$$

则称$\{X(t), t\ge 0\}$为一马尔可夫链。当 t 仅取整数值时,$\{X(t), t\ge 0\}$为离散时间的马尔可夫链。当 t 在非负轴上取值时,$\{X(t), t\ge 0\}$为连续时间的马尔可夫链,或马尔可夫过程。

概要地讲,一个马尔可夫过程的“将来”只是通过“现在”与“过去”发生联系,一旦知道“现在”,“将来”和“过去”就是无关的了。

马尔可夫过程可以记为$\{X(t), t\in T\}$,它所可能取值的全体,称为过程的状态空间。其中每一元素,即$\{X(t), t\in T\}$所可能取的值,称为状态。$X(t)=x$,常指在时刻 t,过程(或系统)处在状态 x 中,一般研究的状态空间是$(-\infty, +\infty)$,或它的某个子集,如$\{0, 1, 2, \cdots\}$,或$[0, +\infty)$等。

对于马尔可夫过程$\{X(t), t\in T\}$,最重要的一个概念是转移概率 $P(S, x; t, A)$,其中,

$S, t \in T$, 且 $S < t$, A 是状态空间的子集, 它表示在 $X(S) = x$ 的条件下, $X(t) \in A$ 的条件概率, 即:

$$P(S, x; t, A) = P\{X(t) \in A \mid X(S) = x\}.$$

特别当 $A = (-\infty, y)$ 时, 有:

$$\begin{aligned} &P(S, x; t, (-\infty, y)) \\ &\quad = P\{X(t) \in (-\infty, y) \mid X(S) = x\} \\ &\quad = P\{X(t) < y \mid X(S) = x\}. \end{aligned}$$

这时为方便起见, 也常把 $P(S, x; t, (-\infty, y))$ 简记为 $P(S, x; t, y)$。而且, 更常用的是 $F(S, x; t, y)$, 它称为转移概率分布。

§ 3.3 常用参数估计方法

在讨论参数估计问题时, 分布 $F(x; \theta)$ 中所含未知参数 θ 的可能的取值范围 Θ 称为参数空间, 这里允许 θ 为向量。

用来估计某个未知参数 θ 的统计量, 称为估计量。这样的估计, 也称为点估计。

关于估计量的一些性质、概念, 如无偏性、最小方差性、一致性等, 请读者参阅专业的统计书籍, 这里我们仅对常用的一些方法进行介绍。

3.3.1 最大似然法

最常用的经典统计方法为最大似然估计方法。设母体 X 具有密度函数 $f(x; \theta)$, $\theta \in \Theta$, 且 $x = (x_1, x_2, \cdots, x_n)$ 是来自 X 的一个给定的子样, 则似然函数:

$$L(\theta, x) = \prod_{i=1}^{n} f(x_i; \theta)$$

是参数 θ 的函数。假如 $\hat{\theta}(x)$ 存在, 则有:

$$L(\hat{\theta}(x); x) = \sup_{\theta \in \Theta} L(\theta, x).$$

我们称 $\hat{\theta}(x)$ 为参数 θ 的最大似然估计。

若密度函数 $f(x; \theta)$ 关于 θ 是可微的, 则 θ 满足下列似然方程:

$$\frac{\partial}{\partial \theta} \ln L(\theta, x) \Big|_{\theta = \theta_i} = 0, \quad i = 1, 2, \cdots, k.$$

当母体 X 具有离散分布时, 用分布列 $p(x_i; \theta)$ 代替上述密度函数, 则这里的定义及计算方法仍然适用。

最大似然估计具有许多特殊性质。如果 $g(\theta)$ 是未知参数 θ 的连续函数, $\hat{\theta}$ 是 θ 的最大似然估计, 则 $g(\hat{\theta})$ 一定是 $g(\theta)$ 的最大似然估计。当子样大小 $n \to \infty$ 时, $\hat{\theta}$ 依概率收敛到 θ, 且渐近地服从正态分布 $N\left(\theta, \frac{1}{nI(\theta)}\right)$, 其中 $I(\theta)$ 是 X 的信息函数, 定义为

$$I(\theta) = E_\theta \left(\frac{\partial}{\partial \theta} \log f \right)^2.$$

3.3.2 最小二乘法

设经过某一变换,寿命分布可以写成与未知参数的线性关系,即:

$$g(R(t)) = \alpha_1 l_1(t) + \cdots + \alpha_m l_m(t) \qquad (3.3.1)$$

其中, $\alpha_1, \cdots, \alpha_m$ 为未知参数, $l_1, \cdots, l_m$ 为已知的函数的线性不相关集合,于是参数 $\alpha_1, \cdots, \alpha_m$ 可通过求下列的最小值的方法进行估计:

$$Q = \sum_{i=1}^{k} [Y_i - \alpha_1 l_1(t_i) - \cdots - \alpha_m l_m(t_i)]^2.$$

很多寿命分布都可经过变换写成(3.3.1)的形式。如:对于 Weibull 分布,我们有:

$$\log[-\log R(t)] = \beta \log \lambda + \beta \log t$$
$$= \alpha + \alpha \log t,$$

其中, $\alpha = \beta \log \lambda$ $\alpha = \beta$.

设随机变量 Y 由 p 个参数($\beta_1, \cdots, \beta_p$)及 p 个变量 $X_1, \cdots, X_p$ 组成,且

$$E(Y) = \sum_{i=1}^{p} \beta X_i,$$

由数据 $Y_1, \cdots, Y_n$ 可对参数($\beta_1, \cdots, \beta_p$)进行估计。对($\beta_1, \cdots, \beta_p$)的最小二乘估计,就是寻找($\beta_1, \cdots, \beta_p$),使得:

$$\sum_{i=1}^{n} (Y_i - EY_i)^2$$

最小。对它求导,并令结果为零,即可得:

$$\beta = (X'X)^{-1}X'Y,$$

其中, $\beta = (\beta_1, \cdots, \beta_p)$. 记 $\beta = (\beta_1, \cdots, \beta_p)$,则显然可证,若 $E(Y) = X\beta$, $E(Y - X\beta)(Y - X\beta)' = \sigma^2 I$,(即 $Var(Y_i) = \sigma^2$, $Cov(Y_i, Y_j) = 0$, $i \neq j$),那么, β 为 β 的一致最小二乘无偏估计。

3.3.3 贝叶斯估计

设母体的密度函数为 $f(x|\theta)$,假如将未知参数 θ 看作是定义在参数空间 Θ 上的一个随机变量,并设其分布函数为 $H(\theta)$,则称 $H(\theta)$ 为参数 θ 在 Θ 上的先验分布。当 θ 是 Θ 上的连续型随机变量时, θ 的密度函数 $h(\theta)$ 称为先验密度。若按先验密度 $h(\theta)$ 在 Θ 上产生一个随机观察值 θ,再从母体 $f(x|\theta)$ 中随机地抽取一个子样 $X_1, \cdots, X_n$,于是子样 $(X_1, \cdots X_n)$ 和参数 θ 的联合密度是:

$$f(x_1, \cdots, x_n, \theta) = f(x_1|\theta) \cdot f(x_2|\theta) \cdot \cdots \cdot f(x_n|\theta) \cdot h(\theta).$$

子样($X_1, \cdots, X_n$)的边际密度(或称无条件密度)为:

$$f_n(x_1, \cdots, x_n) = \int_{\Theta} f(x_1, \cdots, x_n, \theta) d\theta$$

则在给定子样($X_1, \cdots, X_n$)的条件下,参数 θ 的条件密度是:

$$h(\theta | x_1, \cdots, x_n) = \frac{f(x_1, \cdots, x_n, \theta)}{f(x_1, \cdots, x_n)}.$$

这就是著名的贝叶斯定理,这个条件密度称为 θ 的后验密度。

设未知参数 θ的估计量 $\hat{\theta}=\hat{\theta}(X)$,用 $\hat{\theta}$去估计 θ所引起的“损失”可用 $\hat{\theta}$与 θ的函数 $L(\theta,\hat{\theta})$表示,$L(\theta,\hat{\theta})$称为损失函数,它一般根据实际情况而定。损失函数 $L(\theta,\hat{\theta})$关于后验密度 $h(\theta|x_1,x_2,\cdots,x_n)$的数学期望:

$$R_h(\hat{\theta}|x_1,x_2,\cdots,x_n)=\int_{\Theta}L(\theta,\hat{\theta})h(\theta|x_1,x_2,\cdots,x_n)\,d\theta$$

称为估计量 θ的后验风险,使得后验风险 $R_h(\hat{\theta}|x_1,x_2,\cdots,x_n)$最小的估计量 $\hat{\theta}$称为 θ的贝叶斯估计。

3.3.4 置信区间与置信水平

由数理统计方法所得的结论往往带有一定程度的不确定性,因此,人们常用置信水平来表示某一结论为“真”的概率。

设对于某一参数 θ,我们通过抽取子样本对它推断出相应的统计量。若抽取了多个样本,则可得到频率函数 f(t),由此可知 t 落在任意两个 t_1,t_2 之间的概率。如果有

$$P(t_1\leq\theta\leq t_2)=r,$$

其中 r 为一特定的概率,则称 t_1 与 t_2 之间(包括 t_1 与 t_2 在内)的所有值的集合为参数 θ的置信区间,记为$[t_1,t_2]$,t_1 与 t_2 称为置信限,r 就是置信水平。

在实际问题中,往往对于事先给定的置信水平 r,估计出置信区间$[t_1,t_2]$。

第四章　软件可靠性模型概述

从本书的§ 1.3 可以看到,软件可靠性模型发展至今已有 20 多年的历史了,各种各样的模型不下百余种。对于这些众多的模型,能不能对它们加以分类,它们各有些什么特点,这就是在这一章里要予以回答的问题。对于模型中所采用的假设,哪些是合理的,哪些是合乎实际的,哪些又有什么样的局限性,在这一章里我们也将进行讨论。最后在这一章里我们还要介绍 Musa 的执行时间概念。

§ 4.1　模 型 的 特 点

软件可靠性是许多因素的函数。首先,它依赖于软件在开发过程中所使用的软件开发方法。如果运用好的软件开发方法,软件中将会含有较少的错误,因此它就更可靠。其次它与验证方法有关,如果有一个完全的测试策略的集合被用来验证一个软件系统,那么这个软件系统将更可靠;如果只应用有限的测试方法对它进行测试,那它一定表现出有限的可靠性。而且,软件可靠性还与所使用的程序设计语言有关,与运行的环境条件有关,与参加开发的软件工程人员有关等等。因此,一个好的可靠性模型,似乎应尽可能多地将这众多因素包括进去,然而实际上并非如此,并且也不可能做到。

作为软件可靠性模型必须要考虑到下面这些软件错误的特性:

- 它们应与每个模型的结构和使用频率无关;
- 软件不存在物理的性能退化过程;
- 软件错误通常是相互有关的,它们一般并不会独立出现,总存在某种逻辑上的关系;
- 在改错的过程中往往会引入新的错误;
- 软件可靠性与人类因素关系密切,与测试策略直接有关。

目前,许多模型为了理论上处理容易,作了不少假设,但其中有许多与实际有出入,因此在实际使用软件可靠性模型进行估测之前,一定要仔细地选择那些与开发实际相符合的或接近的模型,否则会极大地影响到估测精度。关于模型的假设及其局限性,我们在§ 4.3 中再作详细讨论。

软件可靠性模型的特点归纳起来大致有下面这些:

(1) 与使用的程序设计语言无关。具体选用什么程序设计语言来写软件,用以表达算法,与软件可靠性模型的应用之间不应该有什么关系。对于用不同的程序设计语言根据同一个规格说明书写出来的软件,同一个软件可靠性模型应给出同样的估测结果。

(2) 与具体用到的软件开发方法无关。作这个假定很容易也很简单。虽然一个用自顶向下、结构化程序设计的方法开发出来的软件,比一个用非结构化程序设计方法开发出来的软件要更可靠,但要描述这样的影响是很困难的,因为软件开发是一个十分复杂的过程,涉及到许多的人类因素,从而使得对软件的质量难以进行预测。为了保证预测的精度,不妨假设待估测的软件系统是用最坏的软件开发方法开发出来的。

(3) 测试方法的选择问题。原则上,经过彻底的测试可以获得完全可靠的软件。但是,实际上是无法做到的,人们不得不采用有限的测试。于是,目标就是用最少的测试以求最大限度的软件可靠性。这可以用例如边界值测试法、分类测试法、路径测试法等方法来达到。几乎所有的软件可靠性模型都假定测试环境就是将来软件的运行环境,这也就限制了对于要求高可靠性估计情况下的这些模型的可用性。

(4) 改错过程。大多早期的软件可靠性模型假定改错是完全的,但实际上,在改正老的错误时往往会引入新的错误。

(5) 模型要表述的内容。一般,模型应该指示出软件已被测试到什么程度了。详细地说,模型应该指出测试的输入是否已足够地覆盖了输入域,测试的条件和数据是否已准确地模拟了操作环境、是否已足以查出那些相类似的错误等。软件可靠性模型假定测试的条件和数据与操作环境有着同样的分布,也就简捷地假设了上述要表述的内容。

(6) 输入的分布问题。由模型所假设的输入的分布是重要的,因为可靠性估计紧密地依赖于它。例如,作为一个极端的情况,如果输入是一个常数(比如说只用到一个输入),软件将或者出错或者成功地执行,于是就给出可靠性相应地为 0 或是为 1。

(7) 关于软件的复杂性问题。复杂的软件应该比简单的软件要求更多的测试。但是,大多数现有的软件可靠性模型对此都没有予以考虑。

(8) 模型的验证问题。对于发表的模型进行足够的验证是绝对必要的,但由于缺乏实际可用的足够数据,使得对模型的验证无法进行。而且在整个软件寿命周期内,软件几乎呈常数倍地增加,导致可靠性也相应地变化,使得对软件可靠性模型的验证工作更加复杂化。

(9) 关于时间问题。暴露期应该与外部因素,如机器执行时间无关。但是对于某些软件(如一个操作系统),很难断定什么构成了暴露期的基本单位——运行次数——所指的一次运行到底是什么,因此,这时的时间单位就应该是指 CPU 时间。在软件可靠性量测与硬件可靠性量测综合起来对一特定的系统环境进行考核时,将 CPU 时间作为时间单位也是必要的。

(10) 对于模型所要求的数据,应该考虑是容易进行收集的,否则,由于数据问题,也将限制软件可靠性模型的应用范围。

软件可靠性模型,对于软件可靠性估测,起着核心的作用。而对于软件的质量保证有直接意义的模型,就是它们的参数能够以软件的故障历史作为基础来加以估计的那些模型。它们都有一个共同的前提:软件故障能够用某一个概率分布来加以描述。所有的模型

都要求输入故障数据,但它们要求输入的数据在形式上是不尽相同的。它们一般有:日历时间(如:小时数、天数,等)、操作时间(基本上就是计算机机器时间)、执行时间(即 CPU 时间)、排错时间、故障间隔时间、观察到的故障数、排除的错误个数等。而模型的输出,根据它们的估测目的,形式也是不一样的,它们计有:软件中的初始错误个数、剩余错误个数、故障密度函数值、达到特定可靠性指标(或故障率指标)所需继续测试的时间,等等。

§ 4.2 模 型 的 分 类

虽然到目前为止,已经发表了近百种软件可靠性模型,但由于对软件可靠性数学模型的研究仍处在一个初级阶段,目前尚没有一个完整、系统而且科学的分类方法。但是,为了整理、研究这些模型,又需要对它们作一些分类。于是,不少学者对这一问题进行了研究,提出了许多种不同的分类方法。

模型分类可以按它们的假设、测试空间、软件结构、处理的方式方法等进行。

本书采用的是综合模型的假设、测试环境以及数理统计的方法,将模型大致分为:马尔可夫过程模型、泊松过程模型等随机过程类模型,以及非随机过程类模型。另外,究其实质,Musa 的执行时间模型本属于泊松过程模型,但我们将它们单独作为执行时间类模型一类。在随后的第五、第六章中,我们就按这种十分粗略的分类法,介绍它们。

除了这些之外,可靠性模型还可以根据模型本身的数学结构及使用的参数估计方法进行分类。

另外,一些常见的分类方法还有:

- 按随机性分类法:根据随机过程的假设,如过程的确定性或非确定性、马氏过程、泊松过程等,进行分类。
- 按软件出现的故障数进行分类:主要有错误计数模型和非计数模型,可数性或不可数性(无穷)模型。
- 按模型参数的估计方法进行分类:主要有贝叶斯方法或非贝叶斯方法,最大似然估计法或最小二乘法,另外还有线性模型等。
- 按模型使用的时间方式分类:主要有日历时间和执行时间模型。
- 按修复过程分类:主要指强调对软件系统修复过程的一类模型,如:完全修复型的和不完全修复型的,完全排错型的和不完全排错型的模型。
- 按对软件的内部结构是否了解进行分类:可分为黑箱模型和白箱模型。对它们的分类主要根据对软件的内部结构的了解程度以及对它们的结构能加以利用的程度来区分。

下面我们再介绍两种具体的分类方法:一个是由 Ramamoorthy 和 Bastani 开发的根据模型应用于软件寿命周期的阶段进行分类的方法,另一个是由 Musa 和 Okumoto 开发

的分类方法。

先介绍 Ramamoorthy 和 Bastani 的分类方法。

Ramamoorthy 和 Bastani 根据模型是否被应用于软件的开发阶段、验证阶段、操作运行阶段和维护阶段,对它们进行分类。

绝大多数现有模型都是基本上根据软件的过去的行为预测它将来的行为,根据它的错误发生史以估计期望的可靠度。在软件的开发阶段,软件错误一旦被查出,立即就改正它。应用于这一阶段的模型往往都假设在改错时不引入新的错误,因此,软件的可靠性是随测试和分析的深入进行而逐渐增加的。于是,应用于这一阶段的软件可靠性模型又被称为可靠性增长模型。它们又可进一步分为错误个数计数模型和非计数模型。从应用的观点看,非计数模型仅估计软件的可靠性,而计数模型则不但估计软件的可靠性,也估计软件中的剩余错误个数。

在软件的验证阶段,错误被查出以后并不在本阶段改正它们。因而,这些模型往往可以应用于对可靠性要求较高的软件系统,如:核电站的安全控制系统中的软件。

在操作运行阶段,连续不断输入到程序的输入不是独立的,它们通常遵循某种分布。因此,应用于这一阶段的软件可靠性模型,应该根据实际的输入所依从的分布来计算软件的可靠性。如果输入的分布发生了变化,就应重新对软件的可靠性进行计算。

在软件的维护期间,软件的错误被改正了,使得可靠性有小的增加,但是,新的特性和功能又加到软件中来,势必要引入新的错误。因此,在维护期间,软件可靠性不可能有大的增加。于是,再将可靠性增长模型应用于维护阶段,就不是很合适的做法。新的可靠性指标能够用于在验证阶段使用的软件可靠性模型。但是,使用少量的测试条件与数据来估计可靠性的变化是可能的,但要保证不致破坏原有的特性和功能。另外,正确性的测量也是一个重要的软件可靠性模型。

Ramamoorthy 和 Bastani 的分类方法概括在表 4.1 中。图 4.1 示出他们的软件可靠性模型分类法。

下面介绍 Musa 和 Okumoto 的软件可靠性模型分类方法。

Musa 和 Okumoto 的分类方法,强调模型之间的关系,以减少对模型进行比较的工作。模型被按五种不同的属性加以分类:

- 时间:日历时间还是执行时间(CPU 或处理机时间)。
- 类型Ⅰ:在有限时间范围内,出现故障数是有限的,还是无限的。
- 类型Ⅱ:到时间 t 时出现的故障数的分布。
- 类型Ⅲ:(仅限于有限故障类型Ⅰ)作为时间函数的故障密度的函数形式。
- 族(仅限于无限故障类型Ⅰ)作为出现的故障期望数的函数的故障密度的函数形式。

之所以这样分类,是为了极大地简化分析,并且还有其具体的意义。对于泊松和二项式分布的有限类型Ⅰ,类型Ⅲ则既表示了作为时间的函数的故障密度函数形式,也表示了作为单个错误的故障时间分布的函数形式。类型Ⅲ就根据这些分布进行命名。故障间的

表 4.1　Ramamoorthy 和 Bastani 软件可靠性模型分类法一览表

分类模型		特性						
		程序设计语言方法学独立	测试过程	新的错误	表述内容	输入分布	复杂性	时间
开发阶段	J-M 模型 Shooman 模型 Schick-Wolverton 模型 Halden 计划 马尔可夫过程(Trivedi-Shooman)	考虑	不考虑	不考虑	间接方式	考虑	不考虑	t
	Musa 执行时间模型	考虑	部分考虑	部分考虑	间接方式	考虑	不考虑	t
	随机过程(输入域)	考虑	考虑	考虑	间接方式	考虑	不考虑	n
	(Littlewood-Verrall)	考虑	不考虑	考虑	间接方式	考虑	不考虑	t
验证阶段	统计模型(Nelson, Brown, Lipow)	考虑	不考虑	—	考虑	考虑	间接方式	n, t
	输入域(包括正确性概率)	考虑	考虑	—	考虑	考虑	考虑	n
操作运行阶段	输入域	考虑	—	—	—	考虑	—	n
	马尔可夫过程(Littlewood, Cheung)	考虑	—	—	—	考虑	—	t
正确性测量	Seeding(Basin, Mill)	考虑	考虑	不考虑	不考虑	—	间接方式	—
	现象学(Halstead)	不考虑	—	—	—	—	部分考虑	—
	程序变换(Demillo, Lipton)		考虑	考虑	—	间接方式	—	间接方式
	统计模型(Ehrenberger)	考虑	不考虑	—	考虑	考虑	考虑	—

说明:“考虑”:表示包括了该项特性。

“不考虑”:表示不包括该项特性。

“间接方式”:表示对该项特性的包括是作为与之无关的某假设的结果。

“部分考虑”:表示该模型试图包括该项特性。

“—”:表示该项特性对于该模型是不可能的。

“n”:执行次数;

“t”:连续时间。

图 4.1　Ramamoorthy 和 Bastani 的软件可靠性模型分类方法

分布则与类型Ⅲ和类型Ⅱ两者都有关。无限故障模型的族是根据作为期望的故障数的函数的故障密度的函数形式来命名的。

表 4.2 表示了根据后四种属性来进行分类的方法。对于两类不同的时间作了相同的处理。大部分模型可以划入有限故障类型Ⅰ。注意, 只有 Musa 的执行时间模型和 Musa-Okumoto 的对数泊松执行时间模型被明确定义为执行时间类。其它的模型或者是指日历

时间,或时间并未明显定义为哪一类。表中用“T_1”、“T_2”、“T_3”来代表类型Ⅱ;另外用“C_1”代表每个错误的故障分布类型Ⅲ。关于无限故障类型Ⅰ,一个特定的族可以导致一个特殊的模型类型Ⅱ。表中的空格并不意味着能够提出某个模型填入其中,某些族和类型Ⅱ或类型Ⅲ和类型Ⅱ的组合是不可能的。

表 4.2　Musa-Okumoto 的分类方法

	类型Ⅲ	类型Ⅱ		
		泊松分布	二项式分布	其它类型Ⅱ
有限故障类型Ⅰ的模型	指数型	Musa(1975) Moranda(1975) Schneidewind(1975) Goel-Okumoto(1979)	Jelinski-Moranda (1972) Shooman(1972)	Goel-Okumoto(1978) Musa(1979) Keiller-Littlewood (1983)
	Weibull 型		Schick-Wolverton (1973) Wagoner(1973)	
	C_1		Schick-Wolverton(1978)	
	Pareto 型		Littlewood(1981)	
	Γ-型	Yamada-Ohba-Osaki (1983)		

	族	类型Ⅱ			
		T_1	T_2	T_3	泊松分布
无限故障类型Ⅰ的模型	几何型	Moranda(1975)			Musa-Okumoto (1984)
	反线型		Littlewood-Verrall (1973)		
	反多项式(二次)			Littlewood-Verrall (1973)	
	幂				Crow(1974)

§ 4.3　模型的假设与局限性

在随后的两章里,我们将看到,对于每一个软件可靠性模型,都必须以某些假设作为

基础。这些假设是软件可靠性模型建立的主要依据,也是模型在理论上如何进行处理的先决条件。由此可见,模型的成功与否,与模型所取的假设有着很大的关系。

总起来说,模型的假设有以下几个方面的内容:

1. 关于软件出错行为的描述,它们可能有下列几点:
   (1) 软件中的初始错误个数 N_0 为固定不变的常数;
   (2) 软件中的初始错误个数 N_0 与故障率 $\lambda(t)$ 成正比;
   (3) 软件故障的发生彼此独立;
   (4) 错误检测率在固定的期间内是恒定的;
   (5) 每个错误具有相同的故障发生率;
   (6) 错误彼此独立,并且有为常数的发生率;
   (7) $N_0 = N(t)$,是随机变量;
   (8) 故障发生率服从 Γ-分布;
   (9) 故障出现率服从泊松分布;
   (10) 故障间隔时间服从指数分布;
   (11) 其它被假定的分布还有:二项式分布、正态分布、Weibull 分布、Rayleigh 分布,等等。
2. 关于测试与排错环境、行为的描述,主要的有下列内容:
   (1) 一次排除一个错误;
   (2) 完全排错;
   (3) 不引入新的错误;
   (4) 立即排错;
   (5) 非立即排错;
   (6) 测试环境与软件的使用运行环境完全一致;
   (7) 可以允许排错时至多引入一个错误;
   (8) 可以是不完全排错;
   (9) 测试的输入数据与条件完全随机选取;
   (10) 测试的输入空间‘覆盖’软件将来的使用空间;
   (11) 每次查出、改正或引入的错误个数可以大于 1 个;
   (12) 软件中的错误分为容易查出和改正的以及不容易查出和改正的两大类;
   (13) 某些软件错误在其它的软件错误被排除之前,是不可能被查出来的。
3. 其它一些方面的内容:
   (1) 剩余错误个数 $N_r(t)$ 正比于最后一次测试的时间长度 Δt;
   (2) 故障率 $\lambda(t)$ 正比于以前 $i-1$ 个故障间隔时间及半个第 i 个故障间隔时间的和:

$$\sum (t_{i-1}) + \frac{t_i}{2} \quad ;$$

   (3) 相连接的时间区间内的故障发生率以几何形式(等比级数)递减;
   (4) 故障率为分段的常数;
   (5) 错误改正率为 $p(0 \leq p \leq 1)$,不完全改正率为 q,则 $p + q = 1$;

(6) 软件错误的出现/改正率为 $\lambda(k,t)=p\lambda(t)(N_0-k)$，错误的引入率为 $v(k,t)=r\lambda(t)(N_0-k)$；

(7) 软件故障数与 $N_r(t)$ 成正比。

凡此种种，举不胜举。

从以上列举的许多假设来看，有很多是与软件开发实际不相符合的，许多软件工程师与软件管理人员无法接受。下面择其一二，试剖析之：

许多现存模型(特别是那些早期的软件可靠性模型)，考虑到排错引入新的错误会使问题复杂化，于是假设排错不引入新的错误。这样做的结果虽然使理论上的处理简单了，但与实际情况相距太远。软件的开发靠人完成，则排错问题要人完成，人类行为的不可预测性无论在开发还是排错一个软件，同样要表现出来。事实上，由于排错时的某些处置失当，往往会产生许多副作用，引入一些始料不及的新错误，是十分自然的。这也正好解释了我们在对软件中出现的错误进行观察记录时，为什么经常会大幅度地振荡的原因。引入新错，另一方面的原因还在于软件产品各模块(指结构化的软件产品而言)间的逻辑关系错综复杂、互为因果，故而使得局部的某些改动甚至可能产生牵涉全局性的许多问题。

随后的一些模型，虽然允许可以引入的错误数为 1，为 2，或其它，正是由于看到了这一问题，才作了一些调整变动，但仍未能从根本上解决问题。

关于故障率为常数的问题，根据实际的观察，不同错误的发生率因人而异，变化很大，即使是同一个程序员，在不同时期，变化也很大。那些将所有错误处理成具有相同出现率的模型是十分不现实的。理论上也已经证明，这样做的结果，必然会产生乐观的估计偏差。另一方面，对具体错误不加区分，我们实际观察到的结果，也显示出故障率不可能为常数。图 4.2 是我们观察到的结果，它是我们在对开发的一个项目进行系统测试期间，观察记录

图 4.2　开发项目 WPADT 的累积故障数

下来的。图中横轴 t 表示实际测试时的“工作周”，每“周”以 6 天计(它有别于通常日历时间的星期)，简记为 WW，纵轴 n 为每“周”观察记录的累积故障数。从图中可看出，在系统测试的初期，错误是多发性的，而且有大幅度的振荡。究其原因，在于测试初期，软件产品未经充分排错，且在测试开始设计测试用例(test cases)时的原则，就是要通过它们使错误多暴露，暴露得越充分越好。在系统测试的后期，软件产品经过了较充分的排错以后，故

障率自然也就随之下降了。

关于测试时的输入空间“覆盖”使用空间的假设,也是不现实的。为了尽量保证测试能充分反映出软件产品将来所有可能的使用情况,有必要在设计测试的数据和条件时,使它们能尽可能地反映出软件产品将来使用时的情况、条件和环境。但这并不意味着测试就一定是完全的。测试时的输入空间充其量也只能是从使用时的输入集合的全集合中选取的子集合。如果使用时的问题空间是无穷集合(大多数实际情况也正是如此),则这样选 出的子集合,无论如何也不可能“覆盖”问题空间。假如一定要做到“覆盖”,只有使测试无限制地进行下去,而这也正是我们力图要避免的。

由“覆盖”问题引伸出的测试环境与使用运行环境一致的问题,也是同样性质的,这一假设也是人们一种良好愿望的反映。特别对于那些包括软件的无法进行试验的系统,则这一假设更显得不合实际。

不同软件产品的开发过程,由于参加者不同,他们各自的训练、业务经历、程序设计风格都不相同。因此,通过他们各自的大脑思维所产生出来的“逻辑产品”,个性多于共性,这是一种必然现象,也正是软件工程所面临的一大难题。

模型假设的局限性太多,势必影响到它们的应用范围。目前,软件工程界对于软件可靠性模型的诸多疑虑,也多半来自于此。如何改进它们,是软件可靠性今后理论研究的重大课题之一。它的突破,也就会消除软件工程界的疑虑,使软件可靠性理论得到更广泛的应用,从而,必然反过来又促进软件可靠性理论的发展。

§ 4.4 Musa 的执行时间概念

虽然软件可靠性可以用许多变量来定义,但通常都将它定义为时间的函数。关于计算机,一般有三种类型的时间概念是经常使用的:一是关于某个软件的执行时间概念,它是指实际花在执行该软件的程序的指令上所用的处理机时间,也即指 CPU 时间;二是日历时间概念,即平常人们在日常生活中所习惯使用的时间;三是计算机使用的时钟时间,即指用于软件的时间,它包括在正常运转的计算机上执行软件从开始启动到终止时所花去的时间,以及等待时间和其它软件执行的时间,但除开计算机停机时间。如果计算机由该软件使用的利用率(处理机正在执行该软件的时间段)是一个常数,则时钟时间将与执行时间成正比。

Musa 认为在上述三种时间概念中,执行时间是最重要的一种,因为它才是软件发生故障的最基本的测量单位。因此,他提出的执行时间模型和由他与 Okumoto 合作提出的对数泊松执行时间模型,都以执行时间作为基本的时间量度单位。在由时间来描述的四种故障发生特性的基本方式中,也都是关于执行时间来定义的。这四种基本方式是:

- 故障发生时间(亦称为故障时间);
- 故障间的时间间隔(又称故障间的时间区间);

· 到某指定时刻为止的累积故障数(简称作累积故障数);
· 在一给定长度的时间区间内发生的故障数。

但是为了照顾到软件工程师以及软件管理人员的习惯,这两个模型除了执行时间以外,都增加了日历时间部分,以将估测的执行时间换算成人们日常生活中已习惯的日历时间表示。

可是,在目前众多的软件可靠性模型中,除了这两个模型以外,都是或者采用日历时间,或者并不明确指出是什么时间。只有 Musa 和 Okumoto 的模型是采用执行时间的。下面的图 4.3 分别示出平均发生故障数 μ(τ)及故障密度 λ(τ)与执行时间 τ之间的关系。图 4.4 则示出在一段执行时间范围内,软件可靠性 R 与 τ之间的变化关系。

图 4.3 μ(τ)、λ(τ)与 τ的关系

图 4.4 在一段执行时间 τ内可靠性 R 的变化

由图 4.4 可见,在一段无故障的执行时间内,软件可靠性 R 是由 1 向 0 变化的:一开始执行时,由于软件不发生故障,所以可靠性为 1,但随着执行时间 τ的增加,逐渐下降,一旦到发生故障,则软件可靠性变为 0。

第五章　随机过程类模型及应用

本章介绍随机过程类的软件可靠性数学模型的发展以及实际应用的例子。在软件的测试过程中,一旦其中的错误被查出,一般都要进行排错。因此,随着测试工作的进展,软件中的错误不断被排除,于是,软件的可靠性就不断提高。可以这样说,现有的软件可靠性数学模型,都是软件可靠性增长模型。

随机过程类的软件可靠性数学模型主要包括马尔可夫过程模型(Markov Process Model)和非齐次泊松过程模型(Nonhomogeneous Poisson Process Model)。一般如假定错误出现率在软件无改动的区间内是常数,并且随着错误数目的减少而下降,这样的模型属于马尔可夫过程模型;另外,排错过程中的累积错误数目作为时间的函数: $N(t)$,在一定条件下可以近似为一个非齐次泊松过程,这一类的数学模型则属于非齐次泊松过程模型。

通常所说的 Jelinski-Moranda 模型,简称为 JM 模型,属于马尔可夫过程模型;Goel-Okumoto 模型,简称为 G-O 模型,属于非齐次泊松过程模型。另外,Musa 的执行时间模型,究其实质应划入马尔可夫过程类;而 Musa-Okumoto 的对数泊松执行时间模型,应归入非齐次泊松过程类。但它们都是以程序的执行时间,即 CPU 时间为基本的测度,所以我们将它们单独作为一类模型予以介绍,并在技术上将对数泊松执行时间模型作为对基本的执行时间模型的推广。

§ 5.1　马尔可夫过程模型

JM 模型是最具代表性的早期软件可靠性马尔可夫过程的数学模型。随后的许多工作,都是在它的基础上,对其中与软件开发实际不相适合的地方进行改进而提出来的。因此,在这个意义上来说,JM 模型又是对后面的工作有着广泛影响的模型之一。

5.1.1　JM 模型介绍

JM 模型是 Z. Jelinski 和 P. Moranda 于 1972 年提出的软件可靠性数学模型。它的假设是:

(1) 软件中的初始错误个数为一个未知但固定的常数,用 N_0 表示。

(2) 错误一旦被查出即被完全排除,于是每次排错之后,N_0 就要减去 1。

(3) 在任何时候的故障率都与软件中的剩余错误个数成正比,正比例常数用　表示。

其实，一开始Jelinski和Moranda提出的模型假设只有一个，即假设(3)。前面的假设(1)和(2)都是后来人们根据Jelinski和Moranda在模型中对问题的实际处理过程归纳出来的，但既然为大家共同使用，也就自然成了模型的假设了。

根据假设，在第一个错误被排除之后，故障率就由 N_0 变为 (N_0-1)，以下以此类推。

设 $t_1, t_2, \cdots, t_n$ 表示相继出现的错误之间的时间区间样本。根据假设，在每两个错误之间只有唯一的故障率，关于 t_i 的密度即为：

$$P(t_i) = [N_0 - (i-1)]\exp\{-[N_0-(i-1)]t_i\} \qquad (5.1.1)$$

模型中含有两个未知参数 N_0 和 。下面讨论JM模型的参数估计问题。

由(5.1.1)式，可得到似然函数为：

$$L(t_1, t_2, \cdots, t_n) = \prod_{i=1}^{n} [N_0 - (i-1)]\exp\{-[N_0-(i-1)]t_i\} \qquad (5.1.2)$$

其中，n为样本大小，即到当前时刻为止，排除的错误个数。

对(5.1.2)取对数，得到对数似然函数：

$$\ln L = \sum_{i=1}^{n} \ln\{[N_0 - (i-1)]\} - \sum_{i=1}^{n} [N_0 - (i-1)]t_i$$

$$= \sum_{i=1}^{n} \ln[N_0 - (i-1)] + n\ln - \sum_{i=1}^{n} [N_0 - (i-1)]t_i \qquad (5.1.3)$$

对(5.1.3)分别对 N_0 和 求偏导，并令结果为零：

$$\frac{\ln L}{N_0} = \sum_{i=1}^{n} \frac{1}{N_0 - (i-1)} - \sum_{i=1}^{n} t_i = 0 \qquad (5.1.4)$$

$$\frac{\ln L}{} = \frac{n}{} - \sum_{i=1}^{n} [N_0 - (i-1)]t_i = 0 \qquad (5.1.5)$$

令 $\sum_{i=1}^{n} t_i = T$，于是从(5.1.5)可解出：

$$= \frac{n}{N_0 T - \sum_{i=1}^{n} (i-1)t_i} \qquad (5.1.6)$$

(5.1.4)可以写成：

$$\sum_{i=1}^{n} \frac{1}{N_0 - (i-1)} = \frac{nT}{N_0 T - \sum_{i=1}^{n} (i-1)t_i} = \frac{n}{N_0 - \frac{1}{T}\sum_{i=1}^{n} (i-1)t_i} \qquad (5.1.7)$$

方程(5.1.7)不含 ，而由测试收集的数据，可计算出 $T = \sum_{i=1}^{n} t_i$ 与 $\sum_{i=1}^{n} (i-1)t_i$ 的值。将它们代入(5.1.7)，则可解出 N_0 的估计值 N_0：

$$\frac{1}{N_0} + \frac{1}{N_0 - 1} + \cdots + \frac{1}{N_0 - (n-1)} = \frac{n}{N_0 - \sum_{i=1}^{n} t_i^{-1} \cdot \sum_{i=1}^{n} (i-1)t_i}.$$

再将 N_0 代入(5.1.6)，可解出 的估计值 $\hat{}$：

$$\hat{} = \frac{n}{N_0 \sum_{i=1}^{n} t_i - \sum_{i=1}^{n} (i-1)t_i}.$$

5.1.2 模型的推广

1972 年, 由 Schick-Wolverton 提出的 Weibull 模型, 实际上是 JM 模型的 Weibull 公式, 它假定故障间的时间是独立的, 并有形如下式的密度:

$$f(\tau) = (N_0 - i + 1)\tau \exp - \frac{1}{2}\tau^2(N_0 - i + 1),$$

它的 MTTF 为:

$$MTTF = \int_0^{\infty} \tau f(\tau) d\tau$$

$$MTTF_i = \int_0^{\infty} (N_0 - i + 1)\tau \exp - \frac{1}{2}\tau^2(N_0 - i + 1) d\tau$$

1974 年, Lipow 建议对上述模型作适当修改, 他认为 τ 应由 $\frac{1}{2}\tau + \sum_{j=1}^{i-1}\tau$ 代替, 因此得出:

$$f(\tau) = (N_0 - i + 1)\left(\frac{1}{2}\tau + t_{i-1}\right) \exp - (N_0 - i + 1)\left(t_{i-1} + \frac{1}{2}\tau\right)^2.$$

1975 年, Moranda 本人发表的几何 De-Eutrophication 模型中, 认为故障率随时间的增加呈几何级数形式下降, 并以此得到:

$$f(\tau) = \beta^{i-1}\exp(-\beta^{i-1}\tau).$$

Shooman 和 Trividi 的马尔可夫模型定义了软件的“Up”和“Down”两种状态。设软件处于“Up”状态(正常工作状态), 则软件具有状态序列:

$$n_0, (n_0 - 1), (n_0 - 2), \cdots$$

之中任何一个状态; 否则软件处于“Down”状态(发生故障状态), 则软件具有状态序列:

$$m, (m - 1), (m - 2), \cdots$$

之中任何一个状态。系统结构示于图 5.1。

图 5.1 系统结构

设观察到 i 个故障的概率是 $P_{N_0 - i}$, 则:

$$P_{N_0 - i}(t) = \frac{\lambda\mu}{\mu - \lambda} e^{-\lambda t} \sum_{j=0}^{i} \frac{t^{i-j}}{(\mu - \lambda)^j (i - j)!} [(-1)^{i+1} c_{ij} e^{-(\mu - \lambda)t} + (-1)^j d_{ij}],$$

其中,

$$c_{ij}=\begin{cases}0, & (j=0);\\ 1, & (j=1);\\ \dbinom{i+j-1}{j-1}, & (j>1).\end{cases}\qquad d_{ij}=\begin{cases}1, & (j=0);\\ \dbinom{i+j-1}{j}, & (j>0).\end{cases}$$

$$P_{m-i}(t)=\frac{1}{\mu}\left(\frac{\lambda\mu}{\mu-\lambda}\right)^{i+1}e^{-\lambda t}\sum_{j=0}^{i}\frac{t^{i-j}}{(\mu-\lambda)^{j}(i-j)!}c_{ij+1}$$
$$¡¤[(-1)^{j}+(-1)^{i+1}e^{-(\mu-\lambda)t}].$$

于是,可以计算系统的有效性为:

$$A(t)=\sum_{i=0}^{N(t)}P_{N_0-i}(t).$$

1977 年, Goel 提出包括不完全排错的马尔可夫模型,仍是 JM 模型的马尔可夫公式表示:

$$P_i(t)=\sum_{j=0}^{N_0-i}\binom{N_0-i}{j}\binom{N_0}{i}(-1)^{j}e^{-p\,t(i+j)}.$$

作为一个特例,有:

$$P_0(t)=\sum_{j=0}^{N_0}\binom{N_0}{j}(-1)^{j}e^{-p\,tj},$$

或更密集的形式:

$$P_0(t)=(1-e^{-p\,t})^{N_0}.$$

设软件达到无错状态时间的概率密度函数是 $g_0(t)$,则:

$$g_0(t)=\frac{P_0(t)}{t}=\sum_{j=1}^{N_0}\binom{N_0}{j}(-1)^{j+1}pje^{-ptj}.$$

1980 年, Shanthikumar 的马尔可夫模型给出剩余错误数的二项式分布:

$$N(t)\sim Bin(N_0,1-e^{-Q(t)}),$$

$$R(\tau\mid N(t))=\exp-[N_0-N(t)\int_{t}^{t+\tau}\lambda(s)ds,$$

$$f(\tau\mid t_i)=[N_0-N(t_i)]\lambda(t_i+\tau)\exp-\int_{t_i}^{t_i+\tau}\lambda(s)ds^{[N_0-N(t_i)]}.$$

1985 年, Kremer 给出将软件可靠性作为生灭过程的公式。在他的模型中,假定:

- 软件状态的转换只是从一个状态向它相邻状态的转移,在任一状态中最多允许改正或引入 1 个错误。
- 错误的出现/改正率为:

$$\lambda(k,t)=p\lambda(t)(N_0-k).$$

- 错误引入率为:

$$v(k,t)=r\lambda(t)(N_0-k),$$

其中,

p = P(出现故障后的成功改错),

r = P(出现故障后的不成功改错并引入新错).

由此得到:

$$P_i(t) = \sum_{j=0}^{i} \binom{N_0}{j} \binom{N_0 + i - j - 1}{N_0 - 1} r^{(N_0 - j)} \beta^{(i-j)} (1 - r - \beta)^j,$$

其中,

$$r = 1 - [e^{Q(t)} + A(t)]^{-1},$$

$$\beta = 1 - e^{Q(t)}[e^{Q(t)} + A(t)]^{-1},$$

$$Q(t) = (p - r)\int_0^t \lambda(u)du,$$

$$A(t) = \int_0^t v(s)e^{Q(s)}ds.$$

当前错误数的期望值为:

$$E[N_r(t)] = N_0 e^{-Q(t)}.$$

剩余错误数的方差为:

$$V[N_r(t)] = N_0 e^{-2Q(t)}\int_0^t [v(s) + \lambda(s)]e^{Q(s)}ds.$$

错误间的时间分布为:

$$R_0(\tau) = \exp[-\int_t^{t+\tau} \lambda(s)ds],$$

$$R(\tau \mid N(t)) = R_0(\tau)^{N_0}\left[\frac{p}{R_0(\tau)} + q + rR_0(\tau)\right]^{N(t)},$$

其中,

q= P(故障之后的不成功改错或当排除已发现的错误时引入新的错误).

在得到 $R(\tau \mid N(t))$ 之后,可以直接得出:

$$f(\tau \mid t) = -\frac{R(\tau \mid N(t))}{\tau}.$$

如果设 p = 1, q= 0, r= 0, $\lambda(t)=$,就得到 JM 模型。

1986 年,由 Sumita 和 Shanthikumar 提出的多故障模型,是一个新的马尔可夫过程型的模型,它允许在同一时刻,查出、改正或引入的错误个数可以多于 1 个。它引入转换矩阵及初始分布,分别用 p 和 a 表示:

$$a = \{a_i\}, \quad a_i = P(N_0 = i),$$

有别于其它模型中的状态概率 $p_i(t)$,这里得出向量 $\bar{p}(t \mid a)$:

$$\bar{p}(t \mid a) = e^{-tk\lambda}\sum_{k=0}^{\infty} \frac{(tk\lambda)^k}{k!} a\left(I + \frac{1}{k\lambda}Q\right)^k,$$

其中,

k 是在软件中可能的错误数的最大数目;

λ是常数;

I 是单位矩阵;

$$Q = -\ k\lambda(I - a_k), \quad a_k = I + \frac{1}{k\lambda}Q. \tag{5.1.8}$$

条件概率向量 $\bar{\alpha}(t_0, t)$ 定义为:

$\bar{\alpha}(t_0, t, n)$ = P(在时刻 t 系统中有 i 个错©¦在区间[0, t_0] 中查出 n 个错).

特别,

$$\bar{\alpha}(t_0, t, n) = \frac{\bar{f}_n(t_0©¦)}{\sum_{i=0}^{k} f_{ni}(t_0©¦)},$$

其中,

$f_{ni}(t_0©¦)$ = P(在[0, t_0] 中观察到 n 个错, 在时刻 t 还有 i 个错).

获得 $\bar{f}_n$ 并进而获得 f_{ni} 的最简单方法是在(5.1.8)中以 T 代替 Q:

$$T = \begin{pmatrix} -v_D & v & 0 & \cdots & 0 & 0 \\ 0 & -v_D & v & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & -v_D & 0 \\ 0 & 0 & 0 & \cdots & 0 & -v_D \end{pmatrix},$$

$$v_D = \{i\lambda\delta_{ij}\},$$

$$v = \{i\lambda p_{ij}\}.$$

M. Xie 在 1987 年的工作, 实际上修正了 JM 模型在事实上关于每个错误的"大小"相等这一显然不正确的假设。Xie 认为, 对于查出并改正的累积错误数 N(t), t≥0, 通常可用一个马尔可夫过程来描述。设从状态 i 转换到它的相邻状态 i+1 的跳跃函数为 λ(i), 此过程又可称为生过程, 则直接应用关于连续时间的马尔可夫链理论, 可得 N(t) 的分布, 即:

$$\{P_i(t) = P(N(t) = i)\}, \quad i = 0, 1, \cdots, N, \quad t \geq 0$$

是:

$$P_0(t) = e^{-\lambda(0)t},$$

$$P_1(t) = \frac{\lambda(0)}{\lambda(1) - \lambda(0)}(e_1 - e_0),$$

$$\vdots$$

$$P_i(t) = \sum_{j=0}^{i} A_j^{(i)} e_j, \quad i < N,$$

$$\vdots$$

$$P_N(t) = -\sum_{j=0}^{N-1} A_j^{(N-1)} ¡¤\frac{\lambda(N-1)}{\lambda(j)} ¡¤e_j,$$

其中,

$$e_j = e^{-\lambda(j)t}.$$

可以利用下面的递归算法求得:

$$A_j^{(i)} = \frac{\lambda(i-1)}{\lambda(i) - \lambda(j)} ¡¤A_j^{(i-1)}, \quad j < i,$$

以及:

$$A_i^{(i)} = -\sum_{j=0}^{i-1} A_j^{(i)}.$$

实际上, JM 模型是本模型的一个特例, JM 模型有: $\lambda(i) = (N - i + 1)$, 因为它实际上假设了每个错误的"大小"一样, 故障率的降低与剩余错误数成线性关系。Xie 在 1987 年提出了两个非线性模型, 用以修正这一假设, 它们是:

$$\lambda_M(i) = \phi_1(N - i + 1)^{\alpha}$$

及:

$$\lambda_M(i) = \phi_2[e^{\beta(N-i+1)} - 1].$$

5.1.3 应用实例

在马尔可夫过程模型的跳跃函数 $\lambda(i)$ 中, 含有许多未知参数, 这些参数要通过实际测试收集的数据来加以估计。下面我们讨论用最大似然法估计未知参数的方法。

假定通过测试, 收集到如下一组数据:

$$\{t_i, \quad i = 1, 2, \cdots, n\},$$

最大似然函数为:

$$L(n, \lambda(i)) = \prod_{j=1}^{n} \lambda(j) e^{-\lambda(j)t_j},$$

则:

$$\ln L = \sum_{j=1}^{n} [\ln \lambda(j) - \lambda(j)t_j]. \tag{5.1.9}$$

函数 $\lambda(i)$ 中的参数可以通过求(5.1.9)的最大值来求出, 这通常需要利用一些计算方法以求得结果。

但是, 对于一些模型中的比例常数, 比如: $\lambda_M(i)$ 中的 ϕ_1, 可以直接求得。令:

$$\frac{\ln L}{\phi_1} = 0,$$

可得:

$$\phi_1 = n / [\sum_{i=1}^{n} (N - i + 1)^{\alpha} t_i].$$

将它代入(5.1.9)式, 则可将其中的维数从三维降至二维, 再利用计算方法, 以求结果, 可以简化计算过程。

上面的过程应用于 Currit 等人提供的测试数据(测试的经验数据, 表中的数据项都是执行时间), 见表 5.1。从表上可以看出, 虽然可靠性有所改善, 但变化仍很大。

表 5.1 经验数据

85, 85, 479, 965, 469, 385, 796, 277, 927, 340,
405, 150, 277, 503, 496, 620, 4050, 3064, 522, 1797,
2329, 3798, 2775, 2393, 909, 994, 28212, 14956, 1971, 5308

表 5.2a～表 5.2c 给出固定的 α,N 和 λ 的最大似然估计。从表中可看出 α越大,估计的下一次故障密度的结果越保守。有时 α的最大似然估计结果不合适,如果存在,也总比 1 大。与前面指出的,我们认为 α= 2 是比较起来最合适的估计。表 5.3 中给出在 α= 2 及观察数 n 时,N 和 λ 的最大似然估计。

表 5.2a　n= 20 的估计结果

α	N	logL	$\lambda \cdot N^{\alpha}$	λ(n+ 1)
1.00	22	- 150.89314	0.00290	0.00026378
1.50	25	- 150.85446	0.00333	0.00029760
2.00	30	- 150.87079	0.00336	0.00037307
2.50	34	- 150.88899	0.00351	0.00038184
3.00	39	- 150.90386	0.00352	0.00040742
3.50	44	- 150.91609	0.00354	0.00042399
4.00	48	- 150.92456	0.00363	0.00042042
5.00	58	- 150.93744	0.00364	0.00043902

表 5.2b　n= 25 的估计结果

α	N	logL	$\lambda \cdot N^{\alpha}$	λ(n+ 1)
1.00	27	- 196.06515	0.00224	0.00016592
1.50	30	- 195.60388	0.00270	0.00018369
2.00	34	- 195.42690	0.00295	0.00020655
2.50	38	- 195.34742	0.00315	0.00021538
3.00	43	- 195.30190	0.00320	0.00023464
3.50	48	- 195.27704	0.00324	0.00024696
4.00	52	- 195.26063	0.00337	0.00024467
5.00	62	- 195.24128	0.00341	0.00025810

表 5.2c　n= 30 的估计结果

α	N	logL	$\lambda \cdot N^{\alpha}$	λ(n+ 1)
1.00	30	- 250.40751	0.00161	0.00000000
1.50	31	- 248.33310	0.00240	0.00001392
2.00	33	- 247.68058	0.00299	0.00002468
2.50	36	- 247.43676	0.00329	0.00003729
3.00	39	- 247.36666	0.00356	0.00004376
3.50	43	- 247.33878	0.00360	0.00005462
4.00	46	- 247.34767	0.00382	0.00005590
5.00	54	- 247.38275	0.00386	0.00006694

表 5.3　α= 2, 不同样本大小 n 时, N 和 λ₀ 的最大似然估计

n	N	logL	$\lambda \cdot N^{\alpha}$	$\lambda(n+1)$
20	30	- 150.87079	0.00336	0.00037307
21	30	- 159.63300	0.00338	0.00030408
22	30	- 168.88558	0.00336	0.00023858
23	31	- 177.88173	0.00323	0.00021528
24	32	- 186.81963	0.00313	0.00019587
25	34	- 195.42690	0.00295	0.00020655
26	37	- 203.99810	0.00269	0.00023763
27	31	- 216.58287	0.00334	0.00005555
28	31	- 227.21138	0.00336	0.00003144
29	32	- 237.35307	0.00317	0.00002786
30	33	- 247.68058	0.00299	0.00002468

§ 5.2　非齐次泊松过程(NHPP)模型

累积故障数 N(t), 作为时间的变化量, 可以用一个 NHPP 过程模型加以描述。NHPP 模型在许多实际的应用中, 特别是硬件可靠性分析中, 为许多学者及工程人员所采用。

有不少软件可靠性模型也属于这一类。利用 NHPP 模型, 可以在一定前提条件下, 对许多软件可靠性模型中的参数予以推导。

最著名的 NHPP 模型应算 G-O 模型, 它于 1979 年由 Goel 和 Okumoto 提出, 之后又有许多人提出了不少类似的模型。下面我们将详细介绍 G-O 模型, 并以它为基础, 介绍利用 NHPP 模型怎样进行估测及参数估计的方法, 之后我们再对其它 NHPP 模型进行概述。

A. L. Goel 和 K. Okumoto 于 1979 年提出关于连续时间的非齐次泊松过程(Nonhomogeneous Poisson Process)模型, 简称为 NHPP 类的 G-O 模型。他们的工作, 对随后的软件可靠性模型的建立有着很大的影响, NHPP 类模型已成为软件可靠性模型的一个大类。

设 NHPP 过程具有均值函数 m(t), 强度函数 $\lambda(t)$, $t \geq 0$, 则对于 NHPP 中的未知参数, 可应用最大似然估计法予以估计。它的最大似然函数为:

$$L = \exp[-m(t_n)] \prod_{i=1}^{n} \lambda(t_i),$$

其中, n 是在区间[0, t]内观察到的故障数, 且 $t_1 < t_2 < \cdots < t_n$。

另外, 我们还可利用条件: $N(t_i) - N(t_{i-1}) = 1$, $N(0) = 0$, 再利用泊松分布

$$P_r\{N(t_i) = i, N(t_{i-1}) = i-1\}$$
$$= [m(t_i) - m(t_{i-1})]\exp[-m(t_i) + m(t_{i-1})]$$

导出另一种形式的最大似然函数为:

$$L = \exp[-m(t_n)]\prod_{i=1}^{n}[m(t_i) - m(t_{i-1})].$$

5.2.1 G-O 模型介绍

Goel-Okumoto 提出用 NHPP 描述的软件错误查出模型,亦可称为软件错误查出的指数类增长模型。在下面的讨论中,设 $N(t)$ 和 $m(t)$ 分别表示在区间 $[0, t]$ 内的累积错误数和期望错误数,则:

$$m(t) = m_E(t) = a(1 - e^{-bt}), \quad a > 0, \quad b > 0 \qquad (5.2.1)$$

其中,a 是最终查出的期望错误个数; b 是在时刻 t 每个错误的错误查出率。

查出的错误个数的增长曲线,即 $m_E(t)$ 的行为,如图 5.2 所示,呈指数分布。

图 5.2 关于 G-O 模型的一种典型的 $m(t)$ 和 $\lambda(t)$ 图形($a = 175, b = 0.05$)

G-O 模型的推导过程如下:

记 $N(t)$ 为到 t 时刻为止的累积错误个数,并设 $m(0) = 0, m(\infty) = a$, a 为一未知常数,而且,在 $(t, t + \Delta t)$ 中错误出现的个数与 $(a - m(t))$,即剩余错误数(在 t 时刻)成正比,即:

$$m(t + \Delta t) - m(t) = b \cdot (a - m(t)) \cdot \Delta t,$$

其中,b 为比例常数,令 $\Delta t \to 0$,则得:

$$m'(t) = ab - bm'(t).$$

利用边界条件解之,可得:

$$m(t) = a(1 - e^{-bt}), \quad a > 0, \quad b > 0.$$

它就是 G-O 模型的均值函数,即表达式(5.2.1)。

由(5.2.1)有:

$$m_E(t)/a = 1 - e^{-bt}$$

为指数分布,具均值 $1/b$,本模型的强度函数是:

$$\lambda(t) \equiv \lambda_E(t) = abe^{-bt} \tag{5.2.2}$$

表示在时刻 t 的错误查出率。

G-O 模型的有关统计量为:

$$\lim_{t\to\infty} P_r\{N(t) = n\} = \frac{a^n}{n!}e^{-a} \quad (n = 0, 1, 2, \cdots) \tag{5.2.3}$$

它也表示如果无限地对软件系统测试下去,查出的总错误数,也服从于具均值 a 的泊松分布。剩余在系统中的错误个数是:

$$N(t) \equiv N(\infty) - N(t) \tag{5.2.4}$$

则 N(t)的期望和方差是:

$$\begin{aligned} E\{N(t)\} &= m_E(t) = a - m_E(t) \\ &= Var\{N(t)\} = ae^{-bt} \end{aligned} \tag{5.2.5}$$

设在时刻 t,已查出 n_d 个错误。N(t)在给定 $N(t) = n_d$ 时的条件分布和它的期望是:

$$P_r\{N(t) = x \mid N(t) = n_d\} = \frac{[m_E(t)]^x}{x!}e^{-m_E(t)} \tag{5.2.6}$$

$$E\{N(t) \mid N(t) = n_d\} = m_E(t) \tag{5.2.7}$$

(5.2.6)式对于决定软件系统的投放时间,是很有用的。关于这一点,我们在第八章还要讨论。

考察随机变量序列$\{X_n, n = 1, 2, \cdots\}$,它表示故障间的间隔时间,则 $S_n = \sum_{k=1}^{n} X_k (n = 1, 2, \cdots)$ 表示第 n 个故障出现的时间。X_n 的条件可靠性函数,在给定 $S_{n-1} = s$ 的条件下(它即为软件的可靠度)是:

$$R_{X_n \mid S_{n-1}}(t \mid s) = \exp\{-a[e^{-bs} - e^{-b(t+s)}]\} \tag{5.2.8}$$

模型未知参数的估计:

假设在软件开发过程中,观察到 N 个故障出现的时间 $s = (S_1, S_2, \cdots, S_N)$ $(0 \le S_1 \le S_2 \le \cdots \le S_N)$,则 s 的联合概率密度函数(具均值函数 $m_E(t)$ 的 NHPP 的联合概率密度函数)

$$\begin{aligned} &f_{S_1, S_2, \cdots, S_N}(S_1, S_2, \cdots, S_N) \\ &= \prod_{k=1}^{N} abe^{-bS_k} \exp[-a(1 - e^{-bS_N})] \end{aligned} \tag{5.2.9}$$

是在给定 s 时,关于 a, b 的似然函数。由(5.2.9)的对数似然函数,可得下列方程:

$$\frac{N}{a} = 1 - e^{-bS_N},$$

$$\frac{N}{b} = \sum_{k=1}^{N} S_k + aS_N e^{-bS_N}.$$

解上述方程,即可得 a, b 的估计值 a, b.

下面假设累积软件故障数,即在时刻 $t_k (k = 1, 2, \cdots, N)$ 查出的累积软件错误数为 y_k,则关于 a 和 b 的似然函数就是:

$$P_r\{N(t_1) = y_1, N(t_2) = y_2, \cdots, N(t_N) = y_N\}$$

$$= \prod_{k=1}^{N} \frac{\{a(e^{-bt_{k-1}} - e^{-bt_k})\}^{y_k - y_{k-1}}}{(y_k - y_{k-1})!} \exp[-a(1 - e^{-bt_N})] \qquad (5.2.10)$$

其中, $t_0 = 0$, $y_0 = 0$, 由(5.2.10)的对数似然函数, 可得下列方程:

$$a(1 - e^{-bt_N}) = y_N,$$

$$at_N e^{-bt_N} = \sum_{k=1}^{N} \frac{(y_k - y_{k-1})(t_k e^{-bt_k} - t_{k-1} e^{-bt_{k-1}})}{(e^{-bt_{k-1}} - e^{-bt_k})}.$$

解上述方程, 即可得 a, b 的估计值 a, b。

5.2.2 G-O 模型的推广

1984 年, Yamada 和 Osaki 提出两种错误类型的软件可靠性模型。它与 G-O 模型的区别就在于: 将系统中的错误分为两种类型, 总共含有 $p_1 N_0$ 个 I 类错误和 $p_2 N_0$ 个 II 类错误。因此, 模型由两个各具相应均值函数的泊松过程来描述, 且这两个过程必须是互不相关的。在这种情况下, 这两个过程的和也必定是一个泊松过程。这样的两个过程所具有的均值分别为:

$$m_1(t) = p_1 N_0 (1 - e^{-\beta_1 t}),$$

$$m_2(t) = p_2 N_0 (1 - e^{-\beta_2 t}).$$

另外,

$$p_r\{N(t) = n\} = \frac{[m(t)]^n}{n!} \exp[-m(t)],$$

$$m(t) = N_0 \sum_{i=1}^{2} p_i (1 - e^{-\beta_i t}),$$

$$p_1 + p_2 = 1,$$

$$\lambda(t) \equiv \frac{dm(t)}{dt} = N_0 \sum_{i=1}^{2} p_i \beta e^{-\beta_i t}.$$

在时间 t, 每个错误(每单位时间)的错误查出率定义为:

$$d(t) = \lambda(t) / [N_0 - m(t)]$$

$$= \sum_{i=1}^{2} \frac{p_i e^{-\beta_i t}}{p_1 e^{-\beta_1 t} + p_2 e^{-\beta_2 t}} \quad i¤\beta.$$

它随时间增加单调下降, 并且有 $d(0) = \sum_{i=1}^{2} p_i \beta$, $d(\infty) = \beta$. 另外, 有:

$$N_r(t) = N_0 \sum_{i=1}^{2} p_i e^{-\beta_i t}.$$

给定最后一个故障出现的时间为 s, 则在区间 $(s, s+x]$ 中, 软件不出故障的概率为:

$$R(x \mid s) = \exp[-N_0 \sum_{i=1}^{2} p_i (e^{-\beta_i s} - e^{-\beta_i (s+x)})],$$

它就是具有参数 m(t) 的 NHPP 模型的软件可靠性。

Ohba 的增长模型在形式上与 Yamada 和 Osaki 的两种类型模型十分相似, 它不是将软件中的错误分类, 而是将软件中的错误按其出现的模块来分类。也即是说, 错误的出现具有的均值函数依赖于该错误所出现在其中的模块, 它具有如下的形式:

$$m_i(t) = N_{0i}(1 - e^{-\ i^t}).$$

这两个模型的实际区别仅在于参数估计方面。Ohba 将错误的出现按有错模块分组，于是得到一系列的数据集合：$\{n_{ij}\}$，并据此估计出$\{\ _j\}$和$\{N_{0j}\}$。

1985 年，Goel 提出了一般的 NHPP 模型。大多数故障间的时间模型和故障记数模型都假设：软件系统在测试期间都表现出故障率递减的模式。换言之，软件的质量是随着测试的进展，不断改进的。在实际情况中，大多数实际观察到的现象都是故障率首先上升，而后再下降。为了对这种上升/下降的故障率模式建立软件可靠性模型，Goel 提出 G-O 模型的下述一般形式：

$$p_r\{N(t) = y\} = \frac{[m(t)]^y}{y!}e^{-m(t)}, \quad y = 0, 1, 2, \cdots,$$

$$m(t) = a(1 - e^{-bt^c}),$$

其中，a 是最终要下降的错误的期望数，b 和 c 是反映测试质量的常数。显见，如 c= 1，则为 G-O 原始模型。

模型中故障率由下式给出：

$$\lambda(t) \equiv m'(t) = abce^{-bt^c}t^{c-1}.$$

图 5.3　关于 Goel 一般的 NHPP 模型的一种典型的 $m(t)$ 和 $\lambda(t)$ 图形($a= 500, b= 0.015, c= 1.5$)

1984 年，Ohba 提出 G-O 模型的推广——错误相关性模型。他假设：某些错误在其它一些错误被排除之前，不可能被查出来。因而，有：

$$m(t) = N_0\frac{1 - e^{-\ t}}{1 + \beta e^{-\ t}}.$$

它又可写成：

$$m(t) = N_0(1 - e^{-\ t})w(t)^r,$$

其中，

$$w(t) = \frac{1}{(1 + \beta e^{-\ t})^{1/r}},$$

r 是结构指数。

根据 Parr 的理论，$w(t)^r$ 是一个错误为“叶子错误”的概率，即叶子错误不再隐藏有任

何其它错误。因此,

$$m(t) = N_0 P_r\{\text{一个错误是叶子错误}\}(1 - e^{-\beta t})$$

或

$$m(t) = E[\text{叶子错误的个数}](1 - e^{-\beta t}).$$

Ohba 应用线性近似,得到:

$$P_r\{\text{第 } i \text{ 个错误是叶子错误}\} = r + (1 - r)\frac{i}{N_0},$$

其中,

$$r = \frac{\underline{\text{可观察到的错误数}}}{N_0},$$

而上面出现的 β 就定义为: $\beta = (1 - r)/r$.

最大似然函数为:

$$L = \sum_{i=1}^{k} n_i \ln N_0 \frac{1 - e^{-\beta t_i}}{1 + \beta e^{-\beta t_i}} - \frac{1 - e^{-\beta t_{i-1}}}{1 + \beta e^{-\beta t_{i-1}}}$$

$$- N_0 \frac{1 - e^{-\beta t_i}}{1 + \beta e^{-\beta t_i}} - \frac{1 - e^{-\beta t_{i-1}}}{1 + \beta e^{-\beta t_{i-1}}} .$$

于是得出下列方程:

$$N_0 = \sum_{i=1}^{k} n_i \frac{1 + \beta e^{-\hat{\beta} t_k}}{1 - e^{-\hat{\beta} t_k}}$$

$$N_0 t_k e^{-\hat{\beta} t_k} \frac{1 + \beta}{(1 + \beta e^{-\hat{\beta} t_k})^2}$$

$$= \sum_{i=1}^{k} n_i \frac{t_i e^{-\hat{\beta} t_i} - t_{i-1} e^{-\hat{\beta} t_i}}{e^{-\hat{\beta} t_{i-1}} - e^{-\hat{\beta} t_i}} + \frac{t_i}{1 + \beta^{\hat{\beta} t_i}} + \frac{t_{i-1}}{1 + \beta e^{-\hat{\beta} t_{i-1}}} .$$

5.2.3 其它的 NHPP 模型

1983 年, Yamada 和 Osaki 提出关于查错的 Γ-类增长模型。他们指出,查出错误数的增长曲线是 S-形的,即开始增长缓慢,然后快速增长,最后趋于饱和。

他们提出在时刻 t, S-形的均值函数:

$$m(t) = m_G(t) = a(1 - (1 + bt)e^{-bt}), \quad a > 0, \quad b > 0 \qquad (5.2.11)$$

的 NHPP 模型,其中 a 的意义与 G-O 模型相同, b 是在稳定状态下每个错误的错误查出率。因为:

$$\lim_{t\to\infty} \mu(t) \equiv \lim_{t\to\infty} \lim_{h\to 0} \frac{m_G(t + h) - m_G(t)}{(a - m_G(t))^h} = b,$$

其中, $\mu(t)$ 是在时刻 t 的每个错误的错误查出率。$m_G(t)$ 的行为示于图 5.4,以下将它称为 Y-O 模型。

由(5.2.11),有 $m_G(t)/a$ 变成具有参数 2 和标参 b 的 Γ-分布,强度函数是:

$$\lambda(t) \equiv \lambda_G(t) = ab^2 te^{-bt} \qquad (5.2.12)$$

关于 $m_E(t)$, $m_G(t)$ 间的关系,有:

图 5.4　$M_G(t)$的拟合曲线

$$\frac{dm_G(t)}{dt} = b(m_E(t) - m_G(t)),$$

$$\frac{dm_E(t)}{dt} = b(a - m_E(t)),$$

$$m_G(0) = m_E(0) = 0,$$

$$m_G(\infty) = m_E(\infty) = a.$$

关于模型的参数估计,估计方法与 G-O 模型相同: 如 N 个软件故障出现的时间 $s=(S_1, S_2, \cdots, S_N)(0\leq S_1\leq S_2\leq\cdots\leq S_N)$,则解下列方程,可得 a,b 的估计值 a,b:

$$\frac{N}{a} = 1 - (1 + bS_N)e^{-bS_N},$$

$$\frac{2N}{b} = \sum_{k=1}^{N} S_k + abS_N^2 e^{-bS_N}.$$

另一方面,如果到时刻 t_k,查出的错误累积数是 y_k,则可以估计 a,b 如下:

$$a(1 - (1 + bt_N)e^{-bt_N}) = y_N,$$

$$at_N^2 e^{-bt_N} = \sum_{k=1}^{N} \frac{(y_k - y_{k-1})(t_k^2 e^{-bt_k} - t_{k-1}^2 e^{-bt_{k-1}})}{[(1 + bt_{k-1})e^{-bt_{k-1}} - (1 + bt_k)e^{-bt_k}]},$$

$$t_0 = 0,$$

$$y_0 = 0.$$

Yamada, Ohba 和 Osaki 由一附加项 αt,对 G-O 模型进行了推广:

$$m(t) = N_0[1 - (1 + \alpha t)e^{-\alpha t}].$$

因此,　　$m(t) = N_0\Gamma(2, \beta)$.

最大似然函数为:

$$L = \sum_{i=1}^{k} \{n_i \ln[N_0((1 + \alpha t_{i-1})\exp(-\alpha t_{i-1}) - (1 + \alpha t_i)\exp(\alpha t_i))]$$

$$- N_0[(1 + \alpha t_{i-1})\exp(-\alpha t_{i-1}) - (1 + \alpha t_i)\exp(-\alpha t_i)]\}.$$

于是有下面的方程:

$$N_0 t_k^2 e^{-\hat{\alpha} t_k} = \sum_{j=1}^{k} n_i \frac{t_i^2\exp(-\hat{\alpha} t_i) - t_{i-1}^2\exp(-\hat{\alpha} t_{i-1})}{(1 + \hat{\alpha} t_{i-1})\exp(-\hat{\alpha} t_{i-1}) - (1 + \hat{\alpha} t_i)\exp(-\hat{\alpha} t_i)},$$

$$\sum_{i=1}^{k} n_i = N_0[1 - (1 + \hat{\ } t_k)e^{-\hat{\ } t_k}].$$

Yamada 和 Osaki 提出一个离散的 NHPP 模型，它具有均值函数(对应于第 i 类错误)：

$$m_i(s) = p_i N_0[1 - (1 - \quad_i)^s],$$

其中，s 是测试的次数。

进行参数估计的对数似然函数为：

$$L = \sum_{i=1}^{k} n_i \ln N_0 \sum_{j=1}^{2} p_j[(1 - \quad_j)^{s_i} - (1 - \quad_j)^{s_{i-1}}]$$

$$- N_0 \sum_{j=1}^{2} p_j[(1 - \quad_j)^{s_i} - (1 - \quad_j)^{s_{i-1}}] .$$

于是可以得出：

$$\sum_{i=1}^{k} n_i = N_0 \sum_{i=1}^{2} p_j[1 - (1 - \hat{\ }_j)^{s_k}],$$

$$N_0 s_k(1 - \hat{\ }_j)^{s_k - 1} = \sum_{i=1}^{k} \frac{s_i(1 - \hat{\ }_j)^{s_{i-1}} - s_{i-1}(1 - \hat{\ }_j)^{s_{i-1}-1}}{\sum_{i=1}^{2} p_1(1 - \hat{\ }_1)^{s_{i-1}} - (1 - \hat{\ }_1)^{s_i}} .$$

在 1975 年，Moranda 还发表了一个几何泊松模型，均值函数定义如下：

$$\Delta m(t_i) = \quad \beta^{i-1}, \quad \beta < 1.$$

模型使用固定的时间区间长度 θ 因此

$$t_i = i\theta,$$

$$\Delta m(t_i) = m(t_i) - m(t_{i-1}).$$

于是直接导出：

$$m(t_i) = \frac{1 - \beta}{1 - \beta}.$$

如果假定 $\beta < 1$，$\lim_{i \to \infty} m(t_i) = N_0$，则：

$$m(t_i) = N_0(1 - \beta^i),$$

$$\Delta m(t_i) = N_0 \frac{1 - \beta}{\beta} ¡¤ \beta^i,$$

$$E[N(t_i)] = m(t_i),$$

并且，因此有：

$$\Delta m(t_i) = \frac{1 - \beta}{\beta}\{N_0 - E[N(t_i)]\}.$$

模型因而可以看作是用 NHPP 公式表示的 JM 模型的一种变形。其对数似然函数为：

$$\sum_{i=1}^{k} \frac{n_i}{N_0 - N(t_i)} = \hat{\ } \sum_{i=1}^{k} \hat{\theta}^{\beta},$$

$$\sum_{i=1}^{k} n_i \ln \theta = \hat{\ } \sum_{i=1}^{k} \hat{\theta}^{\beta} \ln \theta[N_0 - N(t_i)].$$

注意，当所有的测试区间具有同样长度时，我们可以令 θ = 1，于是就得到 JM 模型，

上面最后一个方程将推出 0= 0。

1980 年, W. D. Brooks 和 R. W. Motley 提出了 IBM 泊松模型, 他们考虑到这样一个事实: 测试数据集仅覆盖系统的一部分。因此得到具有风险的错误个数的期望值是:

$$E(n_i \mathbin{©} N_r(t)) = f_i N_r(t),$$

其中, f_i 是被测试数据集合 i 覆盖的系统部分。支持这一想法的根据是:

在模块 j 中的错误个数是: l_j, 设 w_j 是:

$$w_j = l_j / \sum_{j=1}^{M} l_j,$$

则系数 f_i 定义为:

$$f_i = \sum_{j \in J_i} w_j / \sum_{j=1}^{M} w_j .$$

有两种因素使得模型的复杂性有所增加:

(1) 改正一个错误, 可能发生:

- 在过去某个时间被改正的错误重新被引入;
- 引入一个新的错误。

(2) 对一个错误的改正, 可能为其它到目前为止被别的错误掩盖的错误的暴露创造条件。

其它许多模型也考虑了这些因素, 但没有人象这里这样明白地提出这一点。

IBM 泊松模型用一个 β来描述上列因素, β表示一次改正行为或者(重新)引入一个错误、或者开辟一条包含有错误的新路径的概率。于是就导出简单的模型:

$$N_i = f_i[N_0 - \beta N(t_i)].$$

Ohba 和 Chou 对 G-O 模型中的"完全排错"假设进行了修正。他们认为: 对软件的排错过程, 如果认为它是"完全"的, 是不真实的, 因为在排错的过程中, 引入新错误的概率不等于 0, 在实际的排错过程中可能会引入新的错误。基于这一认识, 他们提出了一个不完全排错的 NHPP 模型。他们假设:

- 发现错误的概率与软件的剩余错误个数成正比;
- 这一比例(即错误查出率)相对于时间而言是一个常数, 以 b 表示, $0 < b < 1$;
- 排错时可能引入新的错误, 且引入新错的概率与被排除的错误数成正比;
- 这一比例(即错误引入率)相对于时间而言是一个常数, 以 β表示, $0 < \beta < 1$;
- 在排除新引入的错误时, 照样有可能再引入别的错误。于是有:

$$\begin{aligned} \frac{d}{dt}m(t) &= b[n(t) - m(t)] \\ \frac{d}{dt}n(t) &= \beta \frac{d}{dt}m(t) \\ m(0) &= 0 \\ n(0) &= a \end{aligned} \qquad (5.2.13)$$

其中, $n(t)$ 为到 t 时刻, 软件中总共引入的错误个数的期望, 其它的与 G-O 模型相同。解上述微分方程, 可得出:

$$m(t) = \frac{a}{1-\beta}[1 - e^{-(1-\beta)bt}] \tag{5.2.14}$$

$$n(t) = \frac{a}{1-\beta}[1 - \beta e^{-(1-\beta)bt}] \tag{5.2.15}$$

我们将它们称为 Ohba **三参数** NHPP **模型。它的三个参数** a, b, β**可以利用最大似然估计法来估计。最大似然函数为**:

$$L = \exp(-m(t_n)) \prod_{i=1}^{n} \lambda(t_i).$$

则它的对数似然函数即为:

$$\ln L = -m(t_n) + \sum_{i=1}^{n} \{(y_i - y_{i-1}) \ln [m(t_i) - m(t_{i-1})] - \ln[(y_i - y_{i-1})!]\}.$$

将 m(t) **的公式代入, 并对** lnL **关于** a, b, β**分别求偏导数, 并令结果等于** 0, **有**:

$$\frac{\ln L}{a} = \frac{\ln L}{b} = \frac{\ln L}{\beta} = 0.$$

经过化简, 最后得到最大似然估计方程组为:

$$(1 - B)a - y_n C = 0 \tag{5.2.16a}$$

$$AC + at_n B = 0 \tag{5.2.16b}$$

$$a[B(bt_n C + 1) - 1] + y_n C + bAC^2 = 0 \tag{5.2.16c}$$

其中,

$$A = \sum_{i=1}^{n} \frac{t_{i-1} e^{-(1-\beta)bt_{i-1}} - t_i e^{-(1-\beta)bt_i}}{e^{-(1-\beta)bt_{i-1}} - e^{-(1-\beta)bt_i}} (y_i - y_{i-1}),$$

$$B = e^{-(1-\beta)bt_n},$$

$$C = 1 - \beta.$$

同样, 对均值函数作如下变化:

$$M(t) = m(t) - t\lambda(t),$$

$$N(t) = n(t) - t\lambda(t),$$

可以得到 S-**形** Ohba **三参数** NHPP **模型**:

$$M(t) = \frac{a}{1-\beta}[1 - (1 + (1-\beta)bt)e^{-(1-\beta)bt}] \tag{5.2.17}$$

$$N(t) = \frac{a}{1-\beta}[1 - \beta(1 + (1-\beta)bt)e^{-(1-\beta)bt}] \tag{5.2.18}$$

类似于上面的讨论, 可以得出估计参数 a, b, β**的值的最大似然估计方程组为**:

$$y_n(1-\beta) + a((1 + (1-\beta)bt_n)e^{-(1-\beta)bt_n} - 1) = 0 \tag{5.2.19a}$$

$$[y_n + (1-\beta)^2 b^2 A'](1-\beta) + a[(1 + (1-\beta)bt_n(1 + (1-\beta)bt_n))e^{-(1-\beta)bt_n} - 1] = 0 \tag{5.2.19b}$$

$$A'(1-\beta) + at_n^2 e^{-(1-\beta)bt_n} = 0 \tag{5.2.19c}$$

其中,

$$A' = \sum_{i=1}^{n} \frac{(y_i - y_{i-1})[t_{i-1}^2 e^{-(1-\beta)bt_{i-1}} - t_i^2 e^{-(1-\beta)bt_i}]}{[1 + (1-\beta)bt_{i-1}]e^{-(1-\beta)bt_{i-1}} - [1 + (1-\beta)bt_i]e^{-(1-\beta)bt_i}}.$$

下面讨论怎样在 Ohba 三参数 NHPP 模型的基础上进行扩充,以引入可以考虑版本升级的概念。

软件工程技术对于软件产品质量的管理有许多方法,其中重要的一种就是版本的管理。软件产品的每一次升级不单是版本号的简单增加,它反映的是软件产品质量的一次明显改善和功能的显著扩充。因此,版本的一次升级,对于软件可靠性分析具有重要意义。

但是,令人遗憾的是,目前存在的许多软件可靠性估测模型并不考虑软件版本升级问题,而事实上软件工程技术又迫切要求反映这一影响软件质量的技术的软件可靠性分析模型。因此,我们在 Ohba 三参数 NHPP 模型的基础上作了扩充,使得到的考虑版本升级的三参数 NHPP 模型具有更广泛的应用意义。模型的假设如下:

1. 每两个错误间的间隔为统计独立的随机变量,记为 $t_1 < t_2 < \cdots < t_n$,于是 $N(t_1)$, $N(t_2) - N(t_1)$, $\cdots$, $N(t_n) - N(t_{n-1})$ 也是统计独立的;

2. 在软件中查出错误的概率与在该软件中的剩余错误个数成正比,其比例关系相对于时间而言是一常数,记为 $b(>0)$;

3. 允许排错过程是在排错时可引入新的错误的不完全排错过程,且引入的错误个数与被改正的错误个数成正比,且每个错误的引入率为常数,记为 $\beta(0<\beta<1$, β一定不等于 1,否则,“每改正一个错误都一定要引入新错”是不可能事件);

4. 记软件的版本号为 v,随着版本升级,查错率变为 $b^{1/v}$,引入率变为 β,v 为正实数。

关于以上假设,我们可作进一步解释如下:

在软件测试阶段,以至于投入正常使用以后的整个维护阶段,希望软件的性能得到不断的提高。软件版本的每次升级,它带来的问题是绝对不容忽视的。从软件可靠性分析的角度来看,即使假定负责维护的人员就是开发它的原班人马(这在国内往往如此),也包括原参加测试的排错人员,在随后的工作中,引入新错仍不可避免。从总体上讲,b 将不断增加,β将不断下降,因此 b 变为 $b^{1/v}$, β变为 β。

根据上述假设,有

$$\frac{d}{dt}m(t) = b^{1/v}[n(t) - m(t)],$$

$$\frac{d}{dt}n(t) = \beta\frac{d}{dt}m(t),$$

$$m(0) = 0,$$

$$n(0) = a.$$

其中,$m(t)$ 为在 t 时刻查出的错误数的期望; $n(t)$ 为在 t 时刻引入的错误数的期望; a 为软件中的初始错误个数; b 为查错率; v 为软件系统的版本号; β为错误引入率。

上式经过变换可得到

$$\begin{aligned} &m'' - b^{1/v}\beta m' + (b^{1/v})^2(\beta - 1)m + (b^{1/v})^2 a = 0 \\ &m(0) = 0 \\ &n(0) = a \end{aligned} \qquad (5.2.20)$$

由:

$$m'' + b^{1/v}\beta m' + (b^{1/v})^2(\beta - 1)m = 0,$$

可得:

$$\lambda^2 - b^{1/v}\beta\lambda + (b^{1/v})^2(\beta - 1) = 0.$$

解它可得:

$$\lambda = \frac{b^{1/v}\beta}{2} \pm \sqrt{\frac{(b^{1/v})^2\beta^2}{4} - (b^{1/v})^2(\beta - 1)} = \frac{b^{1/v}(\beta - 1)}{b^{1/v}}.$$

不妨设式(5.2.20)的解为:

$$m(t) = pe^{b^{1/v}t} + qe^{-b^{1/v}(1-\beta)t} - \frac{a}{1-\beta} \qquad (5.2.21)$$

其中,

$$m(0) = p + q + \frac{a}{1-\beta} = 0,$$

$$m'(0) = b^{1/v}p - b^{1/v}(1-\beta)q = b^{1/v}a,$$

整理后可得:

$$p - (1-\beta)q = a,$$

$$p + q = -\frac{a}{1-\beta}.$$

解得:

$$p = 0, \qquad q = -\frac{a}{1-\beta}.$$

将它们代回式(5.2.21),即得:

$$m(t) = \frac{a}{1-\beta}[1 - e^{-(1-\beta)b^{1/v}t}].$$

从而有:

$$n(t) = \frac{a}{1-\beta}[1 - \beta e^{-(1-\beta)b^{1/v}t}].$$

这两个式子即为我们的考虑到版本升级的三参数 NHPP 模型。在这两个式子中,如取 v=1,它们即变为 Ohba 的三参数 NHPP 模型;如令 v=1,β=0,它们就变为 G-O 模型。

考虑版本升级的三参数 NHPP 模型也含有三个参数 a, b, β 它们可以采用最大似然估计法来估计。

先讨论应用完全数据集合 NTDS 的参数估计方法。似然函数为:

$$L = \exp[-m(t_n)]\prod_{i=1}^{n}[m(t_i) - m(t_{i-1})].$$

则对数似然函数为:

$$\ln L = -m(t_n) + \sum_{i=1}^{n}\ln[m(t_i) - m(t_{i-1})].$$

将 $m(t_i)$ 代入,并求出极大似然方程组:

$$(1-B)a - nC = 0 \qquad (5.2.22a)$$

$$A_0C + nt_nB = 0 \qquad (5.2.22b)$$

$$a[B(b^{1/v}t_nC + 1) - 1] + nC + b^{1/v}C^2A_0 = 0 \qquad (5.2.22c)$$

其中,

$$A_0 = \sum_{i=1}^{n} \frac{t_{i-1}e^{-b^{1/v}t_{i-1}C} - t_i e^{-b^{1/v}t_i C}}{e^{-b^{1/v}t_{i-1}C} - e^{-b^{1/v}t_i C}},$$

$$B = e^{-b^{1/v}t_n C},$$

$$C = 1 - \beta.$$

并且, $a > 0$, $0 < b < 1$, $0 < \beta < 1$。应用 NTDS 数据, 解方程组(5.2.22), 即可得出 a, b, β的估计值, 示于表 5.4。

表 5.4　应用 NTDS 数据的估计结果

	a	b	β
开发阶段	29.37597341545070	7.43567326534399E-03	7.11175578380394E-02
测试 I	28.09755708979977	7.37652888101736E-03	0.12095571824672E-02
用户阶段	27.70078843428846	7.35031482131617E-03	0.13992752940660E-02
测试 II	24.06866252708613	6.94812379986813E-03	0.30374615366116E-02

下面我们再来讨论怎样对不完全数据进行模型的参数估计。

此时用于估计的对数似然函数为

$$\ln L = -m(t_p) + \sum_{i=1}^{p} \{(y(i) - y(i-1)) \ln[m(t_i) - m(t_{i-1})] - \ln[(y(i) - y(i-1))!]\},$$

从而可得极大似然估计方程组

$$(1 - B)a - y(p)C = 0 \qquad (5.2.23a)$$

$$t_p a B + A_1 C = 0 \qquad (5.2.23b)$$

$$(t_p b^{1/v} BC + B - 1)a + y(p)C + b^{1/v} A_1 C^2 = 0 \qquad (5.2.23c)$$

其中,

$$A_1 = \sum_{i=1}^{p} [y(i) - y(i-1)] \frac{t_{i-1}e^{-b^{1/v}t_{i-1}C} - t_i e^{-b^{1/v}t_i C}}{e^{-b^{1/v}t_{i-1}C} - e^{-b^{1/v}t_i C}},$$

$$B = e^{-b^{1/v}t_p C},$$

$$C = 1 - \beta.$$

利用表 2.5 中的不完全数据解式(5.2.23), 可估计出参数的值记为 $\theta = (a_0, b_0, \beta_0)$, 它们分别为:

$$a_0 = 83.3290987576906392; \quad b_0 = 0.0326733480075140411;$$

$$\beta_0 = 0.00390955556213950627.$$

鉴于考虑版本升级的三参数 NHPP 模型的独有特性, 它在国内应有广阔的潜在应用前景, 因为它适用于任何一种类型的软件系统。从实践中可看出, 在软件系统测试的后期, 软件错误出现得越来越稀少(这正反映了软件质量的不断提高过程), 在应用 EM 算法时, 假设数据集合中的最后一个时间间隔内的错误增量为 1 是合理的。我们欢迎有更多用户以不同的数据来检验我们的工作。

5.2.4 应用实例

我们在这里举出两个例子，一个是 Goel-Okumoto 的例子，一个是 Yamada-Osaki 的例子，以说明 NHPP 模型的应用情况。

例 1

Goel 和 Okumoto 使用数据见表 5.5，运用他们自己的模型，对它们进行了分析。该数据来自 Jelinski 和 Moranda(1972)，来源是美国海军舰队计算机程序设计中心(U. S. Navy Fleet Computer Programming Center)，包括的是构成海军战术数据系统(Naval Tactical Data System—NTDS)核心的、在开发复杂的、多机实时系统的软件过程中收集的错误数据。NTDS 软件由 38 个不同的模块组成，而每个模块都要经历三个阶段：制造(开发)阶段、测试阶段、用户阶段。数据以“问题报告”(trouble reports)为基础，它们来自于那些较大的模块，用 A-模块表示。故障间隔时间(以天为单位)以及其它附加的信息列于表 5.5 中。在制造阶段查出 26 个错误，在测试阶段又查出 5 个错误，最后一个错误于 1971 年元月 4 日查出。然后，于用户阶段查出一个错误(1971 年 9 月 20 日)，随后的测试阶段又查出两个错误(分别在 1971 年的 10 月 5 日和 11 月 10 日)。利用 n= 26，解得 a 和 b 的最大似然估计为：

$$a = 33.99, \quad b = 0.00579.$$

表 5.5 NTDS 数据

	错误数 n	错误间隔时间 x_k(日)	累计时间 $S_n = \sum x_k$(日)
制造阶段核查	1	9	9
	2	12	21
	3	11	32
	4	4	36
	5	7	43
	6	2	45
	7	5	50
	8	8	58
	9	5	63
	10	7	70
	11	1	71
	12	6	77
	13	1	78
	14	9	87
	15	4	91
	16	1	92
	17	3	95
	18	3	98
	19	6	104
	20	1	105
	21	11	116
	22	33	149
	23	7	156
	24	91	247
	25	2	249
	26	1	250

测试阶段	27 28 29 30 31	87 47 12 9 135	337 384 396 405 540
用户阶段	32		798
测试阶段	33 34		814 849

注：本表见 Goel-Okumoto, IEEE Trans. Reliab. Vol. R-28, P. 208.

图 5.5 给出这时的实际情况以及置信度为 90% 时的置信区间，并示出置信区间的宽度相当大，这说明对 a 和 b 的估计还有相当大的不确定性。

图 5.5　G-O 模型的估算结果

例 2

Yamada-Osaki 将他们的模型应用于参考文献[B10]中的数据 DS1，解最大似然方程组可得：

$$a = 3266.4,$$

$$b_1 = 0.1467, \quad b_2 = 0.0953.$$

于是估计的均值函数为：

$$\hat{m}_p(t) = 3266.4[(0.9)(1 - e^{-0.1467t}) + (0.1)(1 - e^{-0.0953t})].$$

这里他们假设 $P_1 = 0.9$, $P_2 = 0.1$.

图 5.6 中给出 $\hat{m}(t)$ 的图形以及置信水平为 90% 时的置信区间。

5.2.5　应用 EM 算法于 NHPP 模型的参数调整

为了直接比较不同的拟合结果，我们定义拟合率的数值如下：

定义 5.1　在使用单个或多个软件可靠性估测模型之后，一个故障数据集合的拟合率值：

图 5.6　关于数据 DS1 的估计的$\hat{m}(t)$的图形以及置信度为 90% 时的置信区间

$$F_r = \frac{1}{\dfrac{\sum_{i=1}^{n}\{[f(i) - y(i)]^2 \times [t(i) - t(i-1)]\}}{n}}.$$

其中, $f(i)$是使用单个或多个软件可靠性模型, 在第 i 个数据点上产生的拟合值。故障数据集合以及拟合结果都能直观地用图形表示。关于我们使用的图形, 先给出下面的注解:

注解 1. 图中纵轴表示累积故障数 $y(i)$或累积故障集合的拟合值 $f(i)$, 水平轴表示时间 $t(i)$, 因此图中的点都具有坐标对$(t(i), y(i))$或$(t(i), f(i))$。

注解 2. 图中光滑的或分段光滑的曲线表示拟合结果, 而不光滑的阶梯曲线表示原始的累积故障数据。

注解 3. 在每个图中我们都给出拟合结果的拟合率值。如果结果是由分段拟合过程给出, 则 F_{r1}为第一段分段拟合结果的拟合率值, F_{r2}为第二段的拟合率值, 等等。

先介绍 EM 算法如下。

Dempster, Laird 和 Rubin 分析综合了大量处理不完全数据的方法, 于 1977 年提出 EM 算法。基本想法是将 EM 算法分为 E 步和 M 步: 在 E 步用不完全数据估计出完全数据, 在 M 步则利用极大似然估计法于 E 步估计出的完全数据, 并反复迭代, 以改进估计结果, 迭代过程一直要进行到获得稳定的参数值。

设 θ为待估参数向量, $S = \{S_1, S_2, \cdots, S_n\}$, 集合 S 中的 S_r 表示第 r 个软件错误发生的时间$(r = 1, 2, \cdots, n)$, 并且:

$$0 \leq S_1 \leq S_2 \leq \cdots \leq S_n < +\infty.$$

设 $Y = \{y(0), y(1), \cdots, y(p)\}$, $y(i)$表示直到时刻 $t(i)$, $(i = 0, 1, 2, \cdots, p)$为止, 所查出的累积错误数, 显然, Y 是一个不完全数据集合。据此可得出联合概率密度函数:

$$f(S; \theta) = \exp[-m(S_n; \theta)] \times \prod_{i=1}^{n} \lambda(S_i; \theta),$$

注意: $Q(\theta, \theta') = E[\ln f(s; \theta), \theta']$, 并设$\dfrac{Q}{\theta} = 0$, 则得:

$$\sum_{i=1}^{n}\frac{E[\ln\lambda(s_i;\theta)\mid y_n,\theta']}{\theta}-\frac{[m(S_n;\theta)\mid y_n,\theta']}{\theta}=0 \qquad (5.2.24)$$

这里的关键问题是如何得出条件概率密度函数 $f_r(S\mid y_n,\theta')$，有一个重要的关系：

$$N(s)<r \text{ if } S_r>s,$$

并要求下列条件为真：

$$(t(k-1)<s\le t(k))\wedge(q=\min(y(k),r-1))\wedge(y(k-1)\le r-1),(1\le k\le p).$$

这里我们选用 Ohba 和 Chou 的 NHPP 模型作为软件可靠性估测模型，并称它为 Ohba-Chou 三参数 NHPP 模型：

$$m(t)=\frac{a}{1-\beta}[1-\exp(-(1-\beta)bt)],$$

$$n(t)=\frac{a}{1-\beta}[1-\beta\exp(-(1-\beta)bt)],$$

$$\lambda(t)=m'(t)=ab\exp[-(1-\beta)bt],$$

其中，$m(t)$ 是到时刻 t 为止查出的错误个数，a 是程序中的初始错误个数，b 是满足条件 $b>0$ 的错误查出率，$n(t)$ 是到时刻 t 为止引入到程序中的错误个数，β 是满足条件 $0\le\beta<1.0$ 的错误引入率，$\lambda(t)$ 是 $m(t)$ 的强度函数。

利用方程(5.2.24)和 Ohba-Chou 三参数 NHPP 模型，EM 算法的参数调整方程(EMPAE)可以推出：

$$\sum_{i=1}^{n}\frac{1}{a}-\frac{1}{1-\hat{\beta}}(1-E[\exp(-(1-\beta)bS_n)\mid y_n,\theta'])=0,$$

$$\sum_{i=1}^{n}\frac{1}{\hat{b}}-(1-\beta)\,¡¤E(S_i\mid y_n,\theta']+\frac{a}{1-\hat{\beta}}¡¤\frac{[\exp(-(1-\beta)bS_n)\mid y_n,\theta']}{\hat{b}}=0,$$

$$\sum_{i=1}^{n}b\beta¡¤E[s_i,\mid y_n,\theta']-\frac{b\beta}{1-\hat{\beta}}(1-E[\exp(-(1-\beta)bS_n)\mid y_n,\theta'])$$

$$-\frac{a}{1-\hat{\beta}}¡¤\frac{[\exp(-(1-\hat{\beta})bS_n)\mid y_n,\theta']}{\hat{\beta}}=0 \qquad (5.2.25)$$

通过解最大似然方程和 EMPAE 方程(5.2.25)，并使用表 2.5 和表 2.6 中的数据，我们可以估计出 Ohba-Chou 三参数 NHPP 模型的参数值。经过 EM 算法调整之后的结果示于图 5.7 和图 5.8 中。从图中我们可以看出，拟合曲线并不能完全适合 Ohba-Chou 三参数模型的假设，并且 EM 算法的调整作用不仅是有限的，而且对于不同的数据集合的调整结果也不同。前面已提到节假日包括在数据集合 Martini.d 中，所以使用 Martini.d 要比使用 D.inc 得到的 EM 算法的调整结果更明显。也就是说，数据中掺杂的“水份”越多，应用 EM 算法的调整效果就越显著。

应用 EM 算法的最大问题是计算量的增加，特别在调整方程的数值收敛过程中更是如此。这是在应用 EM 算法时的一个普遍问题。我们认为：为求得更好的估测结果，计算量的适当增加是值得的。而且，我们也采取了一些措施，减缓这一矛盾的尖锐性。一方面我们要求使用的计算机的计算速度要快，并配有浮点运算器；另一方面，我们尽量采用了收敛速度快的算法。此外，一种可供选择的行之有效的方法就是，在收敛精度和计算量之间作出适当的折衷；要求的收敛精度越高，计算量越大。

图 5.7　应用 EM 算法于 D.inc

图 5.8　应用 EM 算法于 Martini.d

下面我们以 G-O 模型为例,详细讨论 EM 算法的调整方程的推导过程。由 NHPP 过程的定义有:

$$P_r\{N(t) = n\} = \frac{[m(t)]^n}{n!} e^{-m(t)}.$$

若 $s \leq t$ 有:

$$P_r\{N(t) - N(s) = n\} = \frac{[m(t) - m(s)]^n}{n!} e^{-[m(t)-m(s)]},$$

其中,$N(t)$,$m(t)$ 分别表示软件在时间区间 $(0, t)$ 内的累积错误数和期望错误数,设强度

函数为 $\lambda(t)$，则：

$$m(t) = \int_0^t \lambda(t)\,dt$$

对于不完全数据，其最大似然函数为：

$$P_r\{N(t_1) = y_1, \cdots, N(t_p) = y_p\}$$
$$= \exp[-m(t_p)] \prod_{i=1}^{p} \frac{[m(t_i) - m(t_{i-1})]^{y_i - y_{i-1}}}{(y_i - y_{i-1})!}, \quad \begin{matrix} t_0 = 0 \\ y_0 = 0 \end{matrix} \qquad (5.2.26)$$

Dempster 等人于 1979 年提出用 EM 算法来处理不完全数据。设 X，Y 是两个样，且从 X 到 Y 存在一个多对多的映射。设 x，y 分别是 X，Y 中的样本，且 $y = Y(x)$，称 X 为完全数据，Y 为不完全数据。EM 算法提供了在观测到不完全数据 y 的情形下，求参数的极大似然估计的一种迭代法。

如果我们通过(5.2.26)式根据不完全数据估计出 NHPP 类模型的参数，并将它们作为初值，再通过 EM 算法迭代，可望提高参数估计的精度。下面讨论具体用 EM 算法调整 NHPP 类模型参数的过程。

设 $S = (S_1, S_2, \cdots, S_n)$，其中 S_r 是第 r 个软件错误发生的时间$(r = 1, 2, \cdots, n)$，且 $0 \leq S_1 \leq S_2 \leq \cdots \leq S_n < +\infty$。$Y = (y_0, y_1, y_2, \cdots, y_p)$，其中，$y_i$ 为到时刻 t_i 时查出的累积错误数$(i = 0, 1, 2, \cdots, p)$，且 $\forall i (i \in \{1, 2, \cdots, p\} \rightarrow y_i - y_{i-1} > 1)$，$t_0 = 0$，$y_0 = 0$，$y_p = n$，则关于 S 的联合概率密度为：

$$f(S; \theta) = L = \exp[-m(S_n; \theta)] \prod_{i=1}^{n} \lambda(S; \theta),$$

其中 θ 是待测参数，可为向量。

记：
$$Q(\theta, \theta') = E[\ln f(S; \theta) \mid Y, \theta']$$
$$= E\left[\sum_{i=1}^{n} \ln \lambda(S_i; \theta) - m(S_n; \theta) \mid Y, \theta'\right]$$
$$= \sum_{i=1}^{n} E[\ln \lambda(S_i; \theta) \mid Y, \theta'] - E[m(S_n; \theta) \mid Y, \theta'],$$

令 $\frac{Q}{\theta} = 0$ 得：

$$\sum_{i=1}^{n} \frac{E[\ln \lambda(S_i; \theta) \mid Y, \theta']}{\theta} - \frac{E[m(S_n; \theta) \mid Y, \theta']}{\theta} = 0 \qquad (5.2.27)$$

显然当(5.2.27)式成立时，$Q(\theta, \theta')$ 达到最大。

为了求解方程组(5.2.27)，需求得 $E[\ln \lambda(S_i; \theta) \mid Y, \theta]$ 及 $E[m(S_n; \theta) \mid Y, \theta']$ 关于 θ 的表达式，而根据随机变量函数的数学期望求解公式知关键在于要求得 $S_r (r = 1, 2, \cdots, n)$ 的条件概率密度 $f_r(s \mid Y, \theta')$。

有一个重要的关系式：

$N(s) < r$ 的充要条件是 $S_r > s$.

因此：

$$P_r\{S_r > s, N(t_1) = y_1, N(t_2) = y_2, \cdots, N(t_p) = y_p\} \quad (s > 0)$$

$$= P_r\{N(s) < r, N(t_1) = y_1, N(t_2) = y_2, \cdots, N(t_p) = y_p\}$$

$$= P_r\{N(s) = \alpha, (\alpha = 0 \vee 1 \vee 2 \vee \cdots \vee (r-1)), N(t_1) = y_1, \cdots, N(t_p) = y_p\}$$

$$= \sum_{\alpha=0}^{r-1} P_r\{N(s) = \alpha, N(t_1) = y_1, N(t_2) = y_2, \cdots, N(t_p) = y_p\},$$

当条件 C:

$$(t_{k-1} < s \leq t_k) \wedge (q = \min(y_k, r-1) \wedge y_{k-1} \leq r-1) \quad (1 \leq k \leq p)$$

成立时,我们有:

$$P_r\{S_r > s, N(t_1) = y_1, N(t_2) = y_2, \cdots, N(t_p) = y_p\}$$

$$= \sum_{\alpha=y_{k-1}}^{q} P_r\{N(t_i) - N(t_{i-1}) = y_i - y_{i-1}, i = 1, 2, \cdots, k-1,$$

$$N(s) - N(t_{k-1}) = \alpha - y_{k-1}, N(t_k) - N(s) = y_k - \alpha,$$

$$N(t_j) - N(t_{j-1}) = y_j - y_{j-1}, j = k+1, \cdots, p\}$$

$$= \sum_{\alpha=y_{k-1}}^{q} \prod_{i=1}^{k-1} P_r\{N(t_i) - N(t_{i-1}) = y_i - y_{i-1}\} \cdot P_r\{N(s) - N(t_{k-1}) = \alpha - y_{k-1}\}$$

$$\cdot P_r\{N(t_k) - N(s) = y_k - \alpha\} \cdot \prod_{j=k+1}^{p} P_r\{N(t_j) - N(t_{j-1}) = y_j - y_{j-1}\}.$$

$$= \sum_{\alpha=y_{n-1}}^{q} \prod_{i=1}^{k-1} \frac{[m(t_i;\theta) - m(t_{i-1};\theta)]^{y_i - y_{i-1}}}{(y_i - y_{i-1})!} \cdot \prod_{j=k+1}^{p} \frac{[m(t_j;\theta) - m(t_{j-1};\theta)]^{y_j - y_{j-1}}}{(y_j - y_{j-1})!}$$

$$\cdot \frac{[m(s;\theta) - m(t_{k-1};\theta)]^{\alpha - y_{k-1}}}{(\alpha - y_{k-1})!} \cdot \frac{[m(t_k;\theta) - m(s;\theta)]^{y_k - \alpha}}{(y_k - \alpha)!} \cdot \exp[-m(t_p;\theta)]$$

则在条件 C **的保证之下,我们可以得到(在上式与式**(5.2.26)**的前提之下)**:

$$P_r\{S_r > s \mid Y\}$$

$$= \frac{P_r\{S_r > s, N(t_1) = y_1, \cdots, N(t_p) = y_p\}}{P_r\{N(t_1) = y_1, \cdots, N(t_p) = y_p\}}$$

$$= \sum_{\alpha=y_{k-1}}^{q} \frac{y_k - y_{k-1}}{y_k - \alpha} \cdot \frac{[m(s;\theta) - m(t_{k-1};\theta)]^{\alpha - y_{k-1}} \cdot [m(t_k;\theta) - m(s;\theta)]^{y_k - \alpha}}{[m(t_k;\theta) - m(t_{k-1};\theta)]^{y_k - y_{k-1}}}$$

(5.2.28)

由式(5.2.28)**得** S_r **的条件分布函数** $F_r(s \mid Y, \theta)$**为**:

$$F_r(s \mid Y, \theta) = 1 - P_r\{S_r > s \mid Y, \theta\}$$

$$= 1 - \sum_{\alpha=y_{k-1}}^{q} \frac{y_k - y_{k-1}}{y_k - \alpha} \cdot \frac{[m(s;\theta) - m(t_{k-1};\theta)]^{\alpha - y_{k-1}} \cdot [m(t_k;\theta) - m(s;\theta)]^{y_k - \alpha}}{[m(t_k;\theta) - m(t_{k-1};\theta)]^{y_k - y_{k-1}}}.$$

则有:

$$f_r(s \mid Y, \theta) = \frac{dF_r(s \mid Y, \theta)}{ds}$$

$$= \sum_{\alpha=y_{k-1}}^{q} \frac{y_k - y_{k-1}}{y_k - \alpha} \cdot \frac{\lambda(s;\theta)}{[m(t_k;\theta) - m(t_{k-1};\theta)]^{y_k - y_{k-1}}}$$

$$\cdot (y_k - \alpha)[m(s;\theta) - m(t_{k-1};\theta)]^{\alpha - y_{k-1}} \cdot [m(t_k;\theta) - m(s;\theta)]^{y_k - \alpha - 1}$$

$$- (\alpha - y_{k-1})[m(s;\theta) - m(t_{k-1};\theta)]^{\alpha - y_{k-1} - 1} \cdot [m(t_k;\theta) - m(s;\theta)]^{y_k - \alpha} \quad (5.2.29)$$

当条件 C **不满足时**, $P_r\{S_r > s \mid Y, \theta\} = 0$, **因此**, $F_r(s \mid Y, \theta) = 1$。

下面，我们以 G-O 模型为例，根据不完全数据 D.inc，具体说明上面讨论的 EM 算法对参数的调整作用。G-O 模型的强度函数为：

$$\lambda(t) = m'(t) = abe^{-bt}$$

将 G-O 模型的公式以及 $\lambda(t)$ 代入(5.2.27)式得 G-O 模型的参数 a，b 的 EM 迭代方程为：

$$\begin{aligned} &\frac{n}{a} - 1 + E(e^{-bs_n} \mid Y_t, a', b') = 0 \\ &\frac{n}{b} - \sum_{r=1}^{n} E(S_r \mid Y_t, a', b') + a \cdot \frac{dE(e^{-bs_n} \mid Y_t, a', b')}{db} = 0 \end{aligned} \tag{5.2.30}$$

其中，

$$E(e^{-bs_n} \mid Y_t, a', b') = \int_0^{+\infty} e^{-bs} f_n(s \mid Y_t, a', b')\,ds,$$

$$E(S_r \mid Y_t, a', b') = \int_0^{+\infty} s f_r(s \mid Y_t, a', b')\,ds, \quad (r = 1, 2, \cdots, n).$$

一般软件测试的后期出错情况会越来越稳定，因此常有 $y_{p-1} = n-1$ 成立。如果 $y_{p-1} = n-1 < y_p = n$，则有：$t_{p-1} < S_n \le t_p$。当 $t_{p-1} < s \le t_p$ 时，由式(5.2.29)得：

$$\begin{aligned} & f_n(s \mid Y_t, a', b') \\ &= \sum_{\alpha=y_{p-1}}^{n-1} \binom{y_p - y_{p-1}}{y_p - \alpha} \frac{\lambda(s; a', b')}{[m(t_p; a', b') - m(t_{p-1}; a', b')]^{y_p - y_{p-1}}} \\ &\quad \cdot (y_p - \alpha)[m(s; a', b') - m(t_{p-1}; a', b')]^{\alpha - y_{p-1}} \\ &\quad \cdot [m(t_p; a', b') - m(s; a', b')]^{y_p - \alpha - 1} \\ &\quad - (\alpha - y_{p-1})[m(s; a', b') - m(t_{p-1}; a', b')]^{\alpha - y_{p-1} - 1} \\ &\quad \cdot [m(t_p; a', b') - m(s; a', b')]^{y_p - \alpha} \\ &= \frac{\lambda(s; a', b')}{m(t_p; a', b') - m(t_{p-1}; a', b')} \\ &= \frac{b e^{-bs}}{e^{-bt_{p-1}} - e^{-bt_p}} \end{aligned} \tag{5.2.31}$$

则有：

$$\begin{aligned} & E(e^{-bs_n} \mid Y_t, a', b') \\ &= \int_{t_{p-1}}^{t_p} e^{-bs} \cdot f_n(s \mid Y_t, a', b')\,ds \\ &= \int_{t_{p-1}}^{t_p} e^{-bs} \cdot \frac{b e^{-bs}}{e^{-bt_{p-1}} - e^{-bt_p}}\,ds \\ &= \frac{b[e^{-(b+b')t_p} - e^{-(b+b')t_{p-1}}]}{(b+b')(e^{-bt_p} - e^{-bt_{p-1}})} \end{aligned} \tag{5.2.32}$$

若 $y_{q-1} < r \le y_q$，$q = 1, 2, \cdots, p$，$r = 1, 2, \cdots, n$，则有：

$$t_{q-1} < s_r \le t_q.$$

因此，有：

$$E(S_r \mid Y_t, a', b') = \int_{t_{q-1}}^{t_q} s \cdot f_r(s \mid Y_t, a', b')\,ds = e_r \tag{5.2.33}$$

当 Y, a ¦ b 给定时, e_r 为一确定值。

将(5.2.31), (5.2.32) 和(5.2.33) 式代入(5.2.30) 式, 即得 $y_{p-1} = n - 1$, 这时, 参数 a, b 的 EM 迭代方程组为:

$$\frac{n}{a} - 1 + \frac{b\left[e^{-(b+b)t_p} - e^{-(b+b)t_{p-1}}\right]}{(b+b)(e^{-bt_p} - e^{-bt_{p-1}})} = 0,$$

$$\frac{n}{b} - \sum_{r=1}^{n} e_r + \frac{ab'}{(b+b)^2(e^{-bt_{p-1}} - e^{-bt_p})}\{(b+b)[t_p e^{-(b+b)t_p} - t_{p-1}e^{-(b+b)t_{p-1}}] + e^{-(b+b)t_p} - e^{-(b+b)t_{p-1}}\} = 0.$$

利用不完全数据 D. inc, 可得参数 a, b 的最大似然估计值为:

$$a_0 = 89.94144, \quad b_0 = 0.021777943.$$

以它们作为初值, 解迭代方程组, 并取控制精度为 0.0001, 可得出 a, b 的稳定值为:

$$a = 94.88520, \quad b = 0.019639589.$$

5.2.6 NHPP 模型拟合质量的改进

由于 EM 算法调整作用的限制, 虽然我们力图通过提高控制精度以提高估计结果的精度, 但其效果依然有限。为了改进拟合结果, 我们对完全数据和不完全数据都采用分段拟合法。

根据实际情况, 对同一故障数据集合采用分段拟合时, 可有许多不同的作法。如: 同一划分多个模型, 多种划分多个模型, 以及多种划分单一模型等等。

1. 同一划分多个模型

Martini 等人采用同一划分两个软件可靠性模型的方法, 分析了故障数据 Martini. d。这两个模型是指数模型和 S 形模型。我们计算出了拟合率的值并列于表 5.6, 其中全拟合率值是指由分段拟合曲线拼起来的拟合曲线的拟合率值。显然对于同一个故障数据集合, 应用不同的软件可靠性模型所得的结果是不同的。在表 5.6～表 5.9 中, 我们用符号 I (i_1, i_2) 表示划分方案。例如用 I (0, 30) 表示的一个拟合段是从 $i_1 = 0$ 开始, 而于 $i_2 = 30$ 处结束。

表 5.6 对于 Martini. d 的拟合值

	指数模型			S-形模型		
	a	b	F_r	a	b	F_r
I(0, 30)	1569	0.0070	0.3560	374	0.10	0.2519
I(31, 42)	2030	0.0023	0.6864	73	0.23	0.7436
I(43, 71)	154	0.0313	0.3315	117	0.11	0.3271
全拟合率值			0.3615			0.2961

2. S 形三参数 NHPP 模型

在本文工作中, 要用到 S 形三参数 NHPP 模型作为软件可靠性模型。

通过仔细的考察,发现在采用分段拟合的过程中,所划分的每一段累积故障数据都呈S形。为此作如下变换:

$$M(t) = m(t) - t\,m'(t) \text{ 和 } N(t) = n(t) - t\,n'(t),$$

于是得到S形三参数NHPP模型:

$$M(t) = \frac{a}{1-\beta}[1 - (1 + (1-\beta)bt)\exp(-(1-\beta)bt)],$$

$$N(t) = \frac{a}{1-\beta}[1 - \beta(1 + (1-\beta)bt)\exp(-(1-\beta)bt)],$$

其中$M(t)$是具有S形增长曲线的NHPP模型的均值函数,$N(t)$是到时刻t为止引入到程序中的错误个数,$m(t)$,$n(t)$,a,b,β的意义同Ohba-Chou三参数模型。在表5.7(a)～(c)中,我们列出关于完全数据集合NTDS的分析结果。

表5.7 (a) 关于NTDS数据的分析结果

	应用的NHPP模型	a	b	β	F_r
I(0,24)	S形三参数	17	0.03473	0.2853	0.3473
I(25,34)	S形三参数	8	0.01028	0.2067	0.1199
全拟合率值					0.1978

表5.7 (b) 关于NTDS数据的分析结果

	应用的NHPP模型	a	b	β	F_r
I(0,21)	S形三参数	31	0.01969	0.2884	0.5797
I(22,28)	S形三参数	4	0.02498	0.4516	0.2024
I(29,34)	S形三参数	5	0.01060	0.2911	0.1024
全拟合率值					0.2085

表5.7 (c) 关于NTDS数据的分析结果

	应用的NHPP模型	a	b	β	F_r
I(0,21)	S形三参数	31	0.01969	0.2884	0.5797
I(22,28)	S形三参数	4	0.02498	0.4516	0.2024
I(29,34)	Ohba-Chou三参数	9	0.00195	0.3373	0.1095
全拟合率值					0.2184

从表5.7(a)～(b)能看出F_r的值随着划分段数的增加而增加。另一方面,在分析过程中我们使用不同的软件可靠性估测模型,产生出不同的拟合结果。我们的目的在于改进拟合质量,其标志就是F_r的值。因此F_r的值,特别是全拟合率的值越大,拟合质量越好。

3. 多种划分多个模型

在表5.8(a)和5.8(b)中,分析结果也是用同一划分多个模型的方法得到的,但它们是关于不完全数据集合D.inc的结果。

从表5.7(a)、(b)中的结果可知,要由它来比较不同的划分方案是困难的,但仍可以

通过计算全拟合率值以决定拟合结果的取舍。

4. 多种划分单一模型

在表 5.9(a)和表 5.9(b)中,我们列出关于不完全数据集合 Martini.d 的分析结果,它们是用单一的软件可靠性模型——S 形三参数 NHPP 模型获得的。在表 5.9(a)中,我们将累积故障数据划分为 4 段,在表 5.9(b)中则分为 5 段。其结果是很明显的:划分段数越多,全拟合率值越大,拟合质量也就越好。图 5.9 示出表 5.9(b)中的结果。

表 5.8 (a) 关于 D.inc 的分析结果

	应用的 NHPP 模型	a	b	β	F_r
I(0,10)	Yamada S 形	31	0.1866	-	0.3977
I(11,29)	Ohba-Chou 三参数	6	0.5506	0.8865	0.3455
全拟合率值					0.3673

表 5.8 (b) 关于 D.inc 的分析结果

	应用的 NHPP 模型	a	b	β	F_r
I(0,10)	S 形三参数	13	0.5126	0.5736	0.4017
I(11,17)	S 形三参数	26	0.2159	0.2560	0.3921
I(18,29)	Ohba-Chou 三参数	19	0.1443	0.2523	0.4500
全拟合率值					0.4241

表 5.9 (a) 关于数据 Martini.d 的分析结果

	应用的 NHPP 模型	a	b	β	F_r
I(0,17)	S 形三参数	61	0.0645	0.6899	0.3193
I(18,30)	S 形三参数	69	0.0549	0.5929	0.4865
I(31,49)	S 形三参数	58	0.0296	0.4207	0.6152
I(50,71)	S 形三参数	44	0.0268	0.4678	0.4312
全拟合率值					0.4749

表 5.9 (b) 关于数据 Martini.d 的分析结果

	应用的 NHPP 模型	a	b	β	F_r
I(0,14)	S 形三参数	82	0.0503	0.5038	0.5986
I(15,30)	S 形三参数	53	0.0659	0.7433	0.4849
I(31,39)	S 形三参数	23	0.0724	0.7212	0.6654
I(40,55)	S 形三参数	24	0.0528	0.6954	0.3696
I(56,71)	S 形三参数	40	0.0383	0.2789	0.6096
全拟合率值					0.4969

从这两小节讨论的结果可以看出,EM 算法的调整作用是明显的,但也是有限的。采用分段拟合技术对于改进拟合质量是有效的。另外,我们定义的拟合率值可以用于判定是否采纳分段拟合过程的一个实际的划分方案。一般地说,F_r 的值越大,拟合的效果越好。

图 5.9　对应于表 5.9(b)的结果

关于分段拟合技术,我们有如下的建议:

1. 划分的实际段数 P_n,可以选择在整数区间[[n/25],[n/8]]内,其中 n 是用于分析过程的故障数据集合的元素的个数。换言之,包含在一个子段中的数据点数,大致可在整数区间[8,25]内选择。这个范围对于软件可靠性分析的过程是合适的。如果每个子段中的数据点数过少,将增加估计过程中的困难,得出的结果也不会好,因此将失去使用分段拟合技术的意义,而且必将使划分的段数增加,过多地增加计算量;如果每个子段中的数据点数过多,则又会使划分的段数过少,也将失去采用分段拟合技术的意义。

2. 可以在故障数据预分析的基础上选择一个划分方案,特别要注意累积数据中的“拐点”,它们往往可以作为分段点。

3. 划分的段数越多,计算量就越大,因此,也存在划分方案和精度要求之间的折衷问题。

4. 在划分方案选定之后,就可以实际估计选定的模型的参数,然后,计算每段的 F_r 值和全拟合率值。

5. 至于软件可靠性模型,我们认为 NHPP 类模型是有用的,特别是 S 形三参数 NHPP 模型更为有用。表 5.9(a)和(b)中的结果就证明了这一点。

5.2.7　三参数 NHPP 模型的参数估计过程的奇异性

我们来仔细考察式(5.2.16)的三个式子,不难发现它们并不独立,实际上,式(5.2.16c)是式(5.2.16a)和式(5.2.16b)两个方程的线性组合:

$$(5.2.16c) = bC\times (5.2.16b) - (5.2.16a),$$

所以方程组(5.2.16)并不是一个适定的问题。一方面,(5.2.16)的解可能有无穷多个,它们并不能由方程组(5.2.16)唯一确定。另一方面,即使已经有了一个比较“靠近”的初始点,在使用牛顿法或拟牛顿法来求解时,也会因为 Jacobi 矩阵线性相关,而无法进行迭代,致使最终无法得出一个精确的解。

在针对上述问题实际进行参数估计时,我们在数学上采用了路径跟踪方法。记(5.2.16a)和(5.2.16b)分别为:

$$\begin{aligned} f_1(a,b,\beta) &= 0,\\ f_2(a,b,\beta) &= 0. \end{aligned} \tag{5.2.34}$$

则(5.2.34)和(5.2.16)在下面的意义上是等价的:若(a, b, β)是(5.2.16)的解,同样它也是(5.2.34)的解,反之亦然。

由于含有一个自由度,通常(5.2.34)的解只是一些解曲线。设它的一条解曲线是{a(s), b(s), β(s)}, s∈[0, S),为一个弧长参数,且:

$$a(0) = a_0, \quad b(0) = b_0, \quad \beta(0) = \beta_0$$

是一个已知的解,则解曲线在(a_0, b_0, β_0)处的切线方向$(\dot{a}_0, \dot{b}_0, \dot{\beta}_0)$可以通过构造下列方程来确定:

$$\begin{aligned} &\frac{f_1}{a}\dot{a}_0 + \frac{f_1}{b}\dot{b}_0 + \frac{f_1}{\beta}\dot{\beta}_0 = 0 \\ &\frac{f_2}{a}\dot{a}_0 + \frac{f_2}{b}\dot{b}_0 + \frac{f_2}{\beta}\dot{\beta}_0 = 0 \\ &\dot{a}_0^2 + \dot{b}_0^2 + \dot{\beta}_0^2 = 1 \end{aligned} \tag{5.2.35}$$

求出$(\dot{a}_0, \dot{b}_0, \dot{\beta}_0)$以后,选择一个适当的步长 Δs,作预估:

$$a_1^{\circ} = a_0 + \Delta s\,\dot{a}_0, \quad b_1^{\circ} = b_0 + \Delta s\,\dot{b}_0, \quad \beta_1^{\circ} = \beta_0 + \Delta s\,\dot{\beta}_0.$$

可以得到解曲线上、下一点的一个初始近似值,然后再利用牛顿法求解扩充系统:

$$\begin{aligned} &f_1(a_1, b_1, \beta) = 0 \\ &f_2(a_1, b_1, \beta) = 0 \\ &(a_1 - a_0)\dot{a}_0 + (b_1 - b_0)\dot{b}_0 + (\beta - \beta_0)\dot{\beta}_0 = \Delta s \end{aligned} \tag{5.2.36}$$

可以得到解曲线上新的一个点(a_1, b_1, β),再以(a_1, b_1, β_1)为新的起始点,继续延拓下去,就可以计算出整个解曲线上的解。

在上述推导算法的过程中,我们假定了先要知道一个初始解(a_0, b_0, β_0),其实在求解过程的实际计算中,这一假设并非必要。由于扩充方程(5.2.36)具有良好的适定性,对于绝大多数任意给定的点(a_0, b_0, β),求解(5.2.36)的牛顿迭代过程都能很快地收敛到一个解,这个解可以用作真正的延拓起始点。

我们利用表 5.10 中的不完全数据,对 Ohba 三参数 NHPP 模型进行参数值的估计。采用路径跟踪方法,得到如下的解曲线:

表 5.10　用于 Ohba 三参数 NHPP 模型参数估计的不完全数据

t(i)	0	2	3	7	8	9	10	11	18	21	33	35	37	44	45	47	48	49	50	51	52	53	55	56	57	63	76
y(i)	0	4	5	7	8	14	17	28	29	30	31	33	41	46	48	50	53	56	59	64	67	68	69	71	74	76	78

估计出来的参数值,列于表 5.11 中。根据估计出的参数值,我们按 F_{max} 的大小,取 $F = \min\{F_{max}\}$ 的那一组作为参数的值,利用模型对这组不完全数据作了拟合,拟合结果列于表 5.12 中。在上面提及的 $F_{max} = \max(|f_1(a, b, \beta)|, |f_2(a, b, \beta)|)$。

表 5.11　参数估计值

a	b	β	F_{max}
70.79731938416829	0.0223293392111892	0.4481236104565089	.781597E-13

图 5.10　问题的解曲线图

72.04728134822393	0.0219419432654410	0.4383799577053851	.377698E-12
75.04719007801295	0.0210648441084692	0.4149951909775343	.296874E-12
78.04709882674068	0.0202551713457203	0.3916104241020541	.301092E-12
81.04700759104335	0.0195054377300255	0.3682256571051625	.589528E-12
84.04691636825108	0.0188092249910214	0.3448408900076743	.751843E-12
87.04682515622650	0.0181609996336428	0.3214561228262518	.917710E-12
90.04673395324498	0.0175559655566500	0.2980713555743359	.319078E-12
93.04664275790458	0.0169899451814203	0.2746865882628570	.783373E-12
96.04655156905784	0.0164592828575980	0.2513018209007578	.146549E-12
99.04646038575953	0.0159607658225743	0.2279170534954094	.597744E-12
102.04636920722630	0.0154915591025266	0.2045322860529145	.857980E-12
105.04627803280540	0.0150491515680194	0.1811475185783651	.806466E-12
108.04618686194970	0.0146313109762203	0.1577627510760219	.907052E-12
111.04609568419880	0.0142360463003165	0.1343779835494783	.563993E-12
114.04600452916290	0.0138615760043374	0.1109932160017716	.701217E-12
117.04591336651110	0.0135063011967089	0.0876034484354788	.779599E-12
120.04582220596070	0.0131687828091156	0.0642236808528039	.666134E-14
123.04573104726980	0.0128477221136968	0.0408389132556379	.926370E-12
126.04563989023020	0.0125419440224090	0.0174541456455972	.759837E-12

为了进行比较,我们对 G-O 模型也利用这一组不完全数据作了参数估计和拟合,并将结果列于表 5.13 中。

在得出解曲线和参数值表中的结果时,我们利用性质: $a \geq y_n$,来缩小延拓的范围,省去了许多不必要的计算。

从解曲线图上我们可以看出: 参数 β 关于 a 的解曲线是一条直线,b 关于 a 的解曲线则是一条双曲线。我们可以用下面的公式来表示它们:

$$a = a_0(1 - \beta)$$

$$b = b_0/(1 - \beta)$$

如果利用上面两个式子,解出 a_0 和 b_0,并代入 Ohba 三参数 NHPP 模型,则有:

$$m(t) = a_0(1 - e^{-b_0 t}),$$

它就是 G-O 模型, 从显式表示来看, 它已与 β无关!

我们再来仔细地考察表 5.12 和表 5.13 中的参数值与拟合结果。先看参数值: 我们用 $a_{G\text{-}O}$ 和 $b_{G\text{-}O}$ 表示 G-O 模型的两个参数值, 用 a_{Ohba}, b_{Ohba} 和 β_{Ohba} 表示 Ohba 三参数模型的三个参数值, 不难验算它们满足关系:

$$\begin{aligned} a_{Ohba} &= a_{G\text{-}O}(1 - \beta_{Ohba}) \\ b_{Ohba} &= b_{G\text{-}O}/(1 - \beta_{Ohba}) \end{aligned} \tag{5.2.37}$$

表 5.12 利用 Ohba 三参数 NHPP 模型估计的参数值和拟合结果

a = 0.120045631151E+03

β= 0.642251701530E-01

b = 0.131688037674E-01

t(i)	m(ti)	y(i)
0	0.00000000E+00	0
2	0.31230709E+01	4
3	0.46559781E+01	5
7	0.10602151E+02	7
8	0.12043459E+02	8
9	0.13467114E+02	14
10	0.14873333E+02	17
11	0.16262330E+02	28
18	0.25520473E+02	29
21	0.29250209E+02	30
33	0.42863724E+02	31
35	0.44943284E+02	33
37	0.46972218E+02	41
44	0.53692329E+02	46
45	0.54605893E+02	48
47	0.56399592E+02	50
48	0.57280000E+02	53
49	0.58149625E+02	56
50	0.59008599E+02	59
51	0.59857052E+02	64
52	0.60695115E+02	67
53	0.61522913E+02	68
55	0.63148219E+02	69
56	0.63945973E+02	71
57	0.64733957E+02	74
63	0.69263280E+02	76
76	0.78000000E+02	78

the Fitability = 0.2534325892798433

表 5.13 利用 G-O 模型估计的参数值和拟合结果

a = 128.2847300372155

b = 1.2323036722285213E-02

t(i)	m(ti)	y(i)
0	0.00000000E+00	0
2	0.31230710E+01	4
3	0.46559782E+01	5
7	0.10602151E+02	7
8	0.12043459E+02	8
9	0.13467114E+02	14
10	0.14873334E+02	17
11	0.16262331E+02	28
18	0.25520473E+02	29
21	0.29250210E+02	30
33	0.42863724E+02	31
35	0.44943287E+02	33
37	0.46972218E+02	41
44	0.53692329E+02	46
45	0.54605892E+02	48
47	0.56399593E+02	50
48	0.57280003E+02	53
49	0.58149624E+02	56
50	0.59008598E+02	59
51	0.59857052E+02	64
52	0.60695114E+02	67
53	0.61522915E+02	68
55	0.63148220E+02	69
56	0.63945972E+02	71
57	0.64733955E+02	74
63	0.69263283E+02	76
76	0.78000000E+02	78

the Fitability = 0.2534325915852046

其次, 我们再对表 5.12 和表 5.13 中的拟合结果加以对照, 不难发现, 它们几乎相同。而且, 拟合率值也近似相等。

通过上面的分析和比较, 我们可以得出:

1. 解曲线上不同的各组解都可得到相同的拟合曲线, 它们对于拟合累积故障数据具有相同的效果。

2. 这些拟合曲线都可以用一条等价的 G-O 模型拟合曲线来表示, 对于任何一条 Ohba 三参数模型的拟合曲线, 只需作式(5.2.37)的变换, 即可得到等价的 G-O 模型拟合曲线。

3. 正因为任取解曲线上的一组解, 对于拟合累积故障数据均具有相同的拟合效果, 反过来讲, 我们不可能仅从已知的累积故障数据中得到唯一的一组参数值 a, b, β。

下面我们回过来再来对 Ohba 三参数 NHPP 模型作进一步的分析。在[B14]中, Ohba 等人指出: "在查错过程中, 我们通常观察到的仅是第 i 次的观察的时间 T_i 及直到 T_i 时刻的累积错误个数"。但在讨论模型参数的估计时, 却引用了"作为不完全排错的结果的、在观察阶段引入的错误个数为 114"这样一个结果。这种作法实质上是给模型增加了一个定解条件:

$$n(t_n) - n(0) = \frac{a\beta}{1-\beta}[1 - e^{-(1-\beta)bt_n}] = 114.$$

此时, 式(5.2.16)变得唯一可解。但是, 他们在同一篇文章中又说, 在排错时引入的错误个数是不可能被观察到的, 除非是在排错时故意人为地制造错误。所以这里讲的增加定解条件的作法, 实际上是行不通的。另外, Ohba 等人又指出, 某些实际工作者发现, 即使在不完全排错的情况下, 也可以应用 G-O 模型。我们上面推导的结果以及所作的分析对比, 正好印证了这一结论。

综上所述, 我们可以得出结论: Ohba 三参数 NHPP 模型引入错误引入率 β这一参数的想法是好的, 但在作参数引入的具体过程中, 却考虑欠周。一方面, 因为 β与其它两个参数不独立, 致使问题变得不适定, 不可能从累积错误数据中唯一确定参数的估计值, 给数值求解增加了困难。另一方面, 即使求得了最大似然估计方程组的某个解, 与 G-O 模型相比, 也不可能提高拟合质量。在不完全排错的情况下, 依然可以应用 G-O 模型, 通过对参数的解释, 同样可以达到 Ohba 三参数 NHPP 模型想达到的拟合效果。

类似的奇异性, 在以 Ohba 三参数 NHPP 模型为基础而演变出来的 NHPP 类模型中都有。如:

$$(5.2.19b) = b^2C^2 \times (5.2.19c) + (5.2.19a),$$

以及:

$$(5.2.23c) = bC \times (5.2.23b) - (5.2.23a).$$

从而由此而引起的参数估计的困难问题, 都可以通过采用相应的路径跟踪具体算法加以克服。

§ 5.3 Musa 模 型

Musa 的执行时间模型以下面的条件作为基础: 在一定的时间范围内, 可靠性估计只能以实际的执行时间作为基础, 而不能以平常计算逝去多少时间的日历时间作为基础。模型的作者认为, 之所以如此, 是因为仅仅只有在执行期间, 软件本身的错误才能被暴露出来。

实际上模型分为执行时间和日历时间两部分, 前者是估计的基础, 后者则将执行时间换算成日历时间。执行时间部分是一个执行时间的指数增长模型, 程序在执行时的当前 MTTF 值是按指数形式增加; 日历时间部分是在充分考虑进度计划管理、可用资源请求

等条件限制的情况下,将执行时间转换成我们日常所习惯的日历时间。

5.3.1 模型介绍

模型的假设共有六条:

(1) 软件中错误全部彼此独立,在任何时候它们都以一个为常数的平均发生率(按每条指令计算的平均数)进行分布。

(2) 各类指令以一种良好方式混合出现,且故障之间的执行时间比平均的指令执行时间要大得多。

(3) 用于测试的空间“覆盖”了软件的使用空间。

(4) 在测试或操作期间,提供给软件每次运行所使用的输入集合都是随机选取的。

(5) 执行期间的故障全部被查出并记录。

(6) 引起每一次故障的错误,在测试重新开始前就被修正了。即:无论该错误出现多少次,错误个数只计为“1”。

模型的参数及其定义见表 5.14 和表 5.15。

表 5.14 Musa 执行时间模型主要参数一览表(之一)

参数	定 义
B	错误递减因子(错误递减率与故障发生率的平均比值)
C	测试压缩因子
f	程序的线性执行频度
k	错误暴露系数
τ	从开始到当前为止的总执行时间
τ'	计划规定的,从当前到测试终止的总执行时间
M_0	为排除软件中所有错误(N_0 个),必须排除的故障数
m	到目前为止已发生的故障数,也即用于估计模型参数时的故障样本的大小
N_0	初始错误个数(假定为常数)
N_C	已修改的错误个数
N_r	剩余错误个数($N_r = N_0 - N_C$)
P_C	机房计算机的倒班数(如三班倒,则 $P_C = 3$)
P_F	用于纠正故障的平均工作人员数
P_I	用于查明故障的平均工作人员数
T_0	测试开始时,软件的 MTTF 初值
T_P	当初的 MTTF 值
T_F	最后的(目标)MTTF 值
θ_C	每一时间段的平均计算机时间花销
θ	每一时间段的平均故障查明工作的时间花销
μ_C	平均每个故障要求的计算机时间花销
μ	平均每个故障要求的故障查明工作的时间花销
ρ_C	计算机利用系数
ρ_F	查明故障的工作人员的工作效率比值

表 5.15　Musa 执行时间模型主要参数一览表(之二)

参数	来　　源	精确程度	由验证项目得到的(加权)平均值	受该参数影响的预测值
B	收集的数据	中等	0.96	T, τt
C	以前相近项目的数据	低	10.9	T, τt
f	从平均的指令执行率和程序的大小计算得出	高		T, τt
k	以前相类似项目的初始数据,并于测试期间重新估算	低到中等	1.31×10^{-6}	T, τt
N_0	从收集的数据和初始估算的编码后的指令计算得出,并在测试期间重新估算	低到中等	错误率= 5.88 个错误/千条指令	T, τt
P_C	计算中心提供	高		t
P_F	项目开发计划	高		t
P_I	项目开发计划	高		t
T_F	通过与用户的联系,由用户提供	“完全的”		τt
θ_C	收集的数据	高	1.11	t
θ	收集的数据	高	2.15	t
μ_C	收集的数据	高	1.67 小时	t
μ_F	收集的数据	高	6.93 小时	t
μ	收集的数据	高	5.28 小时	t
ρ_C	计算中心提供	高	0.8	t
ρ_F	计算得出[对于验证的 4 个项目,假定排错人员积压的故障数不超出 2 个(概率为 90%)]	中等		t

由假设(1)和(2)有: 风险函数 $Z(\tau)$ 与 N_r 成正比,且与线性执行频度 f (平均指令执行率与程序中的指令条数的比值)成正比。于是有:

$$Z(\tau) = kfN_r \tag{5.3.1}$$

其中, k 是错误暴露频度与线性执行频度之比,称为错误暴露系数。只要不令 τ超出下一个错误改正时间范围之外,风险函数与 τ无关。于是就得到一个分段为常数的故障率模型,则:

$$Z(\tau) = fkN_0 - fkN_C \tag{5.3.2}$$

由假设(5)和(6),又设改错不引入新的错,即: 错误改正率 $dN_C/d\tau$ 等于错误暴露率,则:

$$\frac{dN_C}{d\tau} = Z(\tau) \tag{5.3.3}$$

于是得出一阶线性常微分方程:

$$\frac{dN_C}{d\tau} + fkN_C = fkN_0 \tag{5.3.4}$$

它的通解为:

$$N_C = N_0 + Ce^{-fk\tau}.$$

当 τ= 0 时, N_C= 0, 于是 N_0+ C= 0, 得到: C= - N_0, 所以方程(5.3.4)的通解又为:

$$N_C = N_0 - N_0e^{-fk\tau}.$$

所以,有:

$$N_C = N_0[1 - \exp(-fk\tau)] \tag{5.3.5}$$

前面已讨论过, MTTF 的值等于风险函数的倒数,即:

$$T = \frac{1}{Z(\tau)}.$$

它又可以写成:

$$T = \frac{1}{f k N_0 - f k N_C}.$$

将(5.3.5)代入上式, 有:

$$T = \frac{1}{f k N_0 - f k N_0 [1 - \exp(-f k \tau)]}$$

$$= \frac{1}{f k N_0} \exp(f k \tau) \qquad (5.3.6)$$

对于 $\tau = 0$, 有 $N_C = 0$, 于是, 有:

$$T_0 = \frac{1}{f k N_0}.$$

将它代入(5.3.6), 得:

$$T = T_0 \exp(f k \tau),$$

且 $f k = \frac{1}{N_0 T_0}$, 则:

$$f k \tau = \frac{\tau}{N_0 T_0}.$$

所以,

$$T = T_0 \exp \frac{\tau}{N_0 T_0} \qquad (5.3.7)$$

从(5.3.7)可知, 随 τ的增大, T 亦增大, 从这一意义上看, Musa 执行时间模型亦属于可靠性增长模型。根据:

$$R(t) = \exp[-\int_0^t \lambda(x) dx],$$

我们可以写:

$$R(\tau \mid \tau') = \exp[-\int_0^{\tau'} Z(\tau) d\tau].$$

又因为 $T = \frac{1}{Z(\tau)}$, 于是可以得出:

$$R(\tau \mid \tau') = \exp - \frac{\tau'}{T} \qquad (5.3.8)$$

下面作进一步的讨论。先定义错误递减因子 B: B 是错误递减率与故障发生率的平均比值。从收集的经验数据以及排错的实际效果出发, B 的取值不同, 反映出排错效果的不同:

- $B > 1$, 表示一次排除多个错误;
- $B = 1$, 表示错误查出立即被完全排除;
- $B < 1$, 表示引入新的错误;
- $B = 0$, 表示排错无效果;

· B< 0, 表示排错非但无效, 反而增加了。

现在, 错误改正率变成:

$$\frac{dN_C}{d\tau} = BZ(\tau).$$

但是, 还有一个事实不容忽视: 测试过程使故障的发生比使用软件时要更频繁。设在软件使用期间有由 A 个可能的不同输入集合组成的使用空间, 示于图 5.11, 图中点 a 代表第 a 个集合, 并设它在系统的生命周期中总共要被使用 s_a' 次。用 f_a 表示故障次数, τ 表示联系到包括第 a 个输入集合在内的一次"运行"的执行时间, 则在使用期间的故障率 Z 应为:

$$Z' = \sum_{a=1}^{A} \frac{f_a}{s_a'\tau}.$$

又设 s_a 表示在测试期间第 a 个集合出现的次数, 因为潜在的测试空间须覆盖系统的使用空间, 因此在测试期间的平均故障率应为:

$$Z = \sum_{a=1}^{A} \frac{f_a}{s_a \tau},$$

其中, $s_a \geq 1$。

定义测试压缩因子 C 为: 测试期间(包括执行时间在内的)故障的查明率与在使用期间的故障查明率的平均比值, 即:

$$C = \frac{Z}{Z'} = \frac{\sum_{a=1}^{A} \frac{f_a}{s_a \tau}}{\sum_{a=1}^{A} \frac{f_a}{s_a' \tau}}.$$

图 5.11 A 个输入集合的使用空间

如 $s_a = s_a'$, 则 C= 1, 但实际上 $s_a < s_a'$, 于是有: C> 1。

实际上, 由于 A 可能取无穷大, 上面的理论描述无法用于对 C 的计算。为此, 我们提出下面的近似公式:

$$C \approx \frac{\tau}{\tau},$$

其中, τ 表示测试期间, 软件每天的平均执行时间; τ 表示使用期间, 软件每天的平均执行时间。τ 的值可以是计划要求的指标, 也可以根据类似软件项目的使用中的经验数据决定。

于是, 现在错误改正率定义为:

$$\frac{dN_C}{d\tau} = BCZ(\tau),$$

则:

$$m = \frac{N_C}{B}, \quad M_0 = \frac{N_0}{B}.$$

仿照前面的推导过程, 最终得出:

$$\frac{dm}{d\tau} + BCf k_m = BCf kM_0.$$

解之,得通解,然后得出:

$$m = M_0[1 - \exp(-BCf_k\tau)].$$

最后得出:

$$T = \frac{1}{Bf_kM_0}\exp(BCf_k\tau),$$

$$T = T_0\exp\frac{C\tau}{M_0T_0} \tag{5.3.9}$$

为使 T 的值从 T_1 增至 T_2,要多排除 Δm 次故障,则:

$$\frac{T_0}{T} = \frac{1/Bf_kM_0}{(1/Bf_kM_0)\exp(BCf_k\tau)} = \exp(-BCf_k\tau).$$

因此:

$$m = M_0\left(1 - \frac{T_0}{T}\right).$$

故:

$$m = M_0\left(1 - \exp\left(-\frac{C\tau}{M_0T_0}\right)\right) \tag{5.3.10}$$

$$\Delta m = m_2 - m_1 = M_0T_0\left(\frac{1}{T_1} - \frac{1}{T_2}\right) \tag{5.3.11}$$

$$\tau = \frac{M_0T_0}{C}(\ln T - \ln T_0) \tag{5.3.12}$$

$$\Delta\tau = \frac{M_0T_0}{C}\ln\frac{T_2}{T_1} \tag{5.3.13}$$

由于进度管理上的考虑,日历时间是直观的。测试的每一步都受三种有限资源的限制:查明故障的人员、纠正故障的人员和计算机时间。一般在执行时间模型中,计算机时间只用以测量分配给计算机的资源,表现形式常常就是程序在机器中的驻留时间。如果是多道程序,则应该用可以并发执行的驻留程序的个数去除。测试与排错应由不同的人员完成。查明故障仅指判定程序不是根据对它的要求而是以某些特别的方式在工作,查错则属于纠正故障的工作范围。

分别用 $dt_I/d\tau$ $dt_F/d\tau$ $dt_C/d\tau$ 表示由资源制约的每一个单独发生的效应所引起的瞬时日历时间与执行时间的比值,则:

$$\Delta t = \int_{\tau_1}^{\tau_2}\max\left(\frac{dt_I}{d\tau}, \frac{dt_F}{d\tau}, \frac{dt_C}{d\tau}\right)d\tau$$

设:对于软件项目的日历时间在计划时留有较大余地,使得与它相比,程序员个人之间的技术水平和任务的困难程度差异都可忽略不计;而与每次故障所要求的平均故障的纠正工作时间相比,计算机装卸程序的时间,包括在纠正每一次故障的过程中的全部等待时间在内,都可以忽略不计。

全部三种资源的要求都可用经验公式表示为:

$$x = \theta\Delta\tau + \mu\Delta m,$$

其中,x 是资源要求,θ是单位执行时间花的平均资源比率,μ是每次故障的平均资源占有率。

根据上面的讨论,可以得出:

$$\Delta m = M_0 \exp\frac{-C\tau}{M_0T_0} - \exp\frac{-C\tau_2}{M_0T_0},$$

将它代入上面的经验公式,得:

$$x = \theta\Delta\tau + \mu M_0 \exp\frac{-C\tau_1}{M_0T_0} - \exp\frac{-C\tau_2}{M_0T_0} \tag{5.3.14}$$

设 P 表示有效的人员或计算机轮班制,后者是依规定的工作时间(加上平均加班时间)计算的。如:每周工作 8×6 小时,计算机有效工作时间 144 小时,则 P = 144/48 = 3。但实际上不可能工作效率达 100%,所以 P 还要乘上一个系数 ρ(ρ < 1)。于是:

$$\Delta t = \frac{\theta}{\rho P}\Delta\tau + \frac{\mu M_0}{\rho P}\exp\frac{-C\tau_1}{M_0T_0} - \exp\frac{-C\tau_2}{M_0T_0} \tag{5.3.15}$$

就表示联系到每一种资源的日历时间要求。

上式中的 τ_1 看作常量,τ_2 看作变量,即:$\Delta\tau = \tau_2 - \tau_1$,然后对之进行微分,得:

$$\frac{dt}{d\tau} = \frac{\theta}{\rho P} + \frac{\mu C}{\rho P T_0}\exp\frac{-C\tau}{M_0T_0} \tag{5.3.16}$$

将(5.3.9)式代入之,有:

$$\frac{dt}{d\tau} = \frac{\theta T + \mu C}{\rho P T}.$$

为方便计,我们对

$$\Delta t = \int_{\tau_1}^{\tau_2} \max\left(\frac{dt_I}{d\tau}, \frac{dt_F}{d\tau}, \frac{dt_C}{d\tau}\right) d\tau$$

中的积分限 τ_1, τ_2 利用式(5.3.7)进行变换:利用(5.3.7)式,对应于 τ_1,可求出 T_1;对应于 τ_2,可求出 T_2。最终可得出:

$$\Delta t = \frac{M_0T_0}{C}\int_{T_1}^{T_2}\frac{1}{T}\max_k\frac{\theta T + \mu_k C}{\rho P_k T}\,dT \tag{5.3.17}$$

其中,k 可以是 C,F 或 I(C,F,I 分别代表计算机时间、故障纠正人员、故障查明人员这三种资源)。因此,现在对于积分的每一部分有:

$$\Delta t_k = \frac{M_0T_0}{\rho P_k}\left[\mu_k\left(\frac{1}{T_{k_1}} - \frac{1}{T_{k_2}}\right) + \frac{\theta}{C}\ln\frac{T_{k_2}}{T_{k_1}}\right] \tag{5.3.18}$$

其中,T_{k_1}, T_{k_2} 是对应于同一部分的积分限,而 k 是选择值,其选择原则是对于 k 所对应部分中的任意的 T 值,能使下式:

$$\frac{\theta T + \mu C}{\rho P_k T}$$

得出最大值的 k。

每个部分积分的积分限都从由 T_1, T_2 和由下面的三个等式所决定的过渡点所组成的子集合中选取,这三个等式是:

$$\frac{dt_C}{d\tau} = \frac{dt_F}{d\tau},\quad \frac{dt_F}{d\tau} = \frac{dt_I}{d\tau},\quad \frac{dt_I}{d\tau} = \frac{dt_C}{d\tau}$$

因此,过渡点即可由下式给出:

$$T_{kk'} = \frac{C(\rho\mu_{k'}P_k - \rho_{k'}\mu P_{k'})}{\rho_{k'}\theta P_{k'} - \rho\theta_{k'}P_k} \tag{5.3.19}$$

其中，$k = C, F, I$ 而 $k' = F, I, C$。

为了用执行时间来表示 Δt，可以对每一个部分求积分，用(5.3.13)式代入(5.3.18)式，则：

$$\Delta t_k = \frac{1}{\rho P_k}\mu_k M_0 \exp\frac{-C\tau_1}{M_0T_0} - \exp\frac{-C\tau_2}{M_0T_0} + \theta_k\Delta\tau \tag{5.3.20}$$

一般地说，要想用 Δt 明确表示出 $\Delta\tau$是困难的。但如故障纠正人员有限(包括 $\theta = 0$)且 $\tau \geq \frac{M_0T_0}{C}$时，上式可简化为：

$$\Delta\tau = \frac{\rho P_F T_0}{C\mu_k}\Delta t \tag{5.3.21}$$

对于要在以日历时间计算的任一时刻结束测试，从而估算出应花多少执行时间，以上公式是很有用的。

下面讨论有关利用系数及模型参数估计的问题。

在查明故障的工作人员有限时，没有理由不充分地发挥现有人员的效率，因此，Musa在开发执行时间模型时，取 $\rho = 1$。

就是在故障纠正人员有限时，他们也不能够充分施展他们的才干，这主要是因为故障辨认时间的不可预测性和排错人员之间的工作负担分配不均衡。

故障辨认的过程，是一个用日历时间表示、时间不变的泊松过程(已经假定了故障的相互独立性，而且关于日历时间的故障率在故障纠正人员有限时为常数这一事实，上述结论可以很快地得到征明)。设 λ是这一过程的参数，故障一旦被查明，它们就会被按比例地分配给 P_F 倍的排错人员，以便他们根据由各自负责的程序区域去纠正这些故障。由于执行时间模型的假设之一，隐含了测试必须很好地分布于程序的不同部分，对于故障的分配，从日历时间的观点看来，就可以看作是对排错人员作随机选择的结果。因此，对于每个排错人员的输入率就是：λP_F。

假设故障的纠正过程也是一个时间不变的泊松过程，这也意味着一次故障的纠正与其它的故障纠正无关，而且在排错期间的任何时间内，改正一个错误的概率与在此期间的任何其它时间内来改正该错误的概率相等。虽然这样处理的结果，可能在估计那些在短时间内就可修改的错误的发生率时，得到偏高的估计值，但它仍不失为一个好的假设。每个排错人员对于故障纠正的贡献率可以看成是：$1/\mu_k$，其中 μ_k 是修改一个故障的平均工作量。

对于每个排错人员我们可以将利用系数定义为输入率与贡献率之比，即：

$$\rho_k = \frac{\lambda P_F}{1/\mu_k} = \frac{\lambda\mu_k}{P_F} \tag{5.3.22}$$

其中，ρ 为对于每个排错人员的利用系数。

经验证明，任一排错人员无论何时在一大堆故障堆积起来以后，他随时都会停止测试而去忙于进行排错，并且因此也就妨碍了在他所负责的程序区域内其它故障的辨认与纠正。

一个特定的排错人员具有为 m_Q ①的排队,或者有更多的故障等待或正在被纠正的概率,在定态时为 ρ^{m_Q}(假设 $\rho < 1$);而没有排错人员具有为 m_Q 的排队或没有更多故障待纠正的概率 P_{m_Q} 为:

$$P_{m_Q} = (1 - \rho^{m_Q}) P_F \tag{5.3.23}$$

因此,我们只要将 ρ 限制在满足方程

$$\rho = (1 - P_{m_Q}^{1/P_F})^{1/m_Q} \tag{5.3.24}$$

的范围之内,就可以假定在任何一个给定的时刻没有排错人员具有为 m_Q 的排队、或者没有更多故障待纠正的概率是 P_{m_Q}。注意,关于式(5.3.24),限制条件 $\rho < 1$ 仍然要满足。如果我们将 ρ 的值控制在由式(5.3.24)所规定的范围之内,我们将能够最大限度地发挥有效的人力,并且防止过多的工作积压起来。此外,我们也将保持 λ 为一常数,以满足前面的假设。

对于排队长度,使用了概率的定态值,因而导致对时间开销的估计稍微偏高,这是因为在集结瞬态(buildup transient)期间的排队,实际上总是短暂的。但另一方面,我们在前面也已经假定:故障的辨认和纠正是并行的过程。然而,故障纠正总是要在故障辨明之后才能开始,因此,在测试的开始和末尾阶段,总有这样的区间存在:在这些区间内,仅仅只有一个过程出现,或者是故障的辨认过程,或者是故障的纠正过程。因而我们的假设又导致对开销的日历时间估计略微偏低。于是,我们可以认为对于开销的日历时间估计的偏高与偏低两种倾向的因素,大致相互抵销。

计算机利用系数 ρ_c,基本上由控制程序装卸时间的需要来决定,正如以前提及过的,等待时间是可以忽略不计的。如果对装卸程序的时间不能加以控制(如:小规模的软件项目使用一般计算技巧的情况),则 ρ_c 就要取其实际的值,并且根据故障纠正的工作时间加上等待时间与故障纠正的工作时间之比,μ_F 也要有所增加。

Musa 执行时间模型曾有效地应用于 4 个不同的项目。同时收集了不少经验数据,确定了一些参数的经验值(见表 5.10),它们具有一定的参考价值。随着经验数据的进一步收集和积累,这些参数中的某些(如:B,μ_c,μ_F,μ,θ,θ,以及每条指令的平均错误个数,等),对于各种不同的软件开发项目以及软件开发环境,将会被证明是一些相对确定的常数,或者至少也只依赖于少量表示特性的变量。例如:计算机时间和执行时间的比 θ 就可能与应用过程中的输入和输出的比例有关。

参数 μ_c,μ_F,μ.θ 和 θ 可以通过收集故障辨认和故障纠正工作的数据以及关于相类似程序开发环境的计算机使用轮廓(连同执行时间一起)的数据,通过使它们满足(5.3.14)式,并使用加权的最小平方判别准则来进行估值。注意,μ_F,μ 和 θ 都定义为总数的形式,因而其中难免有某一因素组合得过头而未加区分,如:假期、缺席、训练和行政上的活动,等等。通常处理参数的测度将表示纯时间,它们必须由一个适当的扩大因子来增大。等于已经发生的故障数的平方根的加权因子,应该被应用到人们对后来的数据所表现出来的、适合于其解释的更大的信心的最小平方中的每一个数据点上去。参数 B 能够由收集

① m_Q 表示任何排错人员一次所能排除的故障数的极限,谁也不能达到或超过这个数字。

在修改其它的错误时引进的错误个数的数据来确定。由 Miyamoto 所收集的大量关于一般目的的软件系统开发情况的数据来看, B 的值在 0.91～0.95 之间, 这一点与验证 Musa 执行时间模型的 4 个项目的 B 值(在 0.94～1.00 之间)有些出入。在各个测试阶段的开始, 我们可以收集关于平均错误率(每条指令的平均错误数)的数据, 这些数据可用来估计 N_0 和 M_0 的值。由 4 个验证项目收集到的数据和由 Akiyama 和 Endres 给出的数据看来, 在系统测试的开始阶段, 在用汇编语言写的程序中, 每千条指令的错误大致在 3.36～7.98 个范围之内。由指令数加权的平均值是每千条指令 5.43 个错误。

C 的值必须在一个相类似的测试环境中由测量得出[见(5.3.37)式], 或者经由估算得出(如果没有进行估测的基础, 最好采取保守的办法, 取 C= 1)。参数 k 由一个相近的程序所收集到的数据来初步确定, 然后在将来再以某种方式将 k 与程序结构联系起来。

随着测试的进展, 参数 k 和 M_0 可以进行反复的再估算。但我们需要的反复估计的是 T_0 的值, 而不是 k 的值, 这是由于 T_0 所具有的实际意义远较 k 大, 而且 k 的值可以由 $T_0 = \frac{1}{f k N_0}$ 很容易地确定(当然是在确定了 T_0 的值之后)。

我们要用最大似然估计法进行反复的再估计。由下式我们可以有关于 M_0 的隐含表示:

$$\hat{} = (M_0, m) \tag{5.3.25}$$

其中, $\hat{}$ 是故障的动差统计量, 它由下式给出:

$$\hat{} = \frac{1}{m\tau_m}\sum_{i=1}^{m}(i - 1)\tau' \tag{5.3.26}$$

其中, τ' 是两个故障之间的执行时间区间, τ_m 是总的累计执行时间。现在我们有;

$$(M_0, m) = \frac{M_0}{m} - \frac{1}{\Delta\Psi} \tag{5.3.27}$$

其中,

$$\Delta\Psi = \Psi(M_0 + 1) - \Psi(M_0 + 1 - m) \tag{5.3.28}$$

这里, Ψ 是双 r 函数。

T_0 可以由下式给出:

$$T_0 = C\tau_m\left(1 - \frac{m}{M_0}\right)\hat{} \tag{5.3.29}$$

其中的 τ_m 是测试期间的无故障时间的样本平均值(the sample mean of time to failure)。

$\hat{}$ 的方差 Var($\hat{}$)由下式给出:

$$\mathrm{Var}(\hat{}) = -\frac{1}{(\Delta\Psi)^2}\frac{\Delta\Psi'}{(\Delta\Psi)^2} + \frac{1}{m} \tag{5.3.30}$$

其中,

$$\Delta\Psi' = \Psi'(M_0 + 1) - \Psi'(M_0 + 1 - m) \tag{5.3.31}$$

这里的 Ψ' 是三 r 函数。

由确定与选择满足下式的 值对应的 M_0 的值, 我们就可以用来确定置信区间:

$$= \hat{} \pm \left(\frac{1}{1 - P}\right)^{1/2} S.D.(\hat{}) \tag{5.3.32}$$

其中,

$$S.D.(\hat{\ }) = [Var(\hat{\ })]^{1/2} \tag{5.3.33}$$

P 是置信水平。置信区间与切比雪夫不等式的应用有关。虽然这一不等式倾向于保守地产生出大的区间,但在 Musa 执行时间模型应用于软件可靠性的经验积累得更多、更丰富以后,它仍然是可供选择应用的最好方法。

T_0 的方差 $Var(T_0)$ 由下式给出:

$$Var(T_0) = \frac{T_0^2}{m} \tag{5.3.34}$$

在程序的操作阶段(即发行给用户以后的使用阶段),如果我们假定对软件的错误不予以改正,则风险函数 Z(ϑ是一常数,并且最大似然法的使用亦不产生出关于 M_0 和 T_0 的两个独立的方程。然而,在上一次的系统组合测试阶段估计出的最后一个 M_0 值,可以用来估计 T_0 的值。由 $m = M_0\ 1 - \frac{T_0}{T}$ 我们有:

$$T_0 = T\ 1 - \frac{m}{M_0} \tag{5.3.35}$$

现在,在系统组合测试阶段的末尾时的 MTTF,就等于在使用阶段的平均故障区间 τ_m',因此:

$$T_0 = \ 1 - \frac{m}{M_0}\ \tau_m' \tag{5.3.36}$$

由(5.3.36)式、已知的平均值以及 τ的概率分布的方差,易于证明: T_0 的变差系数是 $1/(m)^{1/2}$。注意,T 为常数并等于 τ_m'。

如果在操作阶段的任一时刻,对于出现的故障都予以改正的话,这就好像是测试阶段的延长,只是故障区间由 C 的一个因子来减少罢了。

经验指出,对于小的样本,由修匀$^{\hat{\ }}$,能够考虑对估算的质量进行改进。另一方面,修匀过程最终将被排除掉,因为随着剩余错误数的变小,它会削弱估算算法的敏感程序。经过用大量不同的校平器进行试探之后,采用了一个可变长的校平器。它在 m= 40 处,样本长高达 40 个,而在 m= 79 处,又降至 1 个样本长(即不再进行修匀)。校平器由对应的 $m^{1/2}$ 对$^{\hat{\ }}$ 的每一个值加权。

在一次测试运行之后和使用故障区间数据之前,允许花一、两天的时间,来对记录进行整理,将虚假的故障淘汰掉,并使经过彻底分析过的结果能将漏掉的故障限制在极小的范围之内,这一作法是十分方便的。经验指出,在分析完了之后,人们宁肯漏掉故障不报,也不愿意报告虚假的故障。

虽然我们的经验指出大概 95% 或更多的故障能够很容易地就软件和非软件(硬件故障、操作员故障,等等)这两大类来进行故障分类,但仍不时会出现难以确定的情况。遵循的判定准则是: 将一个故障划归非软件故障的依据主要就是看它在所有软件和确实相同的已知非软件输入都重复运行时,是否不能使得该故障重现。

绝大多数故障都能很容易地与特定的测试时间联系起来,那些不能与特定的测试时间联系的故障,必定会与相当短的时间区间联系在一起。在这样的情况下,我们就取区间的中点作为故障时间。

如果在测试期间只有很少量的故障发生(比如 40 个或更少的故障数),由于小样本的大小过小,则置信区间一定是十分显著的。除开那些最坏情况下预测的某些狭窄区间以外,可以由假设某个故障恰好出现在测试的结束处来处理该故障的发生时间。

测试压缩因子 C 在程序的操作阶段之后,可以由下式得出:

$$C = \frac{(M_0 - m)\,\tau_m'}{(M_0 - \hat{m})\,\tau_n} \tag{5.3.37}$$

它的推导如下:

由(5.3.29)式和(5.3.36)式

$$T_0 = C\tau_n\left(1 - \frac{\hat{m}}{M_0}\right),$$

$$T_0 = \left(1 - \frac{m}{M_0}\right)\tau_m'.$$

我们有:

$$C\tau_n\left(1 - \frac{\hat{m}}{M_0}\right) = \left(1 - \frac{m}{M_0}\right)\tau_m'.$$

因此:

$$C = \frac{M_0 - m}{M_0 - \hat{m}} \cdot \frac{\tau_m'}{\tau_n}.$$

5.3.2 模型的推广

1984 年 3 月,在美国召开的第七届国际软件工程会议上,Musa 和 Okumoto 发表了一个新的执行时间模型——对数泊松执行时间模型。它仍以程序的执行时间作为基本测度,也包括执行时间和日历时间两部分。与原执行时间模型相比,对数执行时间模型虽然以原模型为基础,但有几个新的特点:(1) 模型以观察到的故障为基础,不再考虑对软件中的错误个数进行计数;(2) 新的模型作为非齐次泊松过程类的模型之一;(3) 由于修复行为,故障率是递减的。

下面先讨论模型的执行时间部分。

模型被定义为一个随机过程:$\{M(\tau), \tau \geq 0\}$,表示到执行时间达到 τ 时,发生的故障数。这一随机过程由分布 $M(\tau)$ 表示,且 $M(\tau)$ 具有均值函数:

$$\mu(\tau) = E[M(\tau)] \tag{5.3.38}$$

或者故障密度函数:

$$\lambda(\tau) = \frac{d\mu(\tau)}{d\tau} \tag{5.3.39}$$

模型的假设有:

(1) 在时刻 $\tau = 0$ 时,观察到的故障数为零,即 $M(0) = 0$ 的概率为 1。

(2) 故障密度将随观察的故障减少而呈指数下降。如用 λ_0 表示初始故障密度,用 θ 表示每个故障的故障密度这种标准方式的故障减少率,则:

$$\lambda(\tau) = \lambda_0 e^{-\theta\mu(\tau)} \tag{5.3.40}$$

(3) 对于一个小区间 $\Delta\tau$，在区间$(\tau, \tau+\Delta\tau)$内，发生一次故障的概率是 $\lambda(\tau)\Delta\tau+o(\Delta\tau)$；发生的故障多于 1 的概率是 $o(\Delta\tau)$。其中：随着 $\Delta\tau\to 0$，$\frac{o(\Delta\tau)}{\Delta\tau}\to 0$。注意：在区间$(\tau, \tau+\Delta\tau)$内不发生故障的概率由 $1-\lambda(\tau)\Delta\tau+o(\Delta\tau)$给出。

首先，由假设(1)和(2)导出均值函数和故障密度函数的函数形式。由 $\lambda(\tau)=\frac{d\mu(\tau)}{d\tau}$和 $\lambda(\tau)=\lambda\theta e^{-\theta\mu(\tau)}$有：

$$\mu'(\tau)=\lambda\theta e^{-\theta\mu(\tau)} \quad 或 \quad \mu'(\tau)e^{\theta\mu(\tau)}=\lambda\theta$$

注意：

$$\frac{d[e^{\theta\mu(\tau)}]}{d\tau}=\theta\mu'(\tau)e^{\theta\mu(\tau)},$$

我们得到：

$$\frac{d[e^{\theta\mu(\tau)}]}{d\tau}=\lambda\theta \tag{5.3.41}$$

对它积分，又可得到：

$$e^{\theta\mu(\tau)}=\lambda\theta\tau+C \tag{5.3.42}$$

C 是积分常数。因为 $\mu(0)=0$(假设 1)，于是有 $C=1$，所以有均值函数：

$$\mu(\tau)=\frac{1}{\theta}\ln(\lambda\theta\tau+1) \tag{5.3.43}$$

这是 τ的一个对数函数。从上式出发，再利用 $\lambda(\tau)=\frac{d\mu(\tau)}{d\tau}$，有故障密度函数：

$$\lambda(\tau)=\frac{\lambda\theta}{\lambda\theta\tau+1} \tag{5.3.44}$$

到 τ时刻所发生的故障数 $M(\tau)$是一个随机量。利用假设(1)和(3)，很容易证明 $M(\tau)$取值 m 的概率是：

$$P_r\{M(\tau)=m\}=\frac{[\mu(\tau)]^m}{m!}e^{-\mu(\tau)} \tag{5.3.45}$$

这是一个泊松分布。

假设在区间$(0, \tau]$内，观察到 m_e 个故障。因为泊松过程$\{M(\tau), \tau\geq 0\}$有独立的增量，对于给定 $M(\tau_e)=m_e$，$\tau>\tau_e$ 时，$M(\tau)$的条件分布是在区间$(\tau_e, \tau]$内，发生的故障数的分布，即：对于 $m\geq m_e$ 时，有：

$$\begin{aligned}P_r\{M(\tau)=m\mid M(\tau_e)=m_e\}&=P_r\{M(\tau)-M(\tau_e)=m-m_e\}\\&=\frac{[\mu(\tau)-\mu(\tau_e)]^{m-m_e}}{(m-m_e)!}e^{-[\mu(\tau)-\mu(\tau_e)]}\end{aligned} \tag{5.3.46}$$

设 $T_i'(i=1,2,\cdots)$是表示第 i 个故障区间长度的随机变量，$T_i(i=1,2,\cdots)$是表示第 i 个故障发生的时间的随机变量，即：

$$T_i=\sum_{j=1}^{i}T_j'=T_{i-1}+T_i' \tag{5.3.47}$$

其中, $T_0 = 0$。

现在来考察下面两个等价的事件:

事件 1: 到时刻 τ 为止, 至少发生了 i 个故障。

事件 2: 第 i 个故障发生的时间至少是 τ, 于是:

$$\{M(\tau) \geq i\} \quad \{T_i \leq \tau\} \tag{5.3.48}$$

因此, 可以由(5.3.45)和(5.3.48)得出 T_i 的累积分布函数 c.d.f. 如下:

$$P_r\{T_i \leq \tau\} = P_r\{M(\tau) \geq i\} = \sum_{j=1}^{\infty} P_r\{M(\tau) = j\}$$

$$= \sum_{j=i}^{\infty} \frac{[\mu(\tau)]^j}{j!} e^{-\mu(\tau)} \tag{5.3.49}$$

它表示排除前面 i 个故障的时间分布。

给定 $M(\tau) = m_e$, $i > m_e$, 由(5.3.46)和(5.3.48)可以导出 T_i 的条件 c.d.f. 如下:

$$\begin{aligned} & P_r\{T_i \leq \tau \mid M(\tau) = m_e\} \\ &= P_r\{M(\tau) \geq i \mid M(\tau) = m_e\} \\ &= \sum_{j=i}^{\infty} P_r\{M(\tau) - M(\tau) = j - m_e\} \\ &= \sum_{j=i}^{\infty} \frac{[\mu(\tau) - \mu(\tau)]^{j-m_e}}{(j-m_e)!} e^{-[\mu(\tau)-\mu(\tau_e)]}, \quad \tau \geq \tau_e \end{aligned} \tag{5.3.50}$$

T_i 关于最后的故障时间 $T_{i-1} = \tau_{i-1}$ 的条件可靠度可以用上式导出:

$$\begin{aligned} R(\tau_i \mid \tau_{i-1}) &= P_r\{T_i > \tau_i \mid T_i = \tau_{i-1}\} \\ &= 1 - P_r\{T_i \leq \tau \mid M(\tau_{i-1}) = i-1\} \\ &= 1 - \sum_{j=i}^{\infty} \frac{[\mu(\tau) - \mu(\tau_{i-1})]^{j-i+1}}{(j-i+1)!} e^{-[\mu(\tau_i)-\mu(\tau_{i-1})]} \end{aligned} \tag{5.3.51}$$

注意上面结果的第二项是除了一项以外的泊松概率之和, 因此得到:

$$R(\tau_i \mid \tau_{i-1}) = e^{-[\mu(\tau_{i-1}+\tau_i)-\mu(\tau_{i-1})]} \tag{5.3.52}$$

用(5.3.43)代入(5.3.52), 导出:

$$R\{\tau_i \mid \tau_{i-1}\} = \left[\frac{\lambda\theta\tau_{i-1}+1}{\lambda\theta(\tau_i+\tau_{i-1})+1}\right]^{1/\theta} \tag{5.3.53}$$

这就是关于最后的故障时间 τ_{i-1} 的模型的可靠度。

如果我们对它关于 τ_i 求导, 再取负, 就得 τ_i 的条件密度函数:

$$f(\tau_i \mid \tau_{i-1}) = \lambda(\tau_i+\tau_{i-1}) e^{-[\mu(\tau_i+\tau_{i-1})-\mu(\tau_{i-1})]} \tag{5.3.54}$$

因此, 风险函数就是:

$$Z(\tau_i \mid \tau_{i-1}) = \frac{R(\tau_i \mid \tau_{i-1})}{f(\tau_i \mid \tau_{i-1})} = \lambda(\tau_i+\tau_{i-1}) \tag{5.3.55}$$

模型的风险函数与故障密度函数相同。用(5.3.44)代入(5.3.55), 得出:

$$Z(\tau_i \mid \tau_{i-1}) = \frac{\lambda}{\lambda\theta(\tau_i+\tau_{i-1})+1} \tag{5.3.56}$$

关于模型的两个未知参数 λ 和 θ 可以用最大似然法进行估计。从前面的讨论中, 可以看见我们有两类可用的数据进行估计: 故障区间数据(即完全数据)和每个故障区间内

发生的故障数(即不完全数据)。下面分别进行讨论。

先讨论以故障间间隔时间数据为基础的参数估计方法。假定我们于特定的时刻 τ 进行估计,则在时间区间$(0, \tau]$内发生的故障数就是一个随机变量。于是,可以使用条件联合密度函数作为似然函数。设: 到执行时间为 τ 时,共发生 m 个故障。因为泊松过程的形式$\{T_i, i=1,2,\cdots\}$, T_{m+1}仅依赖于 T_m,则得到关于条件 $M(\tau)=m$ 的$\{T_1,\cdots,T_m\}$的联合密度函数为:

$$g(t_1,\cdots,t_m \mid m)=\frac{f(t_1,\cdots,t_m)P_r\{T_{m+1}>\tau \mid T_m=t_m\}}{P_r\{M(\tau)=m\}} \tag{5.3.57}$$

其中 $f(t_1,\cdots,t_m)$ 表示$\{T_1,\cdots,T_m\}$的无条件联合密度函数。用(5.3.54),有:

$$\begin{aligned} f(t_1,\cdots,t_m) &= \prod_{i=1}^{m} f(t_i \mid t_{i-1}) \\ &= \prod_{i=1}^{m} \lambda(t_i) e^{-[\mu(t_i)-\mu(t_{i-1})]} \\ &= e^{-\mu(t_m)} \prod_{i=1}^{m} \lambda(t_i) \end{aligned} \tag{5.3.58}$$

由(5.3.52),我们又得到:

$$P_r\{T_{m+1}>\tau \mid T_m=t_m\}=e^{-[\mu(\tau)-\mu(t_m)]} \tag{5.3.59}$$

因此,如将(5.3.45),(5.3.58),(5.3.59)代入(5.3.57),就得到条件联合密度函数:

$$g(t_1,\cdots,t_m \mid m)=m!\prod_{i=1}^{m}\frac{\lambda(t_i)}{\mu(\tau)} \tag{5.3.60}$$

注意,这个式子可以应用于任何泊松过程。另外,当随机变量 $T_1,\cdots,T_m$ 的联合密度函数具有上式的形式时,随机变量就是由概率密度函数 $\lambda(t)/\mu(\tau)$产生的有序统计量。换言之,随机有序的故障时间是全同独立分布的。

用(5.3.43),(5.3.44)代入(5.3.60),有:

$$g(t_1,\cdots,t_m \mid m)=m!\prod_{i=1}^{m}\frac{\lambda\theta}{(\lambda\theta\tau+1)(\lambda\theta t_i+1)} \tag{5.3.61}$$

它可用于估计参数 $\omega(=\lambda\theta)$的似然函数。令 $L=\ln g(t_1,\cdots,t_m \mid m)$,则:

$$\frac{L}{\omega}=\frac{m}{\omega}-\sum_{i=1}^{m}\frac{t_i}{\omega t_i+1}-\frac{m\tau}{(\omega\tau+1)\ln(\omega\tau+1)}=0 \tag{5.3.62}$$

因为它的非线性,所以不能得出解析解,但可求得它的数值解。

利用(5.3.62)估计出的是$\hat{\omega}$,而不是 λ 和 θ。为了分别得到 λ 和 θ,我们要利用条件:"到时刻 τ 为止,已观察到 m 个故障"。因此,在 τ 时,有:

$$\mu(\tau)=m \tag{5.3.63}$$

将(5.3.43)代入(5.3.63),得到:

$$\frac{1}{\theta}\ln(\lambda\theta\tau+1)=m \tag{5.3.64}$$

将上面估计出的$\hat{\omega}$代入(5.3.64),得:

$$\theta = \frac{1}{m}\ln(\hat{\tau} + 1) \tag{5.3.65}$$

最后，由 $\hat{\lambda\theta} = \lambda\theta$ 可求出 $\lambda = \hat{\lambda\theta} / \theta$

上面讨论的参数估计方法，只要令 $\tau_m = \tau_n$，则可用于对第 m 个故障出现时的情况下的参数估计。

下面讨论以每个故障间隔内发生的故障数为基础数据的参数估计方法。设观察时间区间 $(0, x_p]$ 划分为 p 个互不相交的子区间 $(0, x_1], (x_1, x_2], \cdots, (x_{p-1}, x_p]$。下面要利用条件联合密度函数，与数据 $y_1, y_2, \cdots, y_p$，对参数 λ 和 θ 进行估计。

参数 $Y_l (l = 1, 2, \cdots, p)$ 构成一个泊松过程，因此：

$$f(y_1, \cdots, y_p) = \prod_{l=1}^{p} P_r\{M(x_l) = y_l\}$$

$$= \prod_{l=1}^{p} P_r\{M(x_e) = y_l \mid M(x_{l-1}) = y_{l-1}\} P_r\{M(0) = 0\} \tag{5.3.66}$$

其中，$x_0 = 0$，$y_0 = 0$，以及 $P_r\{M(0) = 0\} = 1$。将 (5.3.46) 代入 (5.3.66)，得出：

$$f(y_1, \cdots, y_p) = \prod_{l=1}^{p} \frac{[\mu(x_l) - \mu(x_{l-1})]^{y_l'}}{y_l'!} e^{-[\mu(x_l) - \mu(x_{l-1})]} \tag{5.3.67}$$

其中，y_l' 表示区间 $(x_{l-1}, x_l]$ 内的故障数，即：

$$y_l' = y_l - y_{l-1} \tag{5.3.68}$$

以 $M(x_p) = y_p$ 为条件，联合密度函数为：

$$g(y_1, \cdots, y_p \mid y_p) = \frac{f(y_1, \cdots, y_p)}{P_r\{M(x_p) = y_p\}} \tag{5.3.69}$$

由 (5.3.68)，有：$y_p = \sum_{l=1}^{p} y_l'$。将 (5.3.45) 和 (5.3.67) 代入 (5.3.69)，并注意到 y_p 的事实，则有：

$$g(y_1, \cdots, y_p \mid y_p) = y_p! \prod_{l=1}^{p} \frac{1}{y_l'} \left[\frac{\mu(x_l) - \mu(x_{l-1})}{\mu(x_p)}\right]^{y_l'} \tag{5.3.70}$$

将 (5.3.43) 代入 (5.3.70)，得到：

$$g(y_1, \cdots, y_p \mid y_p) = y_p! \prod_{l=1}^{p} \frac{1}{y_l'} \left[\frac{\ln(\lambda\theta x_l + 1) - \ln(\lambda\theta x_{l-1} + 1)}{\ln(\lambda\theta x_p + 1)}\right]^{y_l'} \tag{5.3.71}$$

它可用作似然函数，以估计 $\lambda\theta$ $(= \lambda\theta)$ 的估计值 $\hat{\lambda\theta}$。

令 $L = \ln g(y_1, \cdots, y_p \mid y_p)$，则有：

$$L = \ln(y_p!) - \sum_{l=1}^{p} \ln y_l' + \sum_{l=1}^{p} y_l' \ln[\ln(x_l + 1) - \ln(x_{l-1} + 1)]$$

$$- y_p \ln[\ln(x_p + 1)] \tag{5.3.72}$$

于是，有：

$$\frac{L}{} = \sum_{l=1}^{p} y_l' \frac{\frac{x_l}{x_l + 1} - \frac{x_{l-1}}{(x_l + 1)}}{\ln(x_l + 1) - \ln(x_{l-1} + 1)} - \frac{y_p x_p}{(x_p + 1)\ln(x_p + 1)} = 0$$

(5.3.73)

同样,对于它不能得解析解,但可求出数字解,即$\hat{}$。

于是,有:

$$\theta = \frac{1}{y_p}\ln(\hat{}\, x_p + 1),$$
$$\lambda_0 = \hat{}\,/\,\theta \tag{5.3.74}$$

关于对数泊松执行时间模型的日历时间部分,更实际地应用于软件项目开发的系统测试阶段。它也有3个假设:

(1) 在任何时刻的测试进度都受到三种有限资源之一的限制:查明故障的工作人员(测试小组)——以I表示;纠正故障的工作人员(原设计者)——以F表示;计算机时间——以C表示。这一假设可以写成:

$$\frac{dt}{d\tau} = \max\left(\frac{dt_I}{d\tau}, \frac{dt_F}{d\tau}, \frac{dt_C}{d\tau}\right).$$

(2) 关于执行时间$\frac{dx_k}{d\tau}$的资源开销率可以用下面的经验公式近似表示:

$$\frac{dx_k}{d\tau} \equiv \theta_k + \mu_k \frac{d\mu(\tau)}{d\tau}, \quad k = I, F, C \tag{5.3.75}$$

其中,θ_k是资源开销的执行时间系数,μ_k是资源开销的故障系数,对于特定的资源($k = I, F, C$),θ_k或μ_k可以为零(这里的资源开销即指一种资源用一段时间)。

(3) 在剩余的测试阶段内,有效的资源量都是常数。每一种有效资源的可用的最大程度也是有限的。如用P_k和ρ分别表示,对应的k种($k = I, F, C$)资源的固定可用量和利用因子,则k种资源的有效可用量为ρP_k。

利用上面的三个假设,可以推导出日历时间和执行时间之间的关系。将(5.3.44)代入(5.3.75),可得资源开销率为:

$$\frac{dx_k}{d\tau} = \theta_k + \mu_k \frac{\lambda_0}{\lambda_0\theta\tau + 1}, \quad k = I, F, C \tag{5.3.76}$$

因为有效的可用资源为ρP_k,则:

$$\frac{dt_k}{d\tau} = \frac{dx_k}{d\tau}\Big/\rho P_k$$
$$= \frac{1}{\rho P_k}\left(\theta_k + \mu_k \frac{\lambda_0}{\lambda_0\theta\tau + 1}\right) \tag{5.3.77}$$

由假设(1),可以得出:

$$\frac{dt}{d\tau} = \max_k\left(\frac{dt_k}{d\tau}\right), \quad k = I, F, C \tag{5.3.78}$$

其它的讨论与§ 5.3中的5.3.1讨论的执行时间模型的日历时间部分相同。

5.3.3 实例

在测试期间,由于排错工作的进展,M_0就会发生变化(通常都是下降趋势),而且T_0会逐渐增加(在每次重新开始测试的初始MTTF的值T_0,总比前一次的T_0要大,这正好反映出排错努力的成效)。于是,随着测试的进展,对它们就要持续地反复进行再估算,以此来不断地反映出测试进展的量化程度(主要表现形式就是具有各种不同意义的时间)。

同样,由于对 M_0 和 T_0 进行再估算的结果,又使得对于下述的一系列参数的再估算能很容易地进行。

(1) 当前的 MTTF 的值: T_P

利用到目前为止所使用的执行时间 τ,以及式(5.3.9):

$$T = T_0 \exp\left(\frac{C}{M_0 T_0}\tau\right)$$

以估算 T_P 的值,并将它与 T_F 比较,如达到或超过了 T_F,则表示测试已达目标,测试于是就可以停止了,否则,测试还须继续进行下去。

(2) 对于 T 的最大似然估计,为达到由用户规定的目标 MTTF 的值 T_F,所要求的执行时间的增量: $\Delta\tau$

利用 $T_2 = T_F$, $T_1 = T_P$ 以及式(5.3.13)来计算 $\Delta\tau$

$$\Delta\tau = \frac{M_0 T_0}{C}\ln\frac{T_2}{T_1} = \frac{M_0 T_0}{C}\ln\frac{T_F}{T_P}.$$

(3) 为达到 T_F,所要求的日历时间增量: Δt

使用式(5.3.18)的三个部分的取值:

$$\Delta t_k = \frac{M_0 T_0}{\rho P_k}\left(\mu_k\left(\frac{1}{T_{k_1}} - \frac{1}{T_{k_2}}\right) + \frac{\theta_k}{C}\ln\frac{T_{k_2}}{T_{k_1}}\right),$$

在区间$[T_P, T_F]$上估算式(5.3.17)的值:

$$\Delta t = \frac{M_0 T_0}{C}\int_{T_1}^{T_2}\frac{1}{T}\max_k\left(\frac{\theta_k T + \mu_k C}{\rho P_k T}\right)dT$$

$$= \frac{M_0 T_0}{C}\int_{T_P}^{T_F}\frac{1}{T}\max_k\left(\frac{\theta_k T + \mu_k C}{\rho P_k T}\right)dT.$$

(4) 为达到 T_F,所要求暴露并纠正的剩余故障数: Δm

使用式(5.3.11),并取 $T_1 = T_P$, $T_2 = T_F$,可计算 Δm:

$$\Delta m = m_2 - m_1 = M_0 T_0\left(\frac{1}{T_1} - \frac{1}{T_2}\right) = M_0 T_0\left(\frac{1}{T_P} - \frac{1}{T_F}\right).$$

除了"最可信"的值以外,置信限可以由实施对与 M_0 和 T_0 相对应的置信限的计算来确定(如: 关于 $\Delta\tau$的置信限就是为达到 T_F 的执行时间增量所要求的 T 的置信边界)。整个的结果对于计划进度、估计进展情况、判定何时可以终止测试的管理人员是十分有用的。虽然,50%,90%,和 95% 等置信级别有助于加深对问题的了解,但 75% 的置信级别上的估计是从管理人员的观点出发的、由最有用的经验产生出来的,因而也最适于用作判定的依据。

下面通过一个实际的例子来看看将模型用于测试进度管理的过程。

由于被测试的目标软件本身的特性,在我们的问题讨论中,没有任何实际的意义,因此我们对它们不用去研究。但是,关于对它们进行测试的环境,却与我们这里要研究的问题有很密切的关系,因此,有必要对它们加以说明。事先声明一句: 该测试环境自始至终不变。关于测试环境的参数示于表 5.16 中。

表 5.16　软件项目的测试环境

PARAMETERS	
TTL= * * * * * * SANPLE　RUN　N-1　* * * * * *	
TTL=	
C=	10.600
TF=	27.800
MUC=	1.850
MUF=	8.780
MUI=	7.210
PF=	5.000
PI=	1.000
RHOF=	0.270
THETA C=	1.240
THETA I=	2.400
PC=	2.470
RHOC=	0.800
LMAX=	40
LMIN=	1
L1=	40
L2=	79
MONTH=	2
DATE=	25
YEAR=	78
DAY=	6
WORKDA=	7.500
WORKWK=	3.000

下面,将表中与我们讨论的问题有关的内容逐项予以解释:

C: 测试压缩因子,取 C= 10.6,它表示测试要以比预料中的使用状态的频率和负荷都要高些。它的确定主要以以前的经验数据为基础,是一个相对的值,多少含有主观的因素。

TF: MTTF 的目标值,它的确定主要根据用户对开发的软件所提的要求,这里取 MTTF= 27.8 小时。

MUC: 平均每个故障的辨认及纠正所花费的计算机使用时间,这里取 MUC= 1.85 小时。

MUF: 平均每个故障的修改工作量,取 MUF= 8.78 人-小时。

MUI: 平均每个故障的辨认工作量,取 MUI= 7.21 人-小时。

PF: 修改故障的工作人数,专职与兼职的共计为 5 人。

PI: 辨认故障的工作人数,计 1 人。

RHOF: 修改故障人员的工作效率,取 RHOF= 0.27(即: 共计为 5 人,但实际上只承担了 5× 0.27= 1.35 人的工作量)。

THETA C: 每天计算机的利用时间,取 THETA C= 1.24 小时/天。

THETA I: 每天辨认故障的工作量,取 THETA I= 2.4 人-小时/天。

PC: 计算机的倒班率, 取 PC= 2.4。

RHOC: 计算机利用率, 取 RHOC= 0.8。

另外, 与测试有关的时间如下:

每天用于测试的操作时间共计为 7.5 小时/天;

每星期有 3 天用于测试;

系统测试从 1978 年 2 月 18 日开始;

第一次估算的时间是 2 月 25 日;

以后大约每星期估算一次, 直到达到 MTTF 的目标值 T_F 时为止(27.8 小时)。

表 5.17 示出进行历次估算的日期。

表 5.17 实施估算的日期

日期	共经过的天数(天)	累积故障区间数(个)
1978 年 2 月 25 日	7	13
1978 年 3 月 4 日	14	15
1978 年 3 月 11 日	21	19
1978 年 3 月 18 日	28	23
1978 年 3 月 25 日	35	24
1978 年 4 月 1 日	42	28
1978 年 4 月 8 日	49	34
1978 年 4 月 15 日	56	38

表中“日期”一栏是指历次实施估算的具体日期, 从表中我们可以看出每七天实施估算一次。“共经过的天数”是指从测试开始的第一天算起, 到历次实施估算的那一天, 总共经过的天数。“累积故障区间数”指从开始测试以来到历次实施估算时, 总共用到的故障区间数, 亦即累积的故障数。

表 5.18 示出第一次估算时(1978 年 2 月 25 日)所用到的、输入模型的故障区间数据。

表 5.18 自测试开始(1978 年 2 月 18 日)至第一次估算(1978 年 2 月 25 日)所发生的故障区间数据表

SEQ NO	INTERVALS	TAU P
1	115	1
2	0	1
3	83	3
4	178	3
5	194	3
6	136	3
7	1077	3
8	15	3
9	15	3
10	92	3

11	50	3

续表

SEQ NO	INTERVALS	TAU P
12	71	3
13	606	6
14	- 1189	7

表中,“SEQ NO”是给故障区间按顺序编的号;“INTERVALS”是每次故障发生的间隔时间,以秒为单位计;“TAU P”指相应的故障发生时,离测试开始日期的累积天数。

在这些数据输入模型之后,模型即根据这些基本数据,进行估算,然后,将估算的结果以表 5.19 所示的格式输出。

在表 5.19 中第 14 号故障区间列出的是一个带负号的数据: - 1189,它表示在第一次实施估算时,最后一次发生的故障(第 13 号故障)到进行估算开始时的时间区间。之所以标上负号,是表示输入数据结束,并没有其它意义。

下面逐项将表 5.19 的内容解释如下,输出中的小圆圈及其中的数字是为解释方便而加的,并非输出中的符号。

表 5.19 所示的输出格式分三个大的部分:

(1) 估计用的环境信息(①～⑤):

① 给出了到进行估算时所发生的总故障数,从表中可看出是 13 个故障,即 m 的值;

② 表示到进行估算时的累积执行时间(为 1.06 小时),即 τ

③ 可靠性的目标 MTTF 值: 27.8 小时,即 TF;

④ 从测试开始(1978 年 2 月 18 日)起到本次估算所经过的天数(7 天);

⑤ 进行估算的日期(1978 年 2 月 25 日)。

(2) 可靠性估计结果信息(⑥～⑩):

⑥ 概率分别为 50%, 75%, 90%, 95% 的置信区间及最大似然估计值;

⑦ 估计的总故障数,即 N_0;

⑧ 初始 MTTF 的值,即 T_0;

⑨ 当前 MTTF 的值,即 TP;

⑩ 对于可靠性目标的完成率(%)。

(3) 为达到目标 MTTF(TF)的值,还需要做的工作(⑪～⑭):

⑪ Δm 的估计值;

⑫ $\Delta\tau$的估计值;

⑬ Δt 的估计值;

⑭ 估计出的完成测试的日期。

表 5.19中的(2)和(3)两个部分都给出了估计的置信区间,下面解释一下它们的意义: 图中这两部分的“MOST LIKELY”一项给出最大似然估计值,在它左侧的部分给出估计的各个置信区间的乐观值,它的右侧部分是各置信区间的悲观值。例如: 对于⑨中的

表 5.19　第一次估算的输出(1978 年 2 月 25 日)

SOFTWARE　RELIABILITY　PREDICTION

* * * * * *　S　A　M　P　L　E　　R　U　N　　N- 1　* * * * * *

①→BASED ON SAMPLE OF　　13 TEST FAILURES

②→EXECUTION TIME IS　　1. 06 HRS

③→MTTF OBJECTIVE IS　　27. 80 HOURS

④→CALENDAR TIME TO DATE IS　　7 DAYS

⑤→PRESENT DATE: 2/ 25/ 78

		CONF. LIMITS			MOST LIKELY			CONF. LIMITS	
⑥→	95%	90%	75%	50%		50%	75%	90%	95%
⑦→TOTAL FAILURES	13	13	13	15	80	999999	999999	999999	999999
⑧→INITIAL MTTF(HR)	0.	0. 0676	0. 245	0. 334	0. 550	0. 766	0. 855	1. 03	1. 23
⑨→PRESENT MTTF(HR)	999999.	999999.	999999.	3. 15	0. 710	0. 334	0. 245	0. 0676	0.
⑩→PERCENT OF OBJ	100. 0	100. 0	100. 0	11. 3	2. 55	1. 20	0. 881	0. 243	0.
* * * ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE * * *									
,? →FAILURES	0	0	0	1	60	999999	999999	999999	999999
,? →EXEC. TIME(HR)	0.	0.	0.	1. 03	15. 2	999999. 0	999999. 0	999999. 0	999999. 0
,? →CAL. TIME(DAYS)	0.	0.	0.	1. 69	62. 9	999999. 0	999999. 0	999999. 0	999999. 0
,? →COMPLETION DATE	2/ 1/ 78,	3/ 1/ 78,	3/ 1/ 78,	3/ 3/ 78,	7/ 26/ 78,	99/ 99/ 99,	99/ 99/ 99,	99/ 99/ 99,	99/ 99/ 99

当前 MTTF 的估计值的 50% 的置信区间左边为 3.15, 右边为 0.334, 即表示 T_P 在区间 [0.334, 3.15] 之间的概率为 50%, 也即表示: 0.334 小时≤T_P≤3.15 小时的概率为 50% (估计的概率)。

表5.20和表5.21分别给出1978年4月8日和4月15日进行估算时的故障区间数据:

表 5.20　4 月 8 日进行估算时的故障区间数据(经过 49 天)

SEQ NO	INTERVALS	TAU P
1	115	1
2	0	1
3	83	3
4	178	3
5	194	3
6	136	3
7	1077	3
8	15	3
9	15	3
10	92	3
11	50	3
12	71	3
13	606	6
14	1189	8
15	40	8
16	788	18
17	222	18
18	72	18
19	615	18
20	589	26
21	15	26
22	390	26
23	1863	27
24	1337	30
25	4508	36
26	834	38
27	3400	40
28	6	40
29	4561	42
30	3186	44
31	10571	47
32	563	47
33	2770	47
34	652	48
35	- 5593	49

表 5.21　4 月 15 日进行估算时的故障区间数据(经过 56 天)

SEQ NO	INTERVALS	TAU P
1	115	1
2	0	1
3	83	3
4	178	3
5	194	3
6	136	3
7	1077	3
8	15	3
9	15	3
10	92	3
11	50	3
12	71	3
13	606	6
14	1189	8
15	40	8
16	788	18
17	222	18
18	72	18
19	615	18
20	589	26
21	15	26
22	390	26
23	1863	27
24	1337	30
25	4508	36
26	834	38
27	3400	40
28	6	40
29	4561	42
30	3186	44
31	10571	47
32	563	47
33	2770	47
34	652	48
35	5593	50
36	11696	54
37	6724	54
38	2546	55

表5.22和表5.23分别给出在1978年4月8日和4月15日进行的两次估算的输出结果。

表5.22 4月8日(经过49天)的估算输出结果

SOFTWARE RELIABILITY PREDICTION

****** SAMPLE RUN N-1 ******

BASED ON SAMPLE OF 34 TEST FAILURES
EXECUTION TIME IS 12.89 HRS
MTTF OBJECTIVE IS 27.80 HOURS
CALENDAR TIME TO DATE IS 49 DAYS
PRESENT DATE: 4/8/78

	CONF. LIMITS				MOST			CONF. LIMITS	
	95%	90%	75%	50%	LIKELY	50%	75%	90%	95%
TOTAL FAILURES	34	34	34	35	40	57	80	999999	999999
INITIAL MTTF(HR)	0.382	0.751	1.08	1.24	1.64	2.04	2.20	2.53	2.90
PRESENT MTTF(HR)	999999,	999999,	999999,	28.8	13.2	6.61	4.78	0.751	0.382
PERCENT OF OBJ	100.0	100.0	100.0	100.0	47.3	23.8	17.2	2.70	1.37
	*** ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE ***								
FAILURES	0	0	0	0	3	13	31	999999	999999
EXEC. TIME(HR)	0.	0.	0.	0.	4.63	15.7	29.3	999999.0	999999.0
CAL. TIME(DAYS)	0.	0.	0.	0.	4.00	17.9	38.7	999999.0	999999.0
COMPLETION DATE	4/12/78,	4/12/78,	4/12/78,	4/12/78,	4/21/78,	5/24/78,	7/12/78,	99/99/99,	99/99/99

表5.23 4月15日(经过56天)时的估算输出结果

SOFTWARF RELIABILITY PREDICTION

****** SAMPLE RUN N-1 ******

BASED ON SAMPLE OF 38 TEST FAILURFS
EXECUTION TIME IS 21.54 HRS
MTTF OBJECTIVE IS 27.80 HOURS
CALENDAR TIME TO DATE IS 56 DAYS
PRESENT DATE: 4/15/78

	CONF. LIMITS				MOST			CONF. LIMITS	
	95%	90%	75%	50%	LIKELY	50%	75%	90%	95%
TOTAL FAILURES	38	38	38	38	43	54	66	204	999999
INITIAL MTTF(HR)	0.607	1.08	1.49	1.70	2.21	2.72	2.93	3.35	3.82
PRESENT MTTF(HR)	999999,	999999,	999999,	999999,	24.4	12.9	9.54	4.68	0.607
PERCENT OF OBJ	100.0	100.0	100.0	100.0	87.7	46.3	34.3	16.8	2.18
	*** ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE ***								
FAILURES	0	0	0	0	1	6	13	121	999999
EXEC. TIME(HR)	0.	0.	0.	0.	1.17	10.7	19.5	114.8	999999.0
CAL. TIME(DAYS)	0.	0.	0.	0.	0.835	9.30	19.0	153.5	999999.0

如果进度管理计划安排测试必须于 4 月 20 日完成,通过对 4 月 15 日的估算输出结果的分析,于 4 月 20 日结束测试可能不能达到 TF 的目标:

从 4 月 15 日的最大似然估计结果看来,当前只完成了整个测试工作的 87.7%;在时间范围 4 月 19 日—5 月 11 日完成全部测试工作的概率只有 50%;在时间范围 4 月 19 日—6 月 2 日完成全部测试工作的概率为 75%;如果将概率增加到 90%,则要完成全部测试工作的时间就在范围:1978 年 4 月 19 日—1979 年 4 月 12 日之内。因此,于 4 月 20 日一定完成全部测试工作的把握不大。

于是,有必要采取一系列措施,加快测试的进度。一般可供选择的措施有:

- 要想法强化测试过程,如:考虑新的 test cases,提高测试压缩因子(对于本例子,C = 10.6 已相当高了,不宜再提高,只有在新的 test cases 方面想办法了)。
- 增加每天的工作时间,加班加点地干。如:原来是 7.5 小时/天,可以考虑改为 10 小时/天。
- 增加每星期的工作天数。如:原来用于测试工作的天数是 3 天/星期,现在可以考虑改为 6 天/星期。
- 增加工作人员或提高工作效率。如:原来 PF = 5,可以考虑改为 PF = 10;原来 PI = 1,可以考虑改为 PI = 5。另外,原来 RHOF = 0.27,现在考虑全力以赴从事测试工作,则可以取 RHOF = 1。
- 增加计算机时间,提高机器的使用效率。如:THETAC = 1.24 小时/天,可以改为 THETAC = 5.0 小时/天;原来 RHOC = 0.8,可以考虑改为 RHOC = 0.9。
- 减少每个故障的修改工作量,也即提高修改故障的工作效率。如:可以将原来的 MUF = 8.78 人-小时,压缩为 MUF = 5.0 人-小时。

将上述各种措施进行综合考虑,将测试环境参数作适当的改变,并将它们作为模型的输入,同时使用表 5.21 所示的 4 月 15 日进行估算时的故障区间数据,重新进行估算。

经过修改以后的测试环境参数示于表 5.24 中。

表 5.24 修改以后的测试环境参数

	PARAMETERS		PARAMETERS
C=	10.600	THETA I=	2.400
TF=	27.800	LMAX=	40
MUC=	1.850	LMIN=	1
MUF=	5.000	L1=	40
MUI=	7.210	L2=	79
PC=	2.470	MONTH=	4
PF=	10.000	DATE=	8
PI=	5.000	YEAR=	78
RHOC=	0.900	DAY=	6

RHOF=	1.000	WORKDA=	10.000
THETA C=	5.000	WORKWK=	6.000

将表 5.24 的新测试环境参数和表 5.18 的原故障区间数据一起输入模型,重新估算后的结果如表 5.25 所示。

表 5.25　修改测试环境参数后于 4 月 15 日重新估算以后的输出结果

SOFTWARE RELIABILITY PREDICTION (FOR P O M)

BASED ON SAMPLE OF	38 TEST FAILURES
EXECUTION TIME IS	21.54 HRS
MTTF OBJECTIVE IS	27.80 HOURS
CALENDAR TIME TO DATE IS	56 DAYS
PRESENT DATE: 4/15/78	

	CONF. LIMITS							CONF. LIMITS	
	95%	90%	75%	50%	LIKELY	50%	75%	90%	95%
TOTAL FAILURES	38	38	38	38	43	54	66	204	999999
INITIAL MTTF(HR)	0.607	1.08	1.49	1.70	2.21	2.72	2.93	3.35	3.82
PRESENT MTTF(HR)	999999.	999999.	999999.	999999.	24.4	12.9	9.54	4.68	0.607
PERCENT OF OBJ	100.0	100.0	100.0	100.0	87.7	46.3	34.3	16.8	2.18
	* * * ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE * * *								
FAILURES	0	0	0	0	1	6	13	121	999999
EXEC. TIME(HR)	0.	0.	0.	0.	1.17	10.7	19.5	114.8	999999.
CAL. TIME(DAYS)	0.	0.	0.	0.	0.304	2.91	5.49	35.9	999999.
COMPLETION DATE	4/15/78, 4/15/78, 4/15/78, 4/15/78, 4/17/78, 4/19/78, 4/22/78, 5/27/78, 99/99/99								
STOP									

从表上可以看出:完成日期的最大似然估计是 4 月 17 日。但从置信区间来看分别为:

50%:4 月 15 日—4 月 19 日;

75%:4 月 15 日—4 月 22 日;

90%:4 月 15 日—5 月 27 日。

于 4 月 20 日按时完成全部测试工作的可能性仍不大,但是却证明对于测试环境参数的调整还是发生了效应的。

如果在 4 月 8 日的估算时即注意到了这样的情形,并将测试环境参数作了如表 5.24 所示的相应调整,使用 4 月 8 日所使用过的故障区间数据(示于表 5.20 中),于当天重新进行估算,结果会如何呢?

在这样作了之后,我们从估算输出的结果中可以看到:完成全部测试工作的估算日期的最大似然估计是 4 月 11 日。从置信区间来看,它们分别是:

50%:4 月 8 日—4 月 14 日;

75%: 4月8日—4月20日;

90%:4月8日—……。

可以看出: 在4月20日完成全部测试工作的概率有75%。于4月20日交货大概是可以有把握的了。具体结果示于表5.26中。

表5.26 修改测试环境参数后于4月8日重新估算以后的输出结果

SOFTWARE RELIABILITY PREDICTION (FOR P O M)

BASED ON SAMPLE OF	34 TEST FAILURES
EXECUTION TIME IS	12.89 HRS
MTTF OBJECTIVE IS	27.80 HOURS
CALENDAR TIME TO DATE IS	49 DAYS
PRESENT DATE: 4/8/78	

	CONF. LIMITS			MOST			CONF. LIMITS	
	95%	90%	75%	50%	LIKELY	50%	78%	90%	95%
TOTAL FAILURES	34	34	34	35	40	57	80	999999	999999
INITIAL MTTF(HR)	0.382	0.751	1.08	1.24	1.64	2.04	2.20	2.53	2.90
PRESENT MTTF(HR)	999999.	999999.	999999.	28.8	13.2	6.61	4.78	0.751	0.382
PERCENT OF OBJ	100.0	100.0	100.0	100.0	47.3	23.8	17.2	2.70	1.37
	* * * ADDITIONAL REQUIREMENTS TO MEET MTTF OBJECTIVE * * *								
FAILURES	0	0	0	0	3	13	31	999999	999999
EXEC. TIME(HR)	0.	0.	0.	0.	4.63	15.7	29.3	999999.	999999.
CAL. TIME(DAYS)	0.	0.	0.	0.	1.26	4.66	9.12	999999.	999999.
COMPLETION DATE STOP	4/8/78, 4/8/78, 4/8/78, 4/8/78, 4/11/78, 4/14/78, 4/20/78, 99/99/99, 99/99/99								

至此,将Musa的执行时间模型应用于测试进度管理的例子就介绍完了。从以上介绍看,及早发现无法按时完成任务的情况,以利于及时做出相应的调整,是保证按时交货的关键。

§ 5.4 超几何分布模型及参数估计

Y.Tohma和R.Jacoby等人开发的超几何分布模型,其统计模型为可放回的随机选取问题。设在一袋中有m个白色小球,一次从中取出w个并涂上红色,然后全部放回袋中。重复若干次。在每次的尝试中,w个小球中没有被涂上红色的小球个数是一个随机变量,记为N(i),它服从超几何分布。记P{x(i)=n, w, C(i-1)}表示N(i)以x(i)为其值的概率,于是有

$$P\{x(i)\mid m, w, C(i-1)\} = \frac{\binom{m-C(i-1)}{x(i)} \binom{C(i-1)}{w-x(i)}}{\binom{m}{w}},$$

其中,

$$C(i-1) = \sum_{k=1}^{i-1} x(k),$$

$$C(0) = 0.$$

$x(k)$ 为 $N(k)$ 观察到的一个实例。$N(i)$ 的期望值用 $\overline{N(i)}$ 表示, 有:

$$\overline{N(i)} = \{m - C(i-1)\}\frac{w}{m}.$$

于是, 利用观察值 $\{x(k)\mid k = 1, 2, \cdots, i-1, i, \cdots, n\}$, 就可以估计参数 w, m 的值。

极大似然估计(LME)的方法是: 对于似然函数

$$l(m, w) = \prod_{i=1}^{n} P\{x(i)\mid m, w, C(i-1)\},$$

我们可以得出:

$$\begin{aligned}
&\sum_{i=1}^{n} \frac{1}{m} \ln \frac{\binom{m-C(i-1)}{x(i)} \binom{C(i-1)}{w-x(i)}}{\binom{m}{w}} = 0 \\
&\sum_{i=1}^{n} \frac{1}{w} \ln \frac{\binom{m-C(i-1)}{x(i)} \binom{C(i-1)}{w-x(i)}}{\binom{m}{w}} = 0
\end{aligned} \tag{5.4.1}$$

最小二乘估计(LSE)方法为: 构造估计函数

$$s(m, w) = \sum_{i=1}^{n} \left[x(i) - \frac{[m - C(i-1)]w}{m}\right]^2$$

于是可以得到:

$$\begin{aligned}
&\frac{1}{m}\sum_{i=1}^{n} \left[x(i) - \frac{[m - C(i-1)]w}{m}\right]^2 = 0 \\
&\frac{1}{w}\sum_{i=1}^{n} \left[x(i) - \frac{[m - C(i-1)]w}{m}\right]^2 = 0
\end{aligned} \tag{5.4.2}$$

它的解为:

$$m = \frac{\sum_{i=1}^{n} C(i-1) \sum_{i=1}^{n} x(i) C(i-1) - \sum_{i=1}^{n} x(i) \sum_{i=1}^{n} [C(i-1)]^2}{n \sum_{i=1}^{n} x(i) C(i-1) - \sum_{i=1}^{n} x(i) \sum_{i=1}^{n} C(i-1)}$$

$$w = \frac{\sum_{i=1}^{n} x(i) C(i-1) \sum_{i=1}^{n} C(i-1) - \sum_{i=1}^{n} x(i) \sum_{i=1}^{n} [C(i-1)]^2}{[\sum_{i=1}^{n} C(i-1)]^2 - n \sum_{i=1}^{n} [C(i-1)]^2}$$

超几何分布模型考虑了以下一些因素：测试情况(test instances)，记为 $t(i)$；测试条件与数据(test cases)，记为 $t_c(j)$。模型作者将错误分为两大类：已查出的错误，记为 y；新查出的错误，记为 x。也即是说：在 $t(i)$ 中，被查出的错误数 $w(i) = x + y$，其中 y 是在以前的测试中(即在 $t(1), t(2), \cdots, t(i-1)$ 中)曾被查出过，这次又重新被查出的错误个数。$w(i)$ 称为测试函数的易度(the ease of test function)。于是我们有下面的数据形式(定义 $t(0) = 0, w(0) = 0, C(0) = 0$)：

$t(1), w(1)$　　(全部为新查出的错误。)

$t(2), w(2)$　　(一部分为新查出的错误，下同。)

⋮　　⋮

$t(i-1), w(i-1)$　($C(i-1)$ 由 $t(1), \cdots, t(i-1)$ 所查出的新错误的累积数。)

超几何分布模型的假设如下：

1. 由 $t(i)$ 查出的全部错误不立即排除，其中一部分有可能被 $t(i+1)$ 重新查到。
2. 排错过程不引入新错。
3. $w(i)$ 是随机的，与初始错误个数无关。
4. 由 $t(i)$ 查出的错误个数 $w(i)$，其中有一部分是第一次被查出的(即 m 个中的一部分)，记 $w(i)$ 为 m 和测试进展 $p(i)$ 的函数。$p(i)$ 定义为测试人员的技巧：$w(i) \leq m$，$w(i) = m \cdot p(i)$。

于是得出超几何分布模型的均值函数为：

$$E[C(i)] = E[m] \left[1 - \prod_{j=1}^{i} \left(1 - \frac{w(j)}{E[m]}\right)\right]$$

$$= E[m] \left[1 - \prod_{j=1}^{i} (1 - p(i))\right] \qquad (5.4.3)$$

或：

$$E[C(i)] = E[m] \left[1 - \exp\left(\sum_{j=1}^{i} \ln\left(1 - \frac{w(j)}{E[m]}\right)\right)\right]$$

$$= E[m] \left[1 - \exp\left(\sum_{j=1}^{i} \ln(1 - p(i))\right)\right] \qquad (5.4.4)$$

其中，$E[C(i)]$ 表示在 $t(i)$ 时，新被查出的累积错误个数的估计值，$E[m]$ 表示总的初始错误个数的估计值。

设 $w(i) = E[m] \cdot [E[m] \cdot i + E[b]]$，则有：

$$\prod_{j=1}^{i}(1-aj-b)=(1-b)^{i}\times\prod_{j=1}^{i}(1-cj),$$

$$c=\frac{a}{1-b}.$$

下面讨论三种与超几何分布模型有关的量。

① $w(i)$：在 $t(i)$ 中被查出的错误总数。对于不同的应用，可以定义它的不同形式，因此，它是十分灵活的。根据以上的介绍，我们知道：

$$w(i)=f_n(i)+f_r(i),$$

其中，$f_n(i)$ 表示在 $t(i)$ 中，新被查出的错误个数，$f_r(i)$ 表示在 $t(i)$ 中，重新被查出的错误个数。

② $$f_n(i)=E[C(i)]-E[C(i-1)]$$

$$=\prod_{j=1}^{i-1}(1-p(i))\times w(i) \qquad (5.4.5)$$

显见，$f_n(1)=w(1)$，$f_n(0)=0$，所以超几何分布模型的均值函数又可写为：

$$E[C(i)]=\sum_{k=1}^{i}f_n(k)=\sum_{k=1}^{i}\prod_{j=1}^{k-1}(1-p(j))\,w(k),$$

$$w(k)=E[m]\times p(k) \qquad (i=1,2,\cdots,n) \qquad (5.4.6)$$

③ $$f_r(i)=w(i)-f_n(i)$$

$$=w(i)\times\left[1-\prod_{j=1}^{i-1}(1-p(i))\right] \qquad (i=2,3,\cdots,n) \qquad (5.4.7)$$

$$f_r(1)=0,\quad f_r(0)=0.$$

以上①～③都直接与 $w(i)$ 的形式密切相关。下面列表给出 $w(i)$ 的一些不同形式。

表 5.27　$w(i)$ 的几种不同形式

$w(i)=E[m]\cdot p(i)$	待估计的参数
其它的函数形式	$E[a]$，$E[b]$，
(a)　$w(i)=E[m]\cdot I(i)\cdot(ai+b)$	$E[c]$，$E[p]$，
(b)　$w(i)=E[m]\cdot(ai+b)$	$E[\ ]$，$E[\ ]$.
(c)　$w(i)=E[m]\cdot(ai^2+bi+c)$	
(d)　$w(i)=E[m]\cdot(a\cdot tester(i)^p)$	
(e)　$w(i)=E[b]=E[m]\cdot(1-e^{-\ })$	
(f)　$w(i)=E[m]\cdot\left[1-\left(1+\frac{\rho}{1+\rho(i-1)}\right)e^{-\rho}\right]$	

表 5.27 中的(a)称为信息增值 $w(i)$—函数(Information Enhanced $w(i)$-function)，信息可以是涉及测试的工作人员人数，或在 $t(i)$ 时，用于执行的测试条件与数据的数目。(e)和(f)的 $w(i)$ 分别反映了 G-O 模型与超几何分布模型 δS 形 NHPP 模型与超几何分布模型的相互关系。

对于(a)，(b)，(c)，当 $i\to\infty$ 时，有 $w(i)\to\infty$，这一般是不应出现的。为克服这一问题，

定义:

$$w(i) = E[m] \times \left(1 - e^{-(E[a]i + E[b])}\right) \tag{5.4.8}$$

下面讨论它的参数估计问题。采用最小二乘法,参数的最佳估计值必须满足:

$$\min(EF_1) = \min \frac{1}{n}\sum_{i=1}^{n} |C(i) - E[C(i)]| \tag{5.4.9}$$

$$\min(EF_2) = \min \sum_{i=1}^{n} e_i^2 = \min \sum_{i=1}^{n} (C(i) - E[C(i)])^2 \tag{5.4.10}$$

将(5.4.3)式代入(5.4.10)式,有

$$SSE = S(p(i), E[m]) = \sum_{i=1}^{n} C(i) - E[m] \times \left(1 - \prod_{j=1}^{i}(1 - p(j))\right)^2.$$

作为例子,不妨设 $p(i) = E[a]i + E[b]$,(其它的 $p(i)$函数可以一样处理)。于是有:

$$\frac{SSE}{E[m]} = \frac{S(p(i), E[m])}{E[m]}$$

$$= \sum_{i=1}^{n} C(i) - E[m] \times \left(1 - \prod_{j=1}^{i}(1 - p(i))\right)^2 \times \left(-\left(1 - \prod_{j=1}^{i}(1 - p(j))\right)\right)$$

$$= 0.$$

可以将 $E[m]$ 移到等式的一边,得到:

$$E[m] = \frac{\sum_{i=1}^{n} C(i) \times \left(1 - \prod_{j=1}^{i}(1 - p(i))\right)}{\sum_{i=1}^{n} \left(1 - \prod_{j=1}^{i}(1 - p(j))\right)^2},$$

$$\frac{s(p(i), E[m])}{E[a]} = \sum_{i=1}^{n} C(i) - E[m] \times \left(1 - \prod_{j=1}^{i}(1 - p(j))\right)$$

$$\times E[m] \times \exp\left[\sum_{j=1}^{i} \ln(1 - p(i))\right] \times \sum_{j=1}^{i} \frac{-j}{1 - p(j)} = 0,$$

$$\frac{S(p(i), E[m])}{E[b]} = \sum_{i=1}^{n} C(i) - E[m] \times \left(1 - \prod_{j=1}^{i}(1 - p(j))\right)$$

$$\times E[m] \times \exp\left[\sum_{j=1}^{i} \ln(1 - p(i))\right] \sum_{j=1}^{i} \frac{-1}{1 - p(j)} = 0.$$

第六章　非随机过程类模型

本章我们将讨论运用贝叶斯方法于软件可靠性研究的贝叶斯模型,如 Littlewood-Verrall 模型,将贝叶斯理论应用于 JM 模型的研究而衍生出的一些模型。另外,关于种子撒播/加标记的模型(即 Seeding 模型)、Nelson 的基于输入域的模型,也将在本章内予以介绍。最后,关于时间序列分析法、试探性数据分析方法等一些在软件可靠性研究中有着广泛应用前景的方法,也将加以介绍。

下面我们先从总的方面介绍一下贝叶斯方法在软件可靠性中的应用。

贝叶斯统计方法与传统的经典方法的主要区别在于对先验知识的利用。以参数的统计问题为例,经典方法认为总体的分布是带有参数 θ的密度函数 $P(x,\theta)$,数据是来自 $P(x,\theta)$的一个样本观测值,因此在原则上,样本 $x_1,\cdots,x_n$ 的联合分布密度 $P(x_1,\cdots,x_n,\theta)$是已知的,把 θ看成是客观存在的常数,统计推断是由样本 $x_1,\cdots,x_n$ 对 θ作出估计或判断。

贝叶斯方法则认为:即使在进行观测以得到样本 x 之前,人们对 θ也会有一些知识。这可以是由某种理论、以往在对付同类问题时所积累的经验、或者是观察者的主观认识所产生的。由于这种知识是在试验前得到的,故可称为验前或先验知识(Prior Information)。表述这种先验知识的最简单和方便的方法,是将 θ视为一随机变量,而给出它的一个概率分布 $G(\theta)$,于是根据概率论中贝叶斯公式,就可以求出 θ对样本 $x_1,\cdots,x_n$ 的条件分布,称为后验分布,记为 $h(\theta|x_1,\cdots,x_n)$,则有:

$$h(\theta|x_1,\cdots,x_n)=\frac{P(x_1,\cdots,x_n|\theta)g(\theta)}{\int P(x_1,\cdots,x_n|\theta)g(\theta)\,d\theta} \tag{6.0.1}$$

其中 $g(\theta)$是 $G(\theta)$相应的密度函数。依据(6.0.1)的 $h(\theta|x_1,\cdots,x_n)$就可以对参数 θ进行估计与推断。

在软件可靠性分析中,程序在给定时间内能否正确执行是一个随机问题,这种随机性可以看作是由两个随机因素造成的:一是输入数据的选择,二是程序本身的编制,即不同的程序设计员会编制不同的程序。早期的估测模型均只考虑了前一个随机因素,因此假设故障数据满足泊松分布,即故障间隔时间满足指数分布:

$$pdf(t|\lambda)=\lambda\exp(-\lambda t),\ t>0,\ \lambda>0 \tag{6.0.2}$$

其中故障率 λ是一固定但未知的常数。然而更切实际的分析,还应考虑第二个随机因素,即程序本身的不确定性。程序本身的不确定性将导致引起程序执行出错的输入空间 I_F 不确定。如果接受(6.0.2)式,I_F 的不确定则转化为 λ的不确定。事实上可以认为 λ是 I_F 的一个尺度,这样 λ不再被看作是固定(未知)的参数,而被看作是一个随机变量。将参数看

作随机变量正与贝叶斯方法的观点相符合,这也许是在软件可靠性分析中,采用贝叶斯方法的根本原因之一。利用贝叶斯方法估计早期模型中的参数似乎更切合实际一些,因此,我们下面将讨论 J-M 模型中参数的贝叶斯推断。

大多数软件可靠性估测模型都是可靠性增长模型,例如 Littlewood 和 Verral 提出的模型假设故障率随机下降,可靠性则随机增长。然而实际情况并非总是如此,因为在测试过程中,改正错误的同时可能又引入新的错误,从而导致可靠性下降。如果采用贝叶斯经验贝叶斯方法(Bayes Empirical-Bayes,下面简称 BEB)重新推导 L-V 模型中的参数,则可反映这种可靠性并非总增长的情况。在后面部分我们将给出用 BEB 方法推导 L-V 模型中参数的全过程,并将由此产生的新模型简称为 BEB 模型。

下面介绍贝叶斯方法的一般步骤。

贝叶斯统计推断一般分为两步:

1. 确定参数的先验分布,利用联合密度及贝叶斯公式,求得后验分布。

2. 从后验分布作出对参数的推断。

参数的先验分布的选取是贝叶斯方法的核心问题。面对这个问题,已形成了三种不同的处理方法。一种观点认为,先验信息就是人们心理上的、主观的一个信念,这种信念有时是由经验形成的,有时是逻辑上的判断。为了能客观一些决定一个先验分布,可以多征求一些专家、有经验的实际工作者的意见,然后进行综合决定。这种方法往往因人而异,很难有一定的规则,一般称之为主观学派。另一种观点认为,先验分布的选取应与目前看到的样本有联系(或利用过去的样本),这就是经验贝叶斯方法。第三种方法就是通常所说的无信息的先验分布(Noninformative Priors)。无信息是相对于试验而言的。因为统计问题中的数据都是统计试验的观测值,它们自然提供了关于参数的信息,如果我们并没有使用数据所提供的信息来决定先验分布,这种方法就称为无信息的先验分布。

样本的联合概率密度 $P(x,\theta)$,其中 x 与 θ 均可为向量,反映了样本 x 与参数 θ 之间的联系,它既反映了参数 θ 对 x 的影响,也反映了样本 x 中含有参数 θ 怎样的“信息”。因此先验分布的选取,自然与 $P(x,\theta)$ 的函数形式有关。把 x 看成固定的常数,θ 看成是变量,记密度 $P(x,\theta)$ 为 $L(\theta|x)$,称它为似然函数(θ 的似然函数)。$L(\theta|x)$ 即(6.0.1)式中的 $P(x|\theta)$。

依据(6.0.1)的 $h(\theta|x)$ 就可以对参数 θ 进行估计与推断。例如象点估计,可以有最大后验估计、最小均方误差估计等等,也可以用后验密度给出 θ 的区间估计。如最大后验密度区间估计(HPD 区间估计)。同样也可以进行假设检验。

经验贝叶斯方法是想把经典方法和贝叶斯方法结合起来,寻求一个更好的处理问题的统计方法。它的基本想法是这样的:参数 θ,样本 x 均为随机变量或向量,先验分布函数为 $G(\theta)$,x 对 θ 的条件密度是 $P(x|\theta)$,因此样本 x 的边缘分布就是:

$$f(x)=\int p(x|\theta)\,dG(\theta)$$

利用样本 x 采用经典方法去估计密度 $f(x)$ 所含的参数,由此确定先验分布 $G(\theta)$,这样先验分布也是从经验的值——样本——中获得的,因此称为经验贝叶斯(Empirical Bayes)方法,简称为 EB 方法。

下面讨论 J-M 模型参数的贝叶斯推导。

最早提出且影响较大的模型是 Jelinski 和 Moranda(1972)提出的 J-M 模型。该模型的基本假设是:

(i) 任一时刻,软件的故障率正比于程序中残留的错误数;程序最初具有 N 个错误。

(ii) 每个错误都具有相同的引起故障发生的可能性,用未知量 λ表示这种可能性。

(iii) 已知 λ,N,时间序列 $T_1, \cdots, T_N$ 相互独立(T_i 表示第 i-1 次故障到第 i 次故障间的时间)。则风险函数定义为:

$$\lambda(t_i) = \lambda(N - i + 1).$$

T_i 的条件密度定义为:

$$f(t_i \mid N, \lambda) = \lambda(N - i + 1)\exp\{-\lambda(N - i + 1)t_i\}, (i = 1, \cdots, N),$$

其中,N,λ是未知参数。

已知 $t_1, \cdots, t_k$, $k \leq N$, Jelinski 和 Moranda 采取极大似然估计 N 和 λ。

J-M 模型存在着两个关键问题:

(i) 假设"程序具有固定的初始错误总数"以及"每个错误都具有相同的引起故障发生的可能性"与实际情况不相符。

(ii) 采用经典方法——极大似然估计法——估计参数 N、λ会产生不合实际的结果。

针对问题(i) Littlewood 和 Verral(1973)提出 L-V 模型,该模型不再考虑错误总数,并设故障率随机下降,对此我们将在下面再予以详细说明。这里主要讨论 N、λ的贝叶斯推导。

前面已经指出,用贝叶斯方法估计(6.0.2)式中的参数 λ会更符合实际情况,而根据程序的不确定性,J-M 模型中的参数 N,λ亦会具有不确定性,因此,用贝叶斯方法估计 N,λ同样会更合实际情况。

用贝叶斯方法估计 N,λ必须假设(N,λ)服从某种先验分布 F,下面就分三种具体情况讨论如何用贝叶斯方法推导参数 N,λ。

第一种情况:假设 N 的先验分布是具有参数 θ的泊松分布,λ具有固定值 λ,则 N = q、$q \geq k$ 的似然函数为:

$$L(N = q \mid t_1, \cdots, t_k) \propto \frac{q!}{(q-k)!}\lambda^k \exp\left\{-\lambda\sum_{j=1}^{k}(q - j + 1)t_j\right\},$$

则:
$$L(N = q \mid t_1, \cdots, t_k) \cdot p\{N = q\} \propto \frac{e^{-\theta}\theta^q\lambda^k}{(q-k)!}\exp\left\{-\lambda\sum_{j=1}^{k}(q-j+1)t_j\right\}.$$

由贝叶斯公式得:

$$p\{N = q \mid t_1, \cdots, t_k\} = \frac{\theta^{q-k}}{(q-k)!}\exp\left\{-\lambda\sum_{j=1}^{k}t_j\right\}^{q-k} \cdot e^{-\theta}\exp\left\{-\lambda\sum_{j=1}^{k}t_j\right\} \quad (6.0.3)$$

如果设 $r(t_1, \cdots, t_k) = \theta \cdot \exp\left\{-\lambda\sum_{j=1}^{k}t_j\right\}$,则从(6.0.3)式可看出(N-k)服从泊松分布,其期望值为 $r(t_1, \cdots, t_k)$。

第二种情况:假设 N 为固定的已知值 n,λ服从参数为 μ、α的 Γ-分布,记为 $\lambda \sim \Gamma(\mu, \alpha)$,则 λ的似然函数为:

$$L(\lambda \mid t_1, \cdots, t_k) \propto \frac{n!}{(n-k)!}\lambda^{k}\exp\left\{-\lambda\sum_{j=1}^{k}(n-j+1)t_j\right\},$$

$$L(\lambda \mid t_1, \cdots, t_k)\,P(\lambda) \propto \frac{n!}{(n-k)!}\,\frac{\mu^{\alpha}}{\Gamma(\alpha)}\,\lambda^{k+\alpha-1}\exp\left\{-\lambda\left(\mu+\sum_{1}^{k}(n-j+1)t_j\right)\right\}.$$

对 λ 积分得:

$$f(t_1, \cdots, t_k) \propto \frac{n!}{(n-k)!}\,\frac{u^{\alpha}}{\Gamma(\alpha)}\,\Gamma(k+\alpha)\left(\mu+\sum_{j=1}^{k}(n-j+1)t_j\right)^{-(k+\alpha)}.$$

由贝叶斯公式得 λ 的后验分布服从 $\psi\left(\mu+\sum_{j=1}^{k}(n-j+1)t_j, k+\alpha\right)$。

第三种情况:假设 N 有任意的先验分布,而 $\lambda \sim \psi(\mu, \alpha)$,N 与 λ 相互独立,则:

$$f(t_1, \cdots, t_k, N=q, \lambda) \propto \frac{\mu^{\alpha}}{\Gamma(\alpha)}\,\frac{q!}{(q-k)!}\exp\left\{-\lambda\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)\right\}P\{N=q\}\cdot\lambda^{\alpha+k-1}.$$

对 $q=k, \cdots, \infty$ 求和,对 λ 积分得:

$$f(t_1, \cdots, t_k) \propto \frac{\mu^{\alpha}\Gamma(\alpha+k)}{\Gamma(\alpha)}\sum_{q=k}^{\infty}\frac{q!}{(q-k)!}P\{N=q\}\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)^{-(\alpha+k)}.$$

由贝叶斯公式得:

$$P\{N=q, \lambda=\lambda \mid t_1, \cdots, t_k\}$$

$$= \frac{\lambda^{\alpha+k-1}e^{-\lambda\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)}}{\Gamma(\alpha+k)}\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)^{(\alpha+k)}$$

$$\times \frac{\frac{q!}{(q-k)!}\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)^{-(\alpha+k)}P\{N=q\}}{\sum_{q=k}^{\infty}\frac{q!}{(q-k)!}\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j\right)^{-(\alpha+k)}P\{N=q\}}$$

若已知 $N=q$,则 λ 的后验分布服从 $\psi\left(\mu+\sum_{j=1}^{k}(q-j+1)t_j, k+\alpha\right)$,而 $N=q(q\geq k)$ 的后验分布是:

$$P\{N=q \mid t_1, \cdots, t_k\} = \frac{\frac{q!}{(q-k)!}\left(\mu+\sum_{j=1}^{q}(q-j+1)t_j\right)^{-(\alpha+k)}p\{N=q\}}{\sum_{j=k}^{\infty}\frac{r!}{(r-k)!}\left(\mu+\sum_{j=1}^{q}(r-j+1)t_j\right)^{-(\alpha+k)}P\{N=q\}}.$$

将第一种情况和第二种情况进一步一般化就可分别得到 Goel-Okumoto 模型(1979)和 L-V 模型。

下面讨论贝叶斯经验贝叶斯(BEB)模型的有关问题。

BEB 模型建立在 L-V 模型的基础上,L-V 模型的基本假设如下:

1. 故障间隔时间 $T_i(i=1, 2, \cdots)$ 条件独立且密度函数为:

$$f(t_i \mid \lambda_i) = \lambda_i\exp(-\lambda_i t_i) \qquad (6.0.4)$$

其中,λ_i 表示第 i 个测试阶段的故障率。

2. λ_i 相互独立,密度函数为:

$$f(\lambda_i \mid \psi(i), \alpha) = \frac{[\psi(i)]^{\alpha}\lambda_i^{\alpha-1}\exp\{-\psi(i)\lambda_i\}}{\Gamma(\alpha)},$$

$$\psi(i) = \beta + \gamma i \tag{6.0.5}$$

其中递增函数 $\psi(i)$ 保证了 $\{\lambda\}$ 随 i 随机下降，从而使得可靠性不断增长。参数 α、β、γ 均可采用极大似然法估计，利用贝叶斯方法可求得 λ 的后验分布。显然 L-V 模型属于经验贝叶斯模型。

为了用贝叶斯经验贝叶斯方法推导 L-V 模型的待测参数，还必须增加如下假设：

3. 给定全部背景信息集 H，则参数 α、β、γ 的先验边缘分布密度为：

$$\Pi_1(\alpha|H) = \frac{1}{W}, \qquad 0 < \alpha < W;$$

$$\Pi_2(\beta|H,\gamma) = \frac{b^{\alpha}}{\Gamma(\alpha)}(\beta+\gamma)^{\alpha-1}e^{-b(\beta+\gamma)}, \qquad \beta > -\gamma$$

$$\Pi_3(\gamma|H) = \frac{q^{c}}{\Gamma(c)}\gamma^{c-1}e^{-q\gamma}, \qquad \gamma > 0;$$

其中，$w > 0$，$a > 0$，$b > 0$，$c > 0$，$q > 0$ 均为已知量。

4. 已知 H 时，α 独立于 β、γ。

5. 给定 α、β、γ，$\lambda^{(n)}$，T_i 相互独立，且独立于 α、β、γ 以及 $\lambda_j(j = 1, \cdots, n)$。

其中，$\lambda^{(n)} = \{\lambda_1, \lambda_2, \cdots, \lambda_n\}$，同样设 $t^{(n)} = (t_1, \cdots, t_n)$。

本模型的目的是在已知数据 $X^{(n)}$ 下，推断故障率 λ_i 和下一次出错时间 t_{n+1}。下面就运用贝叶斯经验贝叶斯方法推导 λ_i、T_{n+1}。

将 λ_i 的后验分布记为 $P(\lambda_i|t^{(n)})$。由交换扩展律（law of the extension of conversation）得：

$$P(\lambda_i|t_i^{(n)}) = \iiint P(\lambda_i|t_i^{(n)}, \alpha, \beta, \gamma)\Pi(\alpha, \beta, \gamma|t_i^{(n)})\,d\alpha\,d\beta\,d\gamma \tag{6.0.6}$$

其中，$P(\lambda_i|t_i^{(n)}, \alpha, \beta, \gamma)$ 为给定 $t^{(n)}$、α、β、γ 时 λ_i 的条件概率密度函数。$\Pi(\alpha, \beta, \gamma|t_i^{(n)})$ 是超参数 (α, β, γ) 的后验联合分布。

由贝叶斯公式得：

$$\Pi(\alpha, \beta, \gamma|t_i^{(n)}) = \frac{P(t^{(n)}|\alpha, \beta, \gamma)\Pi(\alpha, \beta, \gamma|H)}{\iiint P(t^{(n)}|\alpha, \beta, \gamma)\Pi(\alpha, \beta, \gamma|H)\,d\alpha\,d\beta\,d\gamma} \tag{6.0.7}$$

其中，$P(t^{(n)}|\alpha, \beta, \gamma)$ 为 α、β、γ 的似然函数，$\Pi(\alpha, \beta, \gamma|H)$ 为超参数 (α, β, γ) 的联合先验分布。

通过扩展以包括 $\lambda^{(n)}$ 得：

$$P(t^{(n)}|\alpha, \beta, \gamma) = \int \cdots \int P(t^{(n)}|\lambda^{(n)}, \alpha, \beta, \gamma) \times P(\lambda^{(n)}|\alpha, \beta, \gamma)\,d\lambda_1 \cdots d\lambda_n$$

由前面关于 λ_i、T_i 相互独立的假设知：

$$P(t^{(n)}|\alpha, \beta, \gamma) = \prod_{i=1}^{n}\int f(t_i|\lambda_i)g(\lambda_i|\alpha, \beta, \gamma)\,d\lambda_i = \prod_{i=1}^{n} P(t_i|\alpha, \beta, \gamma) \tag{6.0.8}$$

由(6.0.4)、(6.0.5)得：

$$P(t_i|\alpha, \beta, \gamma) = \frac{\alpha(\beta+\gamma)^{\alpha}}{(t_i+\beta+\gamma)^{\alpha+1}}.$$

下面需要估计 $P(\lambda_i|t_i^{(n)}, \alpha, \beta, \gamma)$，由贝叶斯公式知：

$$P(\lambda \mid t_n, t^{(n-1)}, \alpha, \beta, \gamma) \propto P(t_n \mid \lambda, t^{(n-1)}, \alpha, \beta, \gamma) \cdot P(\lambda \mid t^{(n-1)}, \alpha, \beta, \gamma).$$

由假设 3 知:

$$P(\lambda \mid t^{(n)}, \alpha, \beta, \gamma) \propto f(t_n \mid \lambda) g(\lambda \mid \alpha, \beta, \gamma),$$

由(6.0.4)、(6.0.5)得:

$$P(\lambda \mid t^{(n)}, \alpha, \beta, \gamma) = \frac{(\lambda)^{\alpha}(t_n + \beta + \gamma)^{\alpha+1} e^{-\lambda(t_n+\beta+\gamma)}}{\Gamma(\alpha+1)} \qquad (6.0.9)$$

由(6.0.6)～(6.0.9)则可求得 λ 的后验分布。

由 $t^{(n)}$ 预测 T_{n+1} 的分布可由下式得到:

$$P(t_{n+1} \mid t^{(n)}) = \iiint P(t_{n+1} \mid t^{(n)}, \alpha, \beta, \gamma) \Pi(\alpha, \beta, \gamma \mid t^{(n)}) \, d\alpha \, d\beta \, d\gamma$$

$$= \iiint P(t_{n+1} \mid \alpha, \beta, \gamma) \Pi(\alpha, \beta, \gamma \mid t^{(n)}) \, d\alpha \, d\beta \, d\gamma \qquad (6.0.10)$$

(6.0.6)和(6.0.10)式可采用 Lindley 提出的近似方法求解。

§ 6.1 运用贝叶斯估计的贝叶斯模型

6.1.1 Littlewood-Verrall 模型

在介绍模型之前,先将要用到的数学符号列举如下:

t_i': 第 $i-1$ 次故障之后的一个故障发生的时间区间

Z_i: 风险函数

T_i': 第 i 个故障区间

λ: 初始故障强度

V_0: 在时间无限远处的故障数的期望值

$\lambda(t)$: 故障强度函数, $\lambda(t) = d\mu/dt$

$\xi(i)$: Γ-函数的标参

α Γ-函数的形参

β, β: 模型参数

$\mu(t)$: 到时刻 t 为止,发生的故障次数的期望值, $\mu(t) = E[m(t)]$ 即 $E[M(t)]$

$f(t_i')$: T_i'的概率密度函数

$f(t_i' \mid Z_i)$: 以风险函数为条件的、T_i'的概率密度函数

$g(Z_i)$: 风险函数的概率密度函数

$\Theta(i)$: 第 $i-1$ 个与第 i 个故障间的 MTTF

下面介绍 L-V 模型。

Littewood-Verrall 提出(在 1973 年): T_i' 有着以 Z_i 为条件的指数分布,即:

$$f(t_i' \mid Z_i) = Z_i \exp(-Z_i t_i'), \qquad (6.1.1)$$

Z_i 被假定为一个随机变量,它具有带标参 $\xi(i)$ 和形参 α 的 Γ-分布。如果用 $g(Z_i)$ 表示风险

函数的概率密度函数,则:

$$g(Z_i) = \frac{\xi(i)[\xi(i)Z_i]^{\alpha-1}\exp[-\xi(i)Z_i]}{\Gamma(\alpha)} \qquad (6.1.2)$$

因此,T_i'的无条件分布就为:

$$f(t_i') = \int_0^{\infty} f(t_i'|Z_i)g(Z_i)\,dZ_i$$

$$= \alpha\frac{\xi(i)}{t_i'+\xi(i)}^{\alpha}\frac{1}{t_i'+\xi(i)} \qquad (6.1.3)$$

这是一个 Pareto 分布。则:

$$\Theta(i) = E[T_i'] = \frac{\xi(i)}{\alpha} \qquad (6.1.4)$$

关于结果软件系统的风险函数,有:

$$Z(t_i') = \frac{\alpha}{t_i'+\xi(i)} \qquad (6.1.5)$$

图 6.1 示出这一关系。

软件的风险函数是随着时间 t_i'的增加而连续下降的,而对于每个故障发生时,其风险函数下降的高度是不相同的。关于在每个故障区间内风险函数的下降,Littlewood 和 Verrall 则坚持认为只要软件是不发生故障运行,则对于"软件的风险函数是低的"这一点的信任程度就在增加。可以证明 $\xi(i)$是个可靠性增长函数,模型是一般性的且很灵活,但是它也是很复杂的,与其它许多模型比较起来,要应用它是困难的。

根据它的形式,模型亦可归为不同的类别。Littlewood 和 Verrall 对它进行变动如下:

图 6.1 L-V 模型的风险函数

$$\xi(i) = \beta_0 + \beta_1 i \qquad (6.1.6)$$

以及:

$$\xi(i) = \beta_0 + \beta_1 i^2 \qquad (6.1.7)$$

这两个式子中的 i 都没有任何限制,所以发生无穷多个故障是可能的。它们分别为反线性的和反多项式(二次)。

对于一个给定的增长函数的参数值,判定其中哪个是最好的比较方法,是以对数据的良好拟合作为依据。Littlewood 和 Verrall 使用 Cramer-von Mises 统计量,采取在多维曲面上进行反复的搜索和分类,并求统计量的极小值的方法。但是这一算法要求的计算机时间太多。由 Iannino 在 1979 年的工作指出,参数的最大似然估计比良好拟合法多少可以降低计算的误差。

关于(6.1.6)的 $\lambda(t)$和 $\mu(t)$的函数式可以推导如下。

不失一般性,我们可以设 $\alpha = 1$。由(6.1.4)和(6.1.6),反线性函数的 MTTF 是一个 i 的线性函数:

$$\Theta(i) = \beta_0 + \beta_1 i \tag{6.1.8}$$

利用下面的近似:

$$\Theta(i) \approx 1/\lambda(t) \tag{6.1.9}$$

和

$$i \approx \mu(t) \tag{6.1.10}$$

将(6.1.9)和(6.1.10)代入(6.1.8),产生出:

$$\frac{1}{\lambda(t)} = \beta_0 + \beta_1 \mu(t) \tag{6.1.11}$$

因为:

$$\lambda(t) = \frac{d\mu(t)}{dt},$$

所以有:

$$[\beta_0 + \beta_1 \mu(t)]\, d\mu(t) = dt.$$

于是可以得到:

$$\beta_0 \mu(t) + \frac{\beta_1}{2}[\mu(t)]^2 = t + C \tag{6.1.12}$$

其中,C 是积分常数。(6.1.12)是 $\mu(t)$的二次方程,解之,得:

$$\mu(t) = \frac{1}{\beta_1} - \beta_0 \pm \sqrt{\beta_0^2 + 2\beta_1(t + C)} \tag{6.1.13}$$

注意: $\mu(t)$是非负函数,且 $\mu(0) = 0$,我们就有: $C = 0$。因此:

$$\mu(t) = \frac{1}{\beta_1} - \beta_0 + \sqrt{\beta_0^2 + 2\beta_1 t} \tag{6.1.14}$$

于是得到:

$$\lambda(t) = \frac{1}{\sqrt{\beta_0^2 + 2\beta_1 t}} \tag{6.1.15}$$

从(6.1.11),我们得到:

$$\lambda(t) = (\beta_0 + \beta_1 \mu)^{-1} \tag{6.1.16}$$

这是 μ的反线性函数,并且因此它是一簇反线性函数的一个模型。利用类似上面的方法,我们很容易得出关于反多项式(二次)的故障强度和 MTTF 的函数形式。

模型的参数反过来应与剩余故障数有关,基于这一概念,Musa 于 1979 年提出:

$$\xi(i) = \frac{v_0 \alpha}{\lambda_0(v_0 - i)} \tag{6.1.17}$$

v_0 是在时间无限远处的故障数的期望值, λ_0 是初始故障强度, α是 Γ-函数的形参, i 是故障数的编号。

Keiller 等人于 1983 年通过研究提出与上述模型相类似的模型。他们的模型的表达式是通过形参导出的,而不是如在(6.1.2)中那样通过标参导出。在(6.1.3),(6.1.4),

(6.1.5)中,分别用 $\alpha(i)$ 代替 α,用 β 代替 $\xi(i)$,就可以得出模型关于 $f(t_i')$, $\Theta(i)$ 和 $Z(t_i')$ 的表达式:

$$f(t_i') = \alpha(i)\left(\frac{\beta}{t_i' + \beta}\right)^{\alpha(i)}\frac{1}{t_i' + \beta},$$

$$\Theta(i) = \frac{\beta}{\alpha(i)},$$

$$Z(t_i') = \frac{\alpha(i)}{t_i' + \beta}.$$

虽然如同上面的模型关于 $\xi(i)$ 的作法一样,对于 $\alpha(i)$ 可以任意选取,但 Keiller 等人仍提出:

$$\alpha(i) = \beta_2 + \beta_3 i \tag{6.1.18}$$

如果我们仍利用(6.1.9)和(6.1.10)的近似,我们得到:

$$\lambda(\mu) = \frac{\beta_2 + \beta_3\mu}{\beta_1} \tag{6.1.19}$$

6.1.2 贝叶斯理论应用于 JM 模型

将贝叶斯理论应用于 JM 模型的研究,关于参数 $\quad$ 和 N_0,应用各种不同的先验分布,可以得到一系列不同的模型。在这一小节里,我们将择其主要者予以介绍。

N. Langberg 和 N. D. Singpurwalla 通过研究,获得了三个不同的软件可靠性模型。最一般的一个模型中用到了下面的先验分布:

$$N_0 \sim F(\,\cdot\,;\vartheta),$$

$$\sim \Gamma(A, B).$$

关于 JM 模型的似然函数是:

$$L(t \mid N_0, \;) = \frac{N(t)!}{[N_0 - N(t)]!}\;^{N(t)}\exp\left[-\sum_{i=1}^{N(t)}(N_0 - i + 1)\tau\right].$$

由上面的先验分布,就得到:

$$h(t, N_0, \;) = L(t \mid N_0, \;)\,\Gamma(A, B) f(N_0).$$

进一步可以得:

$$h(t, No) = \int_0^{\infty} h(t, N_0, \;)\,d$$

或:

$$h(t, N_0) = \frac{A^B}{\Gamma(B)}\,\Gamma(B + N(t))\,\frac{N_0!}{[N_0 - N(t)]!} f(N_0) X(N_0)^{-1},$$

其中,

$$X(N_0) = A + \sum_{i=1}^{N(t)}(N_0 - i + 1)\tau_i^{\,B + N(t)}.$$

边缘分布 $h(t)$ 是:

$$h(t) = \frac{A^B\,\Gamma[B + N(t)]}{\Gamma(B)}\sum_{i=N(t)}^{\infty}\frac{i!\,f(i)}{[i - N(t)]!}X(N_0)^{-1}.$$

$\quad$ 和 N_0 的联合分布在给定 t 的条件下,为:

$$f(\ ,N_0|t_i)=\frac{h(t,N_0,\ )}{h(t)},$$

或者:

$$f(\ ,N_0|t_i)=\frac{^{B+N(t)-1}\exp\ -\ \ A+\sum_{i=1}^{N(t)}(N_0-\ i+\ 1)\ \tau}{\Gamma[B+\ N(t)]}$$

$$\times\ \frac{N_0!}{[N_0-\ N(t)]!}\ \Big/\sum_{i=N(t)}^{\infty}\frac{f(N_0)}{[i-\ N(t)]!}f(i)X(i)^{-1},$$

用 $X(N_0)$ 去乘和除, 就得到:

$$t_i|N_0,t\sim\Gamma\ \ A+\sum_{i=1}^{N(t)}(N_0-\ i+\ 1)\ \tau\ ,B+\ N(t)\ ,$$

$$N_0|t_i\sim f(N_0)\ \frac{N_0}{[N_0-\ N(t)]!}\ \Big/\sum_{i=N(t)}^{\infty}\frac{X(N_0)^{-1}}{[i-\ N(t)]!}f(i)X(i)^{-1}.$$

注意到这样一个事实是有趣的: 在 JM 模型中, 关于 $\ $ 的 Γ 先验分布, 导致关于 $\ $ 的 Γ 后验分布, 而与 N_0 的先验分布无关。

注意本模型的输入是关于 $\ $ 的先验分布的参数和关于 N_0 的先验分布的参数。后者可能是任何形式的分布, 例如: 经验分布。另外, 还需要故障间隔时间作为输入。而模型的输出则是关于 N_0 和 $\ $ 的后验分布。

1985 年, W. S. Jewell 提出: 在 JM 模型中, 每个错误都有故障率 $\ $, 因此似然函数为:

$$L(t|N_0,\ )=\frac{N_0}{N(t)}\ N(t)!\ ^{N(t)}\exp\ -\ \ \sum_{i=1}^{N(t)}\tau+\ N_r(t)t\ \ .$$

如果我们先验地令 $N_0\sim$ 泊松分布(λ), 则得:

$$L(t,N_0|\lambda,\ )=L(t|N_0,\ )\ \frac{\lambda^{N_0}}{N_0!}e^{-\ \lambda},$$

$$L(t,N_0|\lambda,\ )=\frac{1}{N_r(t)!}\ ^{N(t)}e\ -\ \ \sum_{i=1}^{N(t)}\tau+\ N_r(t)t$$

$$\times\ \ \lambda^{N_r(t)+\ N(t)}e^{-\ \lambda},$$

将 λ^{N_0} 重写为 $\lambda^{N_r(t)+\ N(t)}$, 是为了将 $\ $ 和 λ 归并在一起。如果:

$$L(t|\lambda,\ )=(\lambda)^{N(t)}e^{-\ \ \sum_{i=1}^{N(t)}\tau_i+\ \lambda}\sum_{N_r(t)=0}^{\infty}\frac{1}{N_r(t)!}\ \ \lambda e^{-\ t\ N_r(t)},$$

最后给出:

$$L(t|\lambda,\ )=(\lambda)^{N(t)}e^{-\ \ \sum_{i=1}^{N(t)}\tau_i e^{-\ \lambda(1-\ e^{-\ t})}} \tag{6.1.20}$$

如果设 $\ $ 为常数(如 Jelinski-Moranda 所做的那样)以及 $t\gg 1$, 立即可得:

$$L(t|\lambda,\ )=C\lambda^{N(t)}e^{-\ \lambda} \tag{6.1.21}$$

这是一个关于 λ 的 Γ 先验分布。如果设:

$$\lambda\sim\ \Gamma(A,B),$$

由(6.1.21)有:

$$\lambda \mid \theta_t \sim \Gamma(A + N(t), B + 1).$$

$\exp(\lambda e^{-\mu t})$使得检索关于 λ和 μ的先验分布的过程变得复杂化，于是 Jewell 建议采取下面先验地选取的联合分布：

$$\lambda, \theta_t = c \exp(\lambda e^{-\mu t}) \times \Gamma(\lambda \mid A', B') \times \Gamma(\theta_t \mid C', D') \tag{6.1.22}$$

当 $t \to \infty$时，$\exp(\lambda e^{-\mu t}) \to 0$，于是 λ和 μ就独立地符合先验的 Γ-分布。

在一开始，已令 N_0～泊松分布(λ)，对于 μ为常数时的似然函数(6.1.20)就可以写成：

$$L(t \mid \lambda, \mu) = C \lambda^{N(t)} e^{-\lambda(1-e^{-\mu t})}.$$

于是有：

$$N(t) \sim \text{泊松分布}(\lambda(1 - e^{-\mu t})),$$

并且因为 N_0～泊松分布(λ)，我们得到：

$$N_r(t) \sim \text{泊松分布}(\lambda e^{-\mu t}).$$

关于 $N_r(t)$和 λ的联合分布为：

$$f(N_r(t), \lambda) = \frac{\lambda^{A-1} B^A}{\Gamma(A)} e^{-\lambda B} \frac{(\lambda Q)^{N_r(t)}}{N_r(t)!} e^{-\lambda Q} \tag{6.1.23}$$

其中，

$$Q = \exp(-\mu t).$$

$N_r(t)$的边缘分布能直接由(6.1.23)决定：

$$f(N_r(t)) = \binom{A + N_r(t) - 1}{N_r(t)} \left(\frac{B}{B+Q}\right)^A \left(\frac{Q}{B+Q}\right)^{N_r(t)}.$$

这是一个负的二项分布。于是得出后验的判定：

$$E(N_r(t) \mid \theta_t, t) = \frac{A}{B} e^{-\mu t},$$

$$V(N_r(t) \mid \theta_t, t) = \frac{A}{B} e^{-\mu t} \left(1 + \frac{e^{-\mu t}}{B}\right).$$

如果不再假设 μ是常数，我们能够写：

$$h(N_r(t), \lambda, \mu) = f(N_r(t) \mid \lambda, \mu, t) g(\lambda, \mu \mid \theta_t),$$

且：

$$h(N_r(t)) = \int\int \frac{(\lambda Q)^{N_r(t)}}{N_r(t)!} e^{-\lambda} \times k e^{\lambda} \times \frac{\lambda^{A'-1} B^{A'}}{\Gamma(A')} e^{-\lambda B'} \frac{\mu^{D'-1} C^{D'}}{\Gamma(D')} e^{-\mu C'} d\mu d\lambda$$

其中，我们用到了 $g(\lambda, \mu \mid \theta_t)$。上面的积分又产生出：

$$h(N_r(t)) = k \binom{N_r(t) + A' - 1}{N_r(t)} B'^{N_r(t)} \left(\frac{C'}{C' + tN_r(t)}\right)^{D'},$$

其中，k 是一个正规化的常数。我们可以直接看出 $h(0) = k$，因而上式又可以写成：

$$h(N_r(t)) = h(0) \binom{N_r(t) + A' - 1}{N_r(t)} B'^{N_r(t)} \left(\frac{C'}{C' + tN_r(t)}\right)^{D'}.$$

由数值计算可以得出 $h(0)$。但可以更方便地找到 $h(N_r(t)+1)$ 的表达式，也可以更方便地利用上式消去 $h(0)$。这样就给出：

$$\frac{h(N_r(t)+1)}{h(N_r(t))}=\frac{C'+tN_r(t)}{C'+t+tN_r(t)}^{D'}\frac{N_r(t)+A'}{B'}\frac{1}{N_r(t)+1}.$$

可以设 $N_r(t)$ 的众数 N_r^*，是由下式给定的值：

$$\frac{h(N_r(t)+1)}{h(N_r(t))}=1 \tag{6.1.24}$$

因为在连续的情况下它是成立的。Jewell 已经证明真众数是一个大于或等于(6.1.24)的解的整数。因此得：

$$N_r^*+1=\frac{N_r^*+A'}{B'}\frac{C'+tN_r^*}{C'+t+tN_r^*}^{D'}.$$

它可以很容易由标准的数值迭代法来解。

我们可以后验地取 $\lambda\sim\Gamma(A,B)$，并且 将为：

$$\sim\Gamma\left(C+\sum_{i=1}^{N(t)}\tau_i,N_r(t)+D\right).$$

当 λ 为常数时，直接由(6.1.20)和(6.1.22)得：

$$\sim L(t\mid\lambda)\Gamma(C,D)e^{\lambda}.$$

在 JM 模型中，Jelinski 和 Moranda 设每个剩余错误都有相同的故障率 ，因此，整个故障率就是 $[N_0-(i-1)]$，其中有 $i-1$ 个错误已被排除。1980 年，Littlewood 在他的模型中假设每个错误有它自己的发生率 $_i$，且 $\{_i\}$ 是全同独立分布的。Littlewood 使用：

$$\sim\Gamma(A,B).$$

在 i 个错误排除以后，JM 模型规定系统故障率为 (N_0-i)，在 Littlewood 模型中，系统故障率则为：

$$\lambda=\sum_{j=i+1}^{N_0}{}_j \tag{6.1.25}$$

如 JM 模型的作法一样，在 Littlewood 模型中有：

$$\Pr(\text{在区间}[0,t]\text{ 内无错误}\mid)=e^{-\sum_j t},$$

而且，给出每个错误都能被查出的概率：

$$\Pr(\text{错误 }i\text{ 在区间}[0,t]\text{ 内不引起一次故障})=e^{-_i t}.$$

由贝叶斯公式，我们得到：

$$g(_i\mid\text{在区间}[0,t]\text{ 内不存在错误 }i)=\frac{\Pr(\text{在区间}[0,t]\text{ 内不存在错误 }i\mid)g(_i)}{\Pr(\text{在区间}[0,t]\text{ 内不存在错误 }i)},$$

并且因此有：

$$g(_i\mid\text{在区间}[0,t]\text{ 内不存在错误 }i)=\frac{e^{-_i t}g(_i)}{\int_0^{\infty}e^{-_i t}g(_i)d_i}$$

如果在固定长的时间 T_0 内进行测试并观察到 i 个错误，就得到：

$$\lambda_i \sim \Gamma(A, B) \qquad \lambda_i \mid T_0 \sim \Gamma(A, B + T_0),$$

并且,由(6.1.25),有:

$$\lambda \mid T_0, i \sim \Gamma(A(N_0 - i), B + T_0), \tag{6.1.26}$$

其中,λ是当 i 个被观察到的错误在排除之后的当前故障率。观察到的故障率是一种由(6.1.26)所示的 λ的关系。故障间的时间服从指数分布,于是:

$$\tau \sim \int_0^{\infty} f(\tau \mid \lambda) h(\lambda) d\lambda,$$

$$f(\tau) = \int_0^{\infty} \lambda e^{-\lambda\tau} \frac{[(B + T_0)\lambda]^{A(N_0 - i) - 1}(B + T_0)}{\Gamma[A(N_0 - i)]} e^{-\lambda(B + T_0)} d\lambda.$$

这直接给出:

$$f(\tau) = A(N_0 - i) \frac{[(T_0 + B)\lambda]^{A(N_0 - i)}}{(T_0 + \tau + B)^{A(N_0 - i) + 1}}.$$

这是一个 Pareto 分布。更进一步有:

$$F(\tau) = A(N_0 - i)(T_0 + B)^{A(N_0 - i)} \int_0^{\tau} (T_0 + B + u)^{-A(N_0 - i) - 1} du,$$

$$F(\tau) = 1 - \left(\frac{T_0 + B}{T_0 + B + \tau}\right)^{A(N_0 - i)}.$$

其概率为 1- F(τ),当前故障率是:

$$\lambda(t) = -\frac{\ln R(t)}{t},$$

$$\lambda(t) = \frac{A(N_0 - i)}{B + T_0 + t}.$$

当前的 MTTF 为:

$$MTTF_i = \int_0^{\infty} [1 - F(t)] dt,$$

$$MTTF_i = \frac{T_0 + B}{A(N_0 - i) - 1}.$$

对数似然函数为:

$$L = \sum_{i=1}^{N(t)} \{\ln[A(N_0 - i + 1)] + A(N_0 - i + 1)\ln(B + t_{i-1}) - [A(N_0 - i + 1) + 1]\ln(B + t_i)\},$$

于是产生下面的方程:

$$\frac{N(t)}{\hat{A}} = \sum_{i=1}^{N(t)} (\hat{N}_0 - i + 1)\ln\frac{\hat{B} + t_i}{\hat{B} + t_{i-1}} \tag{6.1.27}$$

$$\sum_{i=1}^{N(t)} \hat{A}(\hat{N}_0 - i + 1)\left(\frac{1}{\hat{B} + t_{i-1}} - \frac{1}{\hat{B} + t_i}\right) = \sum_{i=1}^{N(t)} \frac{1}{\hat{B} + t_i} \tag{6.1.28}$$

$$\sum_{i=1}^{N(t)} \frac{1}{\hat{N}_0 - i + 1} = \hat{A} \sum_{i=1}^{N(t)} \ln\frac{\hat{B} + t_i}{\hat{B} + t_{i-1}} \tag{6.1.29}$$

如果用 τ 代替 ln((B+ t_i)/(B+ t_{i-1})),我们就得到 JM 模型的方程式:

$$B + t_i = B + t_{i-1} + \tau,$$

因此,当 $B+t_{i-1}\gg\tau$,就可以得到:

$$\ln\frac{B+t_{i-1}+\tau}{B+t_{i-1}}\approx\frac{1}{B}\tau,$$

并且(6.1.27)和(6.1.29)就转换成:

$$\frac{N(t)}{\hat{A}}=\frac{1}{\hat{B}}\sum_{i=1}^{N(t)}(\hat{N}_0-i+1)\tau \tag{6.1.30}$$

$$\sum_{i=1}^{N(t)}\frac{1}{\hat{N}_0-i+1}=\frac{\hat{A}}{\hat{B}}\sum_{i=1}^{N(t)}\tau \tag{6.1.31}$$

令 $\lambda=A/B$,相当于令 $\lambda_i=E(\lambda)$,λ 的分布由:

$$\lambda_i\sim\Gamma(A,B)\qquad\lambda_i\mid T_0\sim\Gamma(A,B+T_0)$$

而定,则 Littlewood 的贝叶斯模型就退化成 JM 模型。

Littlewood 的这一模型是不是贝叶斯模型,对这一问题是有争议的。Littlewood 曾以另一种形式发表了这一模型,而没有提及贝叶斯框架。关于这一问题的纯粹的贝叶斯公式应该是:

- 我们有一个常数 λ,而当前关于它的信息是由一个带有参数 A 和 B 的 Γ-分布表示的。
- 在这种情况下,象 Littlewood 所做的那样,然后来估计 A 和 B 的作法是不合理的。

但是这一问题的 Littlewood 公式似乎又是:每个错误都有它自己的暴露率 λ_i,这些 λ_i 都满足带有参数 A 和 B 的 Γ- 分布。

Littlewood 的这一模型不是贝叶斯模型,但是可以归入 Weiss 的泊松模型框架中去。在这两种情况下的数学公式是一样的。

6.1.3 关于贝叶斯估计的评价

在应用贝叶斯理论时,有几点情况是应引起我们注意的,这就是:

- 无信息先验分布所得出的估计结果可以与最大似然估计的结果一样好,甚至比最大似然估计的结果还要好。
- 不好的先验分布所产生的估计结果会比最大似然估计的结果还要差。
- 如果模型本身或早期的经验显示出参数之间具有某种关系,那么一个好的先验分布必定会将这种关系考虑进去。

在进行软件可靠性估计时,我们一定要将上面的这些情况考虑进去,因此它们事实上要影响到我们对先验分布的选择。关于如何选择先验分布,我们可以举出下面两点建议:

- 在对有关因素,比如:操作运行环境等,没有任何可资利用的信息时,通常较好的作

法是采用无信息先验分布,而避免采用不适当的信息或错误的信息以选择先验分布。因为这样以不适当的甚至错误的信息选择的先验分布,往往是不好的。

· 如果我们有理由相信问题域相同,而软件的执行频度和路径不相同,则我们可以使用关于 N_0 的先验分布,而关于其它分布参数仍应使用无信息先验分布。

另外,关于故障时间的分布参数变化幅度比每千行代码中的错误个数的变化幅度要小。其理由似乎至少部分地因为人类的共性,使得执行频度和执行路径相对地稳定所致。由此出发,除非我们有很好的理由相信错误个数与代码行数之比与以前的情况是一样的,我们才应该使用关于 N_0 的无信息先验分布和关于其它分布参数的经验先验分布。

§ 6.2 Seeding 模型

种子撒播/加标记(Seeding/Tagging)法,用于估计动物群体或鱼群中的个体数,已使用多年。首先由 H. D. Mills 建议将 Seeding 模型用于估计软件中的错误个数;由 M Hyman 提议用 Tagging 法估计软件中的错误个数。

Seeding 和 Tagging 法是两种作法略有不同的估计过程,其不同之处仅在于"放回"的过程,但实质从数理统计的角度看,它们是彼此等价的。

将同一程序交两个排错员单独排错,他们彼此不作任何通讯。查出的错误要求记录并纠正。经过一段相同的时间,他们的记录不可能完全相同,有共同的部分,也有不同的部分,这是 Tagging 法的作法。

将一程序,人为地加入若干个错,然后交一排错员排错,但加入的错误他事先不知道。在排错之后的记录中,人为加入的错误很可能只被查出一部分。这就是 Seeding 法的作法。

在下面的讨论中,我们假设不引入新错。

构成估计最简模型的基本假设是:排错过程完全随机,即所有的软件错误都是无特征的。每一个都有相同的发生率,因而查明它们任一个的概率都相等,为 $\frac{1}{N}$(N 为软件中初始总错误个数)。在下面的讨论中,为方便计,设 $N = 100$,由第一个排错者(加标记者)查出 t 个错(设 $t = 20$),于是共有 $\frac{t}{N} = 1/5$ 的错误被加上了标记。第二个排错者(采样者)查出 s 个错(设 $s = 25$)。由等概率假设,有:

$$\frac{c}{s} \approx \frac{t}{N},$$

其中,c 是在样本中被发现带有标记的错误个数。

因为 t, s, c 均已知,所以有:

$$N = \frac{st}{c}.$$

由于错误个数不可能为分数,在实际计算时一般都取 N 的向上取整值。如: c= 6, 则:

$$N = \frac{25 \times 20}{6} = 84.$$

这一估计值与我们开始假设的 N= 100 有较大的估计误差。

以上是对于加标记的估计过程,对于种子撒播的过程,可以讨论如下:设初始错误个数为 80, 记: N_x= 80, t= 20 是人为加入的错误个数,于是有: N= N_x+ t= 100, 依上述方法,得 N 的估计值 N= 84。这时,对于 N_x 的估计值应有:

$$N_X = N - t = 64.$$

对于 c= 4, 5, 6, 可分别得出 N 的估计值为 125, 100, 84, 在它们中间再不会有其它中间值出现。这是由于对 c 所加的整数约束,所以可能出现较大的整数误差。

下面讨论 N 的最大似然估计 N_0。s 和 t 是已知参数, c 可由试验得出,则:

$$N_0 = \frac{st}{c}.$$

在 c= 0 时, N_0 不存在,特补充定义如下:

$$N_0 = \begin{cases} \dfrac{st}{c}, & \text{如果 } c > 0; \\ 2st, & \text{如果 } c = 0. \end{cases}$$

修改的最大似然估计 N_1 定义为:

$$N_1 = \frac{(s+1)(t+1)}{c+1} - 1.$$

对于足够大的样本,估计会更精确。极端的情况则是: s= t= N, 这时能得出完全的估计。对于相对小的样本,估计结果就有问题。为提高估计精度,下面讨论多次试验估计法。设使用 m(m≥3)个排错者两两结合来测试同一个软件,每一对得的结果都是独立的。于是, m 个排错者将得出 n= m(m- 1)/2 个可能的数据。然后对它们进行综合,以得出一个新的、可以减小整数误差的估计值。其具体的作法可以是:

- 对每一个 c 值,使用任何单次试验的方法,求出 N 的估计值,对 n 个 N 的估计值取均值,作为最后估计值。
- 另一种可供选择的作法是:先对 n 个 c 值取均值,得:

$$c = \frac{1}{n}\sum_{i=1}^{n} c_i,$$

然后以 c 代替 c,再由任一单次试验的估计公式求出 N 的估计值。

§ 6.3 基于输入域的模型

作为软件可靠性的数学理论之一, Nelson 模型的开发,有着以下三个方面的目的:

· 给软件可靠性的基本要素提供明确的数学定义;

· 推导出基本要素间的数学关系;

· 进一步寻求新的数学方法。

Nelson 模型的基本概念如下:

① 一个计算机程序 P 可以定义为在集合 E 上的可计算函数 F, 其中:

$$E = \{E_i \mid i = 1, 2, \cdots, N\}$$

且:

$$E_i \cap E_j = \quad (i \neq j);$$

② 对于每一输入 E_i, P 的执行产生函数值 $F(E_i)$;

③ 集合 E 定义了全部 P 所能够进行的计算;

④ 由于 P 在实现时的欠缺, P 实际上只定义了一个函数 F′, 它不同于 F;

⑤ 对于某些输入 E_i, P 的实际执行所产生的输出 $F'(E_i)$ 与 $F(E_i)$ 存在允许的误差 Δi, 也即 $|F'(E_i) - F(E_i)| \leq \Delta i$;

⑥ 对于 $E - \{E_i\}$, 它构成 E_e, 在它上面, P 的执行产生的输出超出了可接受的范围, 也即是指: 或者, $|F'(E_i) - F(E_i)| > \Delta i$, $E_j \in E_e$; 或者, 执行过早地终止; 或者, 执行根本不终止(如死循环的发生, 等)。凡出现这些异常, 都称为“执行故障”。

将 E_i 提交给 P 去执行以产生 $F'(E_i)$ 或出现执行故障的过程, 称为 P 的一次运行。但 E_i 的全部值并不一定同时提交给 P。因此, P 的一次运行将导致一次执行故障的概率与在一次运行中所使用的 E_i 是从 E_e 中选取的概率相等, 即:

$$p = \frac{n_e}{N},$$

其中, n 次运行的概率为:

$$R_1(n) = R_1^n = (1 - p)^n = \left[\sum_{i=1}^{N} p_i(1 - y_i)\right]^n.$$

但是, 事实上的情况还要复杂。提供给 n 次运行的输入并不是独立地选取的, 而是按照某一预先排好的顺序选出的确定的序列, 如: 实时系统中按值的递增排列的序列, 或由某个实际的输入来判定的序列, 等等。因此, 操作轮廓不再是上面的 $\{p_i\}$, 而要重新定义选取概率 p_{ji} 如下:

p_{ji} 为选取 E_i 作为在一运行序列中的第 j 次运行的输入的概率。

于是, 第 j 次运行导致一次执行故障的概率 p_j 就是:

$$p_j = \sum_{i=1}^{N} p_{ji} y_i.$$

这样, 在一个 n 次运行的序列中不出现执行故障的概率, 即 P 的可靠性 R(n) 定义如下:

$$R(n) = (1 - p_1)(1 - p_2)\cdots(1 - p_n)$$

$$= \prod_{j=1}^{n}(1 - p_j).$$

可以将上式改写成指数形式:

$$R(n) = \exp\left[\sum_{j=1}^{n} \ln(1 - p_j)\right].$$

$R(n)$ 的某些性质可以由取近似的方法表示如下:

① 对于 p_j n 1, 有:

$$R(n) = \exp\left[-\sum_{j=1}^{n} P_j\right].$$

② 如果对于所有的 j 有: $p_j = p_c$ (为一常数), 则:

$$R(n) = \exp(-p_c ¡¤ n).$$

③ $R(n)$ 还可以经过变换, 改用执行时间来表示。设 Δt_j 表示第 j 次运行的执行时间, 则:

$$t_j = \sum_{i=1}^{j} \Delta t_j$$

表示从执行开始, 一直到第 j 次运行的累积执行时间, 因此, 令:

$$h(t_j) = -\frac{\ln(1 - p_j)}{\Delta t_j},$$

则有:

$$R(n) = \exp\left[\sum_{j=1}^{n} \ln(1 - p_j)\right]$$

$$= \exp\left[-\sum_{j=1}^{n} \Delta t_j h(t_j)\right].$$

当 n 变得很大时, 如 $\Delta t_j \to 0$, 则上式可以写成一个积分式, 从而导出:

$$R(t) = \exp\left[-\int_0^t h(s)\,ds\right].$$

在 p_j n 1 时, $h(t_j)$ 可以解释为风险函数, 因此, Nelson 模型在本质上与可靠性理论的数学表示是完全一致的。

n_e 是在 E_e 中的 E_i 的个数, 故此,

$$R_0 = 1 - p = 1 - \frac{n_e}{N},$$

其中, p 是 P 在从 E 中随机选取的 E_i 上运行, 且将导致一次执行故障的概率; R_0 是 P 在从 E 中随机选取的 E_i 上运行, 且将产生出可接受的输出的概率。

但在实际的工作过程中, 用于运算的输入也可能根据某一特定的要求从 E 中加以选择, 这样就比上面讨论的情况要复杂。这一特定的要求可以用概率分布 p_i 来描述: p_i 是 E_i 从 E 中被选中的概率, p_i 的集合称为“操作剖面”(Operational Profile)。为了计算 p, 定义“执行变量”(Execution Variable) y_i 如下:

$$y_i = \begin{cases} 0, & \text{如 P 在 } E_i \text{ 上的一次运行计算出一个可接受的函数值;} \\ 1, & \text{如 P 在 } E_i \text{ 上的一次运行产生一次执行故障。} \end{cases}$$

于是有:

$$p = \sum_{i=1}^{N} p_i y_i,$$

p 即表示 P 在根据概率分布 p_i 来选取的 E_i 上的一次运行产生一次执行故障的概率，并且有：

$$R_1 = 1 - p = 1 - \sum_{i=1}^{N} p_i y_i$$

$$= \sum_{i=1}^{N} p_i - \sum_{i=1}^{N} p_i y_i$$

$$= \sum_{i=1}^{N} p_i (1 - y_i).$$

R_1 表示 P 在依概率分布 p_i 来选取的 E_i 上一次运行导致一次正确执行的概率。于是根据概率分布 p_i，彼此独立地选取 E_i，不产生执行故障。

对于 P 的可靠性估计，可以让它在具有 n 个输入的样本上运行，并用下述公式计算出 R 的估计值 R：

$$R = 1 - \frac{n_e}{n},$$

其中，$\hat{n}_e$是在 n 次运行中，产生执行故障的次数。

如果该样本中的 n 个输入是依概率分布 p_i 随机地从 E 中选取的，则在“对于 p n 1，在样本概率分布上的 R 的期望值就等于 R”这一意义上来说，“R 是 R 的一个恰当的估计”这一断言可以由引入“样本变量”z_{ij}来证明。

$$z_{ij} = \begin{cases} 1, & \text{如果 } E_i \text{ 在样本 } S_j \text{ 中;} \\ 0 & \text{否则。} \end{cases}$$

在样本 S_j 中，获取互异的输入 n_j 的个数可以小于 n，这是因为同一个 E_i 按随机过程，能不止一次地被选中，因此有：

$$\sum_{i=1}^{N} z_{ij} = n_j.$$

但在绝大多数情况下，可能的输入数 N 比样本的大小要大得多，以至于“任何一个输入都可能重复出现于样本之中”是一个不可靠的假设。如果用 q_j 表示样本 S_j 被选中的概率，M 表示可能的样本个数，则有：

$$\sum_{j=1}^{M} z_{ij} q_j = 1 - (1 - p_i)^n.$$

由于 R 能改写成下面的形式：

$$R_j = \frac{1}{n} \sum_{i=1}^{N} (1 - y_i) z_{ij},$$

于是 R 的期望值 $E(R)$ 就是：

$$E(R) = \sum_{j=1}^{M} q_j R_j$$

$$= \frac{1}{n} \sum_{j=1}^{M} q_j \sum_{i=1}^{N} (1 - y_i) z_{ij}$$

$$= \frac{1}{n}\sum_{i=1}^{N}(1-y_i)\sum_{j=1}^{M}q_j z_{ij}$$

$$= \frac{1}{n}\sum_{i=1}^{N}(1-y_i)\ 1-(1-p_i)^n$$

$$= \sum_{i=1}^{N}(1-y_i)p_i \qquad (\text{对于 } p_i n\ 1)$$

$$= 1-p = R.$$

为了估计 R 的值,操作轮廓$\{p_i\}$的估计就是关键。在实际的估计过程中,可以按下面的方法去做:

① 将输入变量空间划分成 N 个子空间,并以对实际的输入出现的估计为基础,给“某个输入将被从每个子空间中选中”这一事件指定概率,即 p_i 的设定。

② 依 p_i,选取一个有 n 个输入的样本 S,比如,可用随机数发生器来帮助进行选取。

③ 将 P 在 S 上运行 n 次,则必定会出现有的输出对于某些输入是正确的,而对于其它一些输入则会导致执行故障的现象。

④ 在每次执行故障发生时,都不停止运行,不排错,只收集估测数据。

⑤ 待 n 次运行完毕后,由收集到的数据,按公式:

$$R = 1-\frac{\hat{n}_e}{n}.$$

计算出估计值。

从上面的叙述可知,应用 Nelson 模型于软件可靠性的测量,只是一种估计值,只能反映出我们对程序正确运行的信任程度,也即只能反映出我们对程序正确运行的信心。对于这种可靠性,有时又将它称之为程序在实际运行中表现出来的可靠性。

参照 TRW 的试验方式,按照 Nelson 模型的要求,我们曾对我所开发的软件工具系统 WPADT,在系统测试的前阶段(开始不久)应用 Nelson 模型,进行了一次可靠性估计的试验。具体的做法是:根据 WPADT 的具体情况,对它的输入进行了分析,并设计出了划分输入空间的原则。顺便提一下,这一划分原则,对于 Nelson 模型来说,并未具体硬性规定,可以根据具体情况而定,比如说,这一原则可以是功能的,也可以是程序的运行路径等。因为 WPADT 是用于开发软件的工具环境,它的输入是许多对于功能的要求,于是我们设计的划分原则是以功能为主而辅之以程序的运行路径的划分。划分原则定下来以后,我们设计了许多的 test cases,将它们编号,然后在随机数发生器的辅助下,选取了 1000 个 test cases,然后使 WPADT 系统在这 1000 个 test cases 上运行,并记录产生运行故障的次数,具体结果如下:

软件系统名称	n	$\hat{n}_e$	R
WPADT	1000	35	0.965

这一估计值是在系统测试开始不久得出的,我们相信,随着系统测试的不断深入,R 的值将不断上升。

Nelson 模型有着坚实的理论基础,这是其它许多模型无法比的。但是,它也有着许多实际应用上的缺陷:

- 为了获得更高的估计准确度,它需要用到大量的测试数据;
- 它未考虑输入域的连续性问题;
- 对输入域的随机采样限制了它的可应用范围,也使它不能使用很多行之有效的测试策略,如:它不能使用边界值测试方法,它在实时控制系统中就无法使用,因为实时控制系统的许多输入物理量都是连续变化的(温度、电压、电流强度,等等);
- 它不考虑软件复杂性测度。

它的这些问题,都被其它基于输入域的模型克服。

§ 6.4 其它一些方法

除了前面介绍的各类软件可靠性模型以外,还有许多方法,如:非参数分析方法、结构化软件可靠性模型、Cox 的风险函数模型,以及时间序列分析方法,等。在这一节中,我们将逐个对它们进行讨论。

6.4.1 非参数分析

均值函数 $m(t)$ 和故障率函数 $\lambda(t)$,完全单调的事实,产生出一种无需对它们假设一个解析式的分析法——非参数分析方法。

函数 $\lambda(t)$ 完全单调的,当且仅当它具有的所有阶微商有性质:

$$(-1)^n \frac{d^n \lambda(t)}{dt^n} \geq 0, \quad t \geq 0, n \geq 0.$$

完全单调的函数类能被证明为与指数类次序统计量的模型的强度函数总体恒同。因此,这一类函数为估计故障类的非参数方法提供了自然的基础。在某些情况下,重要的是找到一个与故障数据能良好拟合的完全单调函数。

设数据被分为 v 个等长为 u 的时间区间。如果 t_i 是直到第 i 个区间开始的时间,则在每个区间 i 中的故障率 λ_i 可以估计如下:

$$\lambda_i = \frac{c(t_i) - c(t_{i-1})}{u}, \quad 1 \leq i \leq v,$$

其中,$c(t_i)$ 与 $c(t_{i-1})$ 分别表示在时刻 t_i 与 t_{i-1} 时的累积故障数。

事实上,完全单调的函数将被定义为一个单调下降的故障率序列,λ_i 为使这一序列拟合未加工的原始数据,各阶(j)的反向差分算子 D_j,定义如下:

$$D_0(\lambda_i) = \lambda_i,$$
$$D_1(\lambda_i) = \lambda_i - \lambda_{i-1},$$
$$\vdots$$
$$D_j(\lambda_i) = D_{j-1}(\lambda_i) - D_{j-1}(\lambda_{i-1}).$$

“良好”拟合的定义就是对在所有区间上的均方差$(\lambda_i - \lambda_i)^2$之和求极小值。则问题在于找到估计故障率 λ_i的序列,根据单调性,强制对所有的 j 直到某最大度数 d,使之满足:

$$(-1)^j D_j(\lambda_i) \geq 0.$$

而最优化是一个线性约束二次规划问题。

当 d = 1 时,问题化为等张回归问题,这可以通过在故障数据图上画包络来解决。用类似的方法,则可化为上面讨论的预分析问题。如果最后的内部故障时间正好产生于内部故障分布的右尾部,λ_i将低估 $\lambda(t_i)$,在观察阶段末尾的“真实”故障率,而对于 λ_i的单调性限制将不起作用,因此产生负偏差。

在大多数软件可靠性应用中,正偏差更好些。如犯了悲观的错误,则一个负偏估计会更好些。因此更高阶的限制是吸引人的,且实际上,d = 2 或 d = 3 的情形下已被找到,并产生出更低偏差的估计结果。

本方法只估计当前可靠性,但它为参数模型的行为提供有用的校验手段。

A. Sofer 和 D. R. Miller 开发出一个非参数软件可靠性增长模型。他们设:软件执行一段时间 T,在区间[0, T]内查出 n 个错,并全部被排除(完全排错)。故障出现的时间为:$0 < t_1 < t_2 < \cdots < t_n < T$。这样,软件可靠性将得到不断提高。

根据大量的研究可知,在非常一般的条件下,如果软件的使用是随机的、与时间齐次的,且排错是及时且完全的,则软件可靠性增长过程,就有一个完全单调的强度函数。反之,实际上完全单调的函数可用以描述软件可靠性的增长过程的故障率。

下面讨论如何找出一个完全单调的故障率函数,以及与它相联系的均值函数(如果它存在的话)。该均值函数可能不直接满足完全单调性,然而 M(t)是一个非负函数,且它的导数 dM(t)/dt 是一个完全单调的函数。方法就是从数据出发,对所要求的函数先作出一个初始的粗略估计,然后在使用最小二乘法的情况下,用最靠近该初始估计的一个完全单调的函数作拟合,以改进这个估计的函数。为此,作一个初始的粗略估计函数,M(t):

$$M(t) = \begin{cases} i + (t - t_i)/(t_{i+1} - t_i) & t_i \leq t \leq t_{i+1}, \quad i = 0, 1, 2, \cdots, n-1. \\ n + \delta(t - t_n)/(T - t_n), & t_n \leq t \leq T \end{cases}$$

这是一个分段线性估计函数,断点为 t_i。其中,δ是调整系数,它的取值带有几分随意性,如取更大的值将倾向于给出更守恒的估计,一般可考虑取 δ = 0.0, 0.5, 1.0 等,也可取其它值。在实践中,取 θ = T/k,将[0, T]分为 k 个等长区间,并定义 $s_i = i\theta$ i = 0, 1, ⋯, k。因此,$m_i = m(s_i)$就是在定长区间(长为 s_i)内的均值函数的函数值的初始估计。但一般它不能满足完全单调性,因而需作适当修正。

为估计故障率函数,取:

$$r_i = (m_i - m_{i-1})/\theta \quad i = 1, 2, \cdots, k$$

作为在点 s_i 上的故障率的粗略估计。

当估计在离散、等长的时间区间上进行时,一个完全单调的函数的模拟,就是一个完

全单调的序列。当$(-1)^j\Delta^j r_i \geq 0, j+1 \leq i, j=0,1,\cdots$成立时，则序列$(r_i, i=1,2,\cdots)$是完全单调的。其中$\Delta^j$是的反向差分算子(即$D_j$)：

$$\Delta^0 r_i = r_i,\ \Delta^1 r_i = r_i - r_{i-1},\ \cdots,\ \Delta^j r_i = \Delta^{j-1} r_i - \Delta^{j-1} r_{i-1},\quad (j>1) \tag{6.4.1}$$

上已述及，一般初始估计$(r_1,\cdots,r_k)$并不具完全单调性。目的在于找到"最靠近"的完全单调序列$(r_1,\cdots,r_k)$，并以此作为在时间s_i上故障率的估计。依加权最小二乘的标准，问题就在于找到一个矢量r，使下式取极小：

$$D(r,r) = \sum_{i=1}^{k} w_i (r_i - r_i)^2 \tag{6.4.2}$$

并符合反向差分算子定义中的完全单调的限制条件。其中，w_i为一个预先指定的权值集合中的元素。一般要将(6.4.1)中的控制条件适当放宽，至多考虑到d阶导数(d不取∞)，如取$d=3$或$d=4$为好。于是规定限制条件如下：

$$\begin{aligned} &(-1)^d \Delta^d r_i \geq 0, \qquad d+1 \leq i \leq k+l \\ &(-1)^j \Delta^j r_{k+1} \geq 0, \quad 0 \leq j \leq d-1 \end{aligned} \tag{6.4.3}$$

于是问题化成对(6.4.2)式取极小并满足(6.4.3)的限制条件：

$$\min\{D(m,m)\} = \min \sum_{i=1}^{k} w_i (m_i - m_i)^2 \text{，且满足：}$$

$$\begin{aligned} &(-1)^{d+1} \Delta^d m_i \geq 0, \qquad d \leq i \leq k+l \\ &(-1)^j \Delta^j m_{k+1} \geq 0, \qquad 0 \leq j \leq d-1 \\ &m_k \equiv n + \delta, k > 0 \\ &m_0 \equiv 0 \end{aligned} \tag{6.4.4}$$

如果测试终止于一次故障，$(t_n = T)$，则取$\delta = 0$。为缩短测试过程，取$\delta = 0.5$较合适。根据泊松过程的假设，取$\delta = 1$也是一个可行的选择。

下面讨论预测问题。当$l>0$，(6.4.4)式要求大量的计算。为求解二次规划问题，通常要求目标函数的海赛矩阵是正定的。然而，(6.4.2)式的海赛矩阵仅是半正定的，且一定存在奇异性。其结果是在求解时，不仅有大量困难，且其优解必不唯一。实际上，任何两个前面k个分量相等的解向量，都会产生确定的相同目标值。即，如完全单调序列$(r_1,\cdots,r_k)$能在后面外插l个时间区间，且通过某种方式使序列$(r_1,\cdots,r_k,r_{k+1},\cdots,r_{k+l})$为完全单调的，则所有这些可能的外插都具有相同的最小二乘目标。可证，在所有这些外插中，存在一个全局最高与一个全局最低的外插，使得所有其它在以后的、完全单调的外插，一定处在它们中间。因此可以获得一个包络，将所有其它的、完全单调的外插都限制在其中。

考虑d阶的完全单调序列：$R=(r_1,\cdots,r_k)$，如果序列$(r_1,\cdots,r_k,r_{k+1},\cdots,r_{k+l})$是直到$d$阶的完全单调序列，即它满足(6.4.3)式，则$(r_{k+1},\cdots,r_{k+l})$为$R$的一个可行的完全单调的$d$阶外插。又如果任一其它这样的外插$(r_{k+1},\cdots,r_{k+l})$，满足$r_{k+i} \geq r_{k+i}, (i=1,\cdots,l)$，则外插$(r_{k+1},\cdots,r_{k+l})$构成所有可容许的$d$阶外插的一个下界。同样，如$r_{k+i} \leq r_{k+i}, (i=1,\cdots,l)$，则它构成它们的一个上界。下面讨论对于完全单调的外插，它的上、下包络存在的条件。

对于$d=1$与$d=2$，序列$(r_1,\cdots,r_k)$可作这样的外插：令$r_{k+i}=r_k, (i=1,\cdots,l)$。显然，它是1阶和2阶的各所有完全单调外插的可容许的上包络。另外，对于$d=1$，作外插：r_{k+i}

$= 0, (i = 1, \cdots, l)$, 它为保序的所有完全单调外插的下包络。下面的命题说明: 对于可容许的外插($d = 2$)的下包络, 是沿着分段线性函数(斜率为 $\Delta^1 r_k$), 直到零为止, 并在此之后, 就以零为其函数值的。定义:

$$p = \begin{cases} \mathrm{gilb}(-r_k / \Delta^1 r_k), & \Delta^1 r_k > 0 \\ l, & \Delta^1 r_k = 0 \end{cases} \tag{6.4.5}$$

命题 1. 如 $d = 2, l > 0$, 且$(r_1, \cdots, r_k)$为(6.4.3)式取 $l = 0$ 时的一个可容许的解, 则外插:

$$r_{k+i} = \begin{cases} r_k + i\Delta^1 r_k, & i = 1, \cdots, p \\ 0, & i = p+1, \cdots, l \end{cases}$$

是所有可容许的二阶外插的下包络。 (证明略。)

命题 2. 如 $d = 3, l > 0$, 且$(r_1, \cdots, r_k)$为(6.4.3)式取 $l = 0$ 时的一个可容许的解, 则作出的外插仍满足(6.4.3)式的充要条件是:

$$r_k + j\Delta^1 r_k + \frac{1}{2} j(j+1)\Delta^2 r_k \geq 0, \quad j = 1, \cdots, l \tag{6.4.6}$$

另外设:

$$q = \begin{cases} \mathrm{gilb}(-\Delta^1 r_k / \Delta^2 r_k), & \Delta^2 r_k > 0 \\ l, & \Delta^2 r_k = 0 \end{cases} \tag{6.4.7}$$

于是, 对于 $d = 3$ 的所有可容许外插的上包络是:

$$r_{k+i} = \begin{cases} r_k + i\Delta^1 r_k + \frac{1}{2} i(i+1)\Delta^2 r_k, & i = 1, \cdots, q \\ r_{k+q}, & i = q+1, \cdots, l \end{cases}$$

(证明略。)

命题 3. 如 $d = 3, l > 0$, 且$(r_1, \cdots, r_k)$为(6.4.3)式取 $l = 0$ 时的解, 并满足(6.4.6)式, 且 p 的定义为(6.4.5)式:

(a) 如 $p \geq l$, 则外插 $r_{k+i} = r_k + i\Delta^1 r_k$, $i = 1, \cdots, p$, 为$(r_1, \cdots, r_k)$的三阶所有可容许的外插的一个下包络。

(b) 如 $p \leq l$, 设 $u = \min(l, 1 + \mathrm{gilb}(-2r_k / \Delta^1 r_k))$, 则外插:

$$r_{k+i} = \begin{cases} r_k + i\Delta^1 r_k + \frac{1}{2} i(i+1)\left[-2(r_k + u\Delta^1 r_k)\right] / \left[u(u+1)\right], & i = 1, \cdots, u \\ 0, & i = u+1, \cdots, l \end{cases}$$

为$(r_1, \cdots, r_k)$的三阶所有可容许的外插的一个下包络。 (证明略。)

命题 3 说明最小包络是一个斜率为 $\Delta^1 r_k$ 的线性函数, 它给出的线性函数是可容许的(非负的); 另外说明二次函数具有为常数的二阶差分:

$$a = -\frac{2(r_k + u\Delta^1 r_k)}{u(u+1)}$$

在 r_{k+u} 处趋于零, 且从此以后以零为其值。

命题 4. 如果 $d = 4, l > 0$, 且$(r_1, \cdots, r_k)$为满足(6.4.3)式, 取 $l = 0$ 的解, 能被外插成的向量

$(r_1, \cdots, r_k, r_{k+1}, \cdots, r_{k+l})$在 $l > 0$ 时，满足(6.4.3)式的充要条件是：

$$\Delta^1 r_k + j\Delta^2 r_k + \frac{1}{2}j(j+1)\Delta^3 r_k \leq 0, \quad (j = 1, \cdots, l) \tag{6.4.8}$$

$$r_k + l\Delta^1 r_k + \frac{1}{2}l(l+1)\Delta^2 r_k \geq 0 \tag{6.4.9}$$

$$r_k + \frac{2}{3}(j-1)\Delta^1 r_k + \frac{1}{6}j(j-1)\Delta^2 r_k \geq 0, \quad (j = 1, \cdots, l) \tag{6.4.10}$$

如果 $\Delta^1 r_k + l\Delta^2 r_k \leq 0$，则所有这些外插的上包络为：

$$r_{k+i} = r_k + i\Delta^1 r_k + \frac{1}{2}i(i+1)\Delta^2 r_k, \quad (i = 1, \cdots, p) \tag{6.4.11}$$

否则，设：

$$v = \min(l, 1 + \mathrm{gilb}(-2\Delta^1 r_k / \Delta^2 r_k)). \tag{6.4.12}$$

则所有这些外插的上包络为：

$$r_{k+i} = \begin{cases} r_k + i\Delta^1 r_k + \frac{1}{2}i(i+1)\Delta^2 r_k + \frac{1}{6}i(i+1)(i+2)\dfrac{-2(\Delta^1 r_k + v\Delta^2 r_k)}{v(v+1)} & (i = 1, \cdots, v) \\ r_{k+v}, & (i = v+1, \cdots, l) \end{cases}$$

（证明略。）

下面讨论关于均值函数的预测的包络。考虑 d 阶的一个序列：$M = (m_1, \cdots, m_k)$，它满足(6.4.4)式。如果序列$(m_1, \cdots, m_k, m_{k+1}, \cdots, m_{k+l})$满足(6.4.4)式，则序列$(m_{k+1}, \cdots, m_{k+l})$定义为对$(m_1, \cdots, m_k)$的一个可容许的外插。另外，如果任何别的这样一个外插$(\overline{m_{k+1}}, \cdots, \overline{m_{k+l}})$满足$\overline{m_{k+i}} \leq m_{k+i}$ $(i = 1, \cdots, l)$，则这样一个外插对于所有可容许的 d 阶外插而言，构成一个上界；类似地，如$\overline{m_{k+i}} \geq m_{k+i}$，$(i = 1, \cdots, l)$，它就构成一个下界。

然后，讨论关于序列 M 的所有可容许外插的上、下包络存在的条件。均值函数的导数是完全单调的，因此，关于$(m_1, \cdots, m_k)$的所有可容许的 d 阶外插的上、下界，可以由分别对它们合并所有可容许的 d-1 阶外插的上、下界来获取。

作为结果，对于 $d = 1, 2, 3$，序列$(m_1, \cdots, m_k)$总可以被外插的。对于从一阶到三阶的所有可容许的外插的上包络是一线性函数：

$$m_{k+i} = m_k + i\Delta^1 m_k.$$

对于所有可容许的一阶和二阶的外插 $m_{k+i} = m_k$，显然是所有可容许的外插的下包络。下面的命题 5 说明，对于三阶的可容许的外插的下包络，是沿着一条逐渐减少至一常数的二次函数。

命题 5. 当 $d = 3, l > 0$ 时，且$(m_1, \cdots, m_k)$是一个在 $l = 0$ 时，满足(6.4.4)式的可容许解。

设：

$$p = \begin{cases} \mathrm{gilb}(-\Delta^1 m_k / \Delta^2 m_k), & \Delta^2 m_k > 0 \\ l, & \Delta^2 m_k = 0 \end{cases}$$

则外插：

$$m_{k+i}=\begin{cases} m_k+i\Delta^1 m_k+\dfrac{1}{2}i(i+1)\Delta^2 m_k, & (i=1,\cdots,p)\\ m_{k+p}, & (i=p+1,\cdots,l)\end{cases}$$

为$(m_1,\cdots,m_k)$的所有三阶可容许的外插的下包络。

关于它的证明，利用命题 1 即可。

命题 6. 当 $d=4, l>0$ 时，且解$(m_1,\cdots,m_k)$在 $l=0$ 时满足(6.4.4)式，能外插为解$(m_1,\cdots,m_k,m_{k+1},\cdots,m_{k+l})$在 $l>0$ 时满足(6.4.4)式的充要条件是：

$$\Delta^1 m_k+j\Delta^2 m_k+\frac{1}{2}j(j+1)\Delta^3 m_k\geq 0,\quad (j=1,\cdots,l)\qquad (6.4.13)$$

设：

$$q=\begin{cases}\text{gilb}(-\Delta^2 m_k/\Delta^3 m_k), & \Delta^3 m_k>0\\ l, & \Delta^3 m_k=0\end{cases}$$

$$\alpha=\Delta^1 m_k+q\Delta^2 m_k+\frac{1}{2}q(q+1)\Delta^3 m_k.$$

则所有这些外插的上包络为：

$$m_{k+i}=\begin{cases} m_k+i\Delta^1 m_k+\dfrac{1}{2}i(i+1)\Delta^2 m_k+\dfrac{1}{6}i(i+1)(i+2)\Delta^3 m_k, & (i=1,\cdots,q)\\ m_{k+q+(i-q)\alpha}, & (i=q+1,\cdots,l)\end{cases}$$

关于它的证明，利用命题 2 即可。

命题 7. 当 $d=4, l>0$ 时，且$(m_1,\cdots,m_k)$为在 $l=0$ 时(6.4.4)式的解满足(6.4.13)式，p 与命题 5 中的相同：

(a) 如果 $p\geq l$，则外插：

$$m_{k+i}=m_k+i\Delta^1 m_k+\frac{1}{2}i(i+1)\Delta^2 m_k,\quad (i=1,\cdots,p)$$

为对所有的四阶$(m_1,\cdots,m_k)$的可容许的外插的一个下包络。

(b) 如果 $p\leq l$，设：

$u=\min(l,1+\text{gilb}(-2\Delta^1 m_k/\Delta^2 m_k))$，则外插

$$m_{k+i}=\begin{cases} m_k+i\Delta^1 m_k+\dfrac{1}{2}i(i+1)\Delta^2 m_k+\dfrac{1}{6}i(i+1)(i+2)\dfrac{-2(\Delta^1 m_k+u\Delta^2 m_k)}{u(u+1)}, & (i=1,\cdots,u)\\ m_{k+u}, & (i=u+1,\cdots,l)\end{cases}$$

为所有四阶$(m_1,\cdots,m_k)$的可容许外插的一个下包络。

关于它的证明，利用命题 3 即可。

命题 7 说明最小下包络是沿着一个二次函数，或开始于一个三次函数，然后逐步递减至一个常数函数的。

6.4.2 结构化模型

这一方法的例子是 Littlewood 的结构化模型。它假设:

- 系统由一有限的离散模块集合组成,
- 每个模块中故障的发生是泊松过程,
- 每个模块 i 有各自的故障率 λ_i,
- 系统任一时刻只运行一个模块,
- 系统执行按半马尔可夫方式在模块之间转换,
- 每对模块 i 和 j 间接口有一给定的故障概率 p_{ij}。

可以证明,如果在长度大于每一模块的"平均逗留时间"(转换到另一模块之前,系统执行花在一个模块内的时间,称"逗留时间")的期间内观察系统的行为,且设模块及接口的故障率都很低,则整个系统的故障过程,就是一个具有由下式表示的故障率的泊松过程:

$$\lambda = \sum_{i} a_i \lambda_i + \sum_{i,j} b_{ij} p_{ij},$$

其中,a_i 表示系统执行花在模块 i 中的时间的比例常数,λ_i 表示模块 i 的故障率,b_{ij} 表示执行从模块 i 向模块 j 转换的频度,$p_{ij} = P_r\{$执行从模块 i 向模块 j 转换时,在接口上发生故障$\}$,a_i 和 b_{ij} 表示模块的执行轮廓,它们能由记录模块执行情况的软件记录下来。λ_i 可用前述任一黑箱模型在模块测试期间估计之。p_{ij} 可从系统组合起来以后的故障记录中予以推断。

组合起来的系统显示出的全部故障都不是由模块本身的错误引起的,它们在系统测试期间决不会出现。要应用这一模块,关键在于组合之前估计出 a_i, b_{ij} 和 p_{ij}。

由系统设计的某些知识来估计 a_i 和 b_{ij} 应是可能的。特别,b_{ij} 可以期望为是一个稀疏矩阵,因为有许多成对的模块将不可能相互调用。在使用分层结构化设计的情况下,尤其如此。

估计 p_{ij} 的困难最大。即使如此,如果 λ_i 的第一项能单独预先加以估计,则在组合之先,对故障数据设置一个下界,应该是可能的。

组合起来的系统在操作一段时间之后,当关于 p_{ij} 的下界能被估计出来时,在缺乏故障数据,从而使关于黑箱模型的标准参数估计技术不能产生出有用结果的场合,本方法可提供对软件可靠性的估计。

本方法还可推广至表示采用容错设计技术的系统的结构。

6.4.3 Cox 比例风险函数模型

在建立可靠性模型时,对下面各种因素建立模型是可能的:程序大小及复杂性,程序员的能力,系统负载(同时发生的过程的个数、在线终端数,等),硬件结构和处理器的能力,用户类型(科学计算或商用,等)。虽然到目前为止,对它们作系统的量化仍不可能,但

它们已知都是影响到软件可靠性的一些因素。

模型的基本等式是:

$$\lambda(t) = \lambda_0(t)\exp(b_1e_1 + b_2e_2 + \cdots + b_ve_v),$$

其中, e_i 是“解释变量”,它们表示任一被怀疑可能与故障率有关的因素是出现、缺省以及出现的程度。(取值 0 或 1 即表示该因素出现或缺省),或更大的取值范围(表示它出现的程度)。b_i 是要从时间和故障的记录加以估计的未知参数,并表示每个变量所具有的影响(取值 0 意味着没有影响)。$\lambda(t)$ 是将被观察到的(如必要,可在整个时间区间上变化)实际故障率,给定 e_i 的值的特殊集合。$\lambda_0(t)$ 是如果所有的 e_i 为 0 时,应该被观察到的“基线”故障率。

Cox 还提供了估计 $\lambda_0(t)$ 和 b_i 的方法。

6.4.4 时间序列

Crow 和 Singpurwalla 主张在修复行为之间的软件故障的出现时,可以处理成聚丛的一个序列。

聚丛是类似的实体构成的一个组。至于软件故障,聚丛能由操作环境中的各种变化所致。例如:对于软件改变而作出的请求的性质,具有相类似的请求会一个接一个彼此很接近地出现的倾向。于是,其结果就是一连串的故障发生。在随机状态计算机系统中,操作环境影响到软件的性能。在这些系统中,计算机在时刻 t 和 $t+s$(s 很小)的状态一般将是相似的,而环境则在这一段时间内也将是相类似的。因此,在一次软件故障出现之后,将会有另外的故障在不久的将来也跟着出现的机会是增加了。其结果就是软件故障在随机状态计算机系统中的出现是一个聚丛的方式。

下面,我们介绍一个用以描述软件故障的傅利叶级数模型。

设相继的故障之间的时间对于某些故障是短暂的,而对于另一些故障是长的,它们出现于一个聚丛中。聚丛可以是系统的、近似于系统的,或不是系统的,这取决于操作环境。

Crow 和 Singpurwalla 将这一聚丛过程看作一时间系列,并引入一个傅利叶级数模型,以描述以聚丛方式出现的软件故障。

关于软件故障间的时间,$\{t_i\}$的分析模型由下式给出:

$$t_i = f(i) + \varepsilon,$$

其中, $i = 1, 2, \cdots, n$, ε 是具有均值为 0 和常数方差的扰动项, $f(i)$ 是一循环趋势。

如果可能,为了把握住 $f(i)$ 中的循环模式,将它表示成 $\sin A$ 和 $\cos B$ 的线性组合,将是方便的。已知这种表示就是 $f(i)$ 的傅利叶级数表示。为了获得一个 $f(i)$ 的傅利叶级数表示,我们可以将 $f(i)$ 写成:

$$f(i) = \alpha + \sum_{j=1}^{q} \alpha(k_j)\cos\frac{2\pi}{n}k_j i + \beta(k_j)\sin\frac{2\pi}{n}k_j i ,$$

其中, n 被假定为奇数,

$$q = (n-1)/2, \quad k_j = j, \quad j = 1, 2, \cdots, q.$$

α, $\alpha(k_j)$, $\beta(k_j)$ 的估计用最小二乘方法得出如下:

$$\alpha = \frac{1}{n}\sum_{i=1}^{n} t_i,$$

$$\alpha(k_j) = \frac{2}{n}\sum_{i=1}^{n} t_i \cos\frac{2\pi}{n}k_j i,$$

$$\beta(k_j) = \frac{2}{n}\sum_{i=1}^{n} t_i \sin\frac{2\pi}{n}k_j i,$$

其中，$j = 1, 2, \cdots, q$。

$Q(k_j) = \alpha^2(k_j) + \beta^2(k_j)$ 相对于 k_j/n, $j = 1, 2, \cdots, q$ 的图形，称为系列 $\{t_i\}$ 的谱图(Spectrogram)，它用以经辨认与频率 k_j/n 相关的 $Q(k_j)$ 的大值，以发现时间序列中的周期。它标示出聚丛的存在，且如果聚丛存在，则它可帮助辨别它是否是系统的。

谱图还可用以获得傅利叶级数模型的缩略形式，用以描述聚丛式故障数据。设相对的 $Q^2(k_j)$ 的大值出现在 $k_j = j\ (j \in I)$，其中 I 是 $\{1, 2, \cdots, q\}$ 的一个子集合。软件故障之间的时间的模型即为：

$$f(i) = \alpha + \sum_{\substack{k_j = j \\ j \in I}} \alpha(k_j)\cos\frac{2\pi}{n}k_j i + \beta(k_j)\sin\frac{2\pi}{n}k_j i\ .$$

关于聚丛过程，谱图提供我们许多有用的信息。设 L 是集合 I 的最大元，于是 n/l 就是最小的周期。如果 n/j 的值是 n/l 的倍数，则序列是具有最小周期 n/l 的周期性的序列。对最小周期 n/l 的识别隐含了一个软件故障的聚丛，并且在每次长为 n/l 的观察之后，聚丛就系统地出现。如果 n/j 的值近似于 n/l 的倍数，则在长度大致为 n/l 的观察之后，聚丛以重复它们自身的方式，将倾向于系统地出现。如果在 n/l 和 n/j 的值之间不存在倍数关系，则序列具有多种周期。此时，聚丛过程就不是系统的，并且描述软件故障的缩略式模型就不能用于预测将来的故障，而仅仅只能用以解释观察到的故障的描述工具。

Crow 和 Singpurwalla 已使用上述方法分析产生于在各种不同条件下运行的两个软件系统的三个故障数据的集合。分析结果显示出周期性(也即是说，聚丛)是存在的，并且聚丛过程是近似系统的。在所有的情况下，缩略模型提供了对于故障数据的一种适宜的描述形式。聚丛的周期性显示出这里介绍的模型可以用于透彻了解将来的软件故障行为，这些结果证明，傅利叶级数模型能成功地用于描述聚丛式的软件故障，并且证明模型的谱图可以提供关于聚丛过程的有用信息。

Horigome, Singpurwalla 和 Soyer 将软件的可靠性增长过程看作一个时间序列，并引入一个随机系数自回归模型以描述这一序列。

设 x_t, $t = 0, 1, 2\cdots$ 表示软件的无故障时间，x_0 表示软件在开始测试时的无故障时间。因为对软件的修改涉及到小的，但重要的设计变动、布局变化，设 x_t 与 x_{t-1} 有关，且希望 $x_t > x_{t-1}$ 是合理的，当然使得 $x_t \leq x_{t-1}$ 的修改行为也是可能的。

$$x_t = x_{t-1}^{\theta}$$

关于 $t = 0, 1, 2, \cdots$ 成立，其中 θ 是一个系数。θ 的值在 $\theta > 1$ 时表示可靠性增长，$\theta < 1$ 时表示可靠性恶化。

为了对作为可靠性增长模型的幂律引入一种不确定性，或者为计算由它产生的某些细小偏差的可能性，我们引入增殖的误差项 δ，于是有：

$$x_t = x_{t-1}^{\theta}\delta,$$

并假定 δ 是关于已知参数 0 和 σ^2，对数正态的。

对上式两边取自然对数，并令 $v_t = \ln\delta\ (t = 1, 2, \cdots)$，则有：

$$Y_t = \theta Y_{t-1} + v_t,$$

Y_t 定义为 $\ln x_t$, 它与 v_t 都服从正态分布, 并且 v_t 有均值为 0, 方差为 σ^2。

显然, 序列$\{Y_t\}(t = 1, 2, \cdots)$是由一阶自回归过程(具有随机系数 θ)描述的。

关于 θ, 讨论如下。将上面的讨论总述如下:

$$Y_t = \theta Y_{t-1} + v_t \quad (Y_t = \ln x_t),$$
$$v_t \sim N(0, \sigma^2), \ \sigma^2 \text{ 已知},$$
$$\theta \sim N(\lambda, \sigma^2), \ \sigma^2 \text{ 已知},$$
$$\lambda \sim N(\mu, \sigma^2), \ \sigma^2 \text{ 已知}。$$

在上面的表达式中, θ 独立于 v_t 加以估计。

假定 θ 的密度, 满足正态分布, 并有均值 λ和方差 σ^2 是合理的。当 σ^2 很小时, λ的值接近于 0, 将倾向于指示可靠性恶化; 否则它们落在 1 的邻近区域或大于 1, 则倾向于指示可靠性增长。另外, 假定 λ满足一具均值 μ和方差为 σ^2 的正态分布也是合理的。如要考察 λ的值对于可靠性变化趋势的影响, 我们可以选取 μ的值为 1, 0, 其实它选什么值是无关紧要的。σ^2 的值的确定, 将反映出我们关于 μ的选取的相信程度。

给定 $y^{(t)} = (y_1, y_2, \cdots, y_t)$, 主要目的在于对 θ, λ和 Y_{t+1}的推断。注意 θ的后验均值, $E(\theta | y^{(t)})$, 从 $t-1$ 步向 t 步, 可靠性增长或衰退可以由它反映出来。而 $E(\lambda | y^{(t)})$ 则反映出可靠性的过度增长或衰退。例如, 对于 t 和大多数值, 如 $E(\lambda | y^{(t)})$ 倾向于大于 1, 则我们可以得出结论: 可靠性有着过度的增长。在实用中, 这些量对于时间的图形, 给我们管理、监视软件的可靠性, 提供了许多有用的信息。

给定 $y^{(t)}$, λ的后验分布为:

$$(\lambda | y^{(t)}) \sim N(M_t, S_t),$$

其中,

$$M_t = \frac{S_{t-1} y_t y_{t-1} + M_{t-1} r_t}{S_{t-1} y_{t-1}^2 + r_t},$$

$$S_t = \frac{S_{t-1} r_t}{s_{t-1}' y_{t-1}^2 + r_t},$$

$$r_t = \sigma^2 y_{t-1}^2 + \sigma^2,$$

$$M_0 = \mu, \quad S_0 = \sigma^2.$$

θ的后验分布在给定 $y^{(t)}$时, 为:

$$(\theta | y^{(t)}) \sim N(\hat{\theta}, \hat{\sum}_t),$$

其中,

$$\hat{\theta} = \frac{\sigma^2 M_t + \sigma^2 y_t y_{t-1}}{r_t},$$

$$\hat{\sum}_t = \frac{\sigma^2(\sigma^2 S_t + \sigma^2 r_t)}{r_t^2}.$$

给定 $y^{(t-1)}$, Y_t 的预测分布也是一个正态分布:

$$(Y_t | y^{(t-1)}) \sim N(M_{t-1} y_{t-1}, \ y_{t-1}^2 S_{t-1} + r_t).$$

Horigome, Singpurwalla 和 Soyer 应用上述可靠性增长(衰退)模型, 对 Musa 的一个实际数据进行了分析, 结果显示出模型对于监控可靠性变化和预测下一次故障时间(的自然对数值), 都取得了很好的效果。

第七章　模型的比较与选择

在第五、六两章里，我们讨论了许多软件可靠性模型，揭示了各种各样的方法。但是，怎样对它们进行比较，哪些模型是最好的，哪些模型对于具有不同要求和目的的软件工程师和软件管理人员是最适用的呢?

某些研究者已试着对各种不同的模型进行比较，但是，由于高质量的软件故障数据的缺乏和关于不同模型比较和评价标准尚未建立起来，所有这些研究的努力都碰到了障碍。因此，首先得从这两方面着手开展工作。于是，有利用价值的软件故障数据的收集工作已为人们重视，并收集了一些不同软件系统的故障数据，象 Musa 所收集的许多高质量的关于执行时间的故障数据。另外，许多从事软件可靠性研究的学者，开展了许多联合研究工作，从各个方面为建立一致的比较和评价软件可靠性模型的标准，进行了卓有成效的合作，如: Iannino, Musa, Okumoto 和 Littlewood 于 1984 年的合作。

本章我们要讨论关于软件可靠性模型比较和评价的标准，并结合对软件进行测试的各种策略，讨论怎样选用合适的模型于实际的应用，最后讨论工程上常使用的一些比较技术。

§ 7.1　比 较 的 标 准

我们在这里要讨论的标准，是由 Iannino, Musa, Okumoto 和 Littlewood 于 1984 年提出的原则性标准。利用这些标准，我们可以评价软件可靠性模型，判定它们的价值，比较它们的优劣，在对它们进行比较时，应考虑到它们与各种软件系统之间的关系。虽然这并不意味着对模型的评价直接依赖于应用，但是只使用少量低质量的软件故障数据或不一致的数据，对模型是不可能作合理的比较的。

经过比较，很可能会淘汰一些与我们将要讨论的标准并不符合的模型。另一方面，在可接受的模型之间也很可能无法作出明确的选择。根据模型被应用的环境条件，可以给不同的标准设置相应的权。当我们对两个模型进行比较时，应该同时考虑所有的条件。不应在考虑其它标准之前就根据某一标准淘汰某些模型，除非它们的预测有效性太差。另外，也不要期望某个模型能满足所有的标准，它在某些方面总有些差强人意，这时就要靠我们对它进行综合平衡的考虑了。

本节要介绍的比较标准包括: 预测的有效性、能力、假设的质量、可应用性以及简捷性五个方面。我们将在下面进行详细的介绍。

7.1.1 预测的有效性

预测的有效性是指模型根据现在和过去的故障行为(即指故障数据),预测将来的故障行为的能力。这一能力只有当故障行为发生变化时才是有意义的。因此,它可以应用于测试阶段,或操作运行的维护阶段,对软件作特别的修改时,也可以应用它。

预测的有效性至少可以通过两种方式表示出来,它们是以两种等价的、描述故障随机过程特性的方式为基础的。它们是:

(1) 故障数方式;

(2) 故障时间方式。

故障数方式比故障时间方式更能产生出一种可实际应用的方法。前一种方式使用 $[M(t), t \geq 0]$ 以描述故障随机过程,它表示到时间 t 为止,发生的故障数的期望值。这种计数过程是由规定 $M(t)$ 的分布来描述的,也包括均值函数 $m(t)$。

设在测试时间 t_q 的末尾我们已观察到 q 个故障,使用到时间 t_e 为止的故障数据($t_e \leq t_q$)以估计 $m(t)$ 的参数。将参数的估计值代入 $m(t)$,以估测到时间 t_q 为止的故障数 $m(t_q)$,并将它与实际观察的值 q 进行比较。对于 t_e 的各种不同的值,重复这一过程。

将 $[m(t_q) - q]/q$ 与 t_e/t_q 作为两个轴,以它们的不同值作出图来,就可以直观地检验该模型的预测有效性。当 t_e 接近于 t_q 时,数值 $[m(t_q) - q]/q$ 将接近于 0。如果点为正,则表示估值偏高;点为负,则估值偏低。数值越接近零,则表示估计越精确,因此,模型也就更好。

上述规范化方法使我们能够覆盖由不同的故障数据集合所获得的错误数曲线。关于有关模型的预测有效性,作为一个总的结论,我们可以比较关于不同的数据集合作出的中位线的图形。如果一个模型产生出的曲线最接近于零,我们就认为该模型最优。注意使用这一方式的一个好处在于:规范化的图形是与具体软件项目无关的。

故障时间方式利用故障时间以描述故障随机过程,故障时间一般有执行时间和日历时间两种。设在测试期间,我们观察到 m 个故障时间,记为 $T_1, T_2, \cdots, T_m$。累积分布函数为随机过程提供了最一般(含最多信息量)的描述,所有其它的量,如:MTTF(如果存在)、故障密度、可靠性等,都可以由它获得。每一个软件可靠性模型都含有从故障时间数据 $t_1, t_2, \cdots, t_m$ 推断累积分布函数参数的过程。与模型以及推断过程相关的对于第 i 个故障间隔时间 T_i 的预测分布,可以写成 $F_i(t_i \mid t_1, t_2, \cdots, t_m)$。

注意:$i > m$,因为我们关心的是预测。我们对模型以及与之相关的经验数据的推断过程所产生的故障间隔的预测分布,以判定模型的预测有效性。这可以通过作出 U-图(由 Littlewood 和 Verrall 于 1973 年提出)来进行。

我们的目的是用 m 个故障间隔时间预测第 i 个故障间隔。设 $k = i - m$,则 k 表示预测长度,或我们往下将要预测 k 个故障间隔时间。因为我们需要许多数据点以检验 F_i 的预测有效性,我们可以固定 m,让 $k(k > 0)$ 变化,或都固定 k,让 m 变化,在这两种作法里,i 都要变。对于前一种情况,我们以固定的样本大小为基础,考察各种长度的预测结果。对于后一种情况,我们以不同的样本大小为基础,考察固定长度的预测结果。预测的有效性应由这两种方法来予以检验。如果在这两种方法的检验过程中,某个模型具有等效的预测

有效性,且达到等效性越早(也即,要求更小的故障样本),则该模型就越好。

数据的稀疏性(对于每个预测分布只有一段数据)是一个问题。这时应该对于在不同的点上的分布之间的函数关系取平均值,以某种方式来评价预测的有效性。

虽然大多数模型的提出要求与特定的参数推断过程相联系,例如:最大似然估计法或最小二乘法,如果在预测结果不好时,我们可以考虑其它的参数推断过程。通过生成模拟的故障间隔时间,并使用它们于预测的方案,可以考察参数推断的质量。这样做,排除了模型的任何影响。在预测分布和实际分布之间的任何不矛盾的差别,只反映出推断过程的影响。另外,我们还可使用观察到的值于参数估计,并比较参数估计的分布,以评价推断过程的质量。

7.1.2 模型的能力

模型的能力涉及到模型能在软件工程师、软件管理人员制定软件项目开发计划、管理软件项目开发过程中,对他们所要求的量,以令人满意的精确度进行估计的能力,也涉及到用户在操作运行软件系统时,对他们所要求的量作出精确估计的能力。我们必须依据这些量的数量及重要性规定模型能力的等级。

这些量,按照一种大致的重要性程度可以定为:

(1) 当前的可靠性,平均无故障时间(MTTF),或故障密度;

(2) 期望达到一规定可靠性目标的日期,MTTF,或故障密度目标的日期;

(3) 与达到规定目标有关的人力和计算机资源以及成本要求。

7.1.3 模型假设的质量

关于模型应用的假设的质量问题,下面的一系列考虑应逐个地应用于模型的每一个假设。

(1) 如果对一个假设进行测试是可能的,则由数据所支持的假设的重要性的优先级别应高于其它的假设。而且所有采用该假设的模型,都应优于其它模型。

(2) 如果对假设的测试不可能进行,就应从逻辑上的一致性观点和软件工程经验的观点出发,以判断假设的真实程度。比如,联系到软件和软件开发的其它信息来看,该假设是否有理?

(3) 应该对一个假设的明确性和显然成立的程度进行判断。

对于判断是否将一个模型应用到特定的软件系统或项目开发的管理中去,上面的特性往往都是必需的。

7.1.4 可应用性

对于模型可应用性的优先等级,我们应该用各种不同大小、不同结构、不同功能的软件产品,对它进行判断。另外,还应该将模型应用于不同的开发环境、不同的操作运行环境,以及不同的软件寿命周期阶段,以判定其可应用性。但是,如果某个模型对于很狭窄的软件产品或软件开发环境的范围,给出的预测结果并不理想,也不应轻易地淘汰它。

在实际应用中至少有四种特殊情况是常常要碰到的,一个模型应该或者有能力直接

处理它们,或者与能处理它们的过程并不发生矛盾。这四种特殊的情况是:

(1) 软件逐渐的进化发展;

(2) 根据故障严重程度将它们划分为不同的类;

(3) 处理不完全数据或带有不确定测度的数据的能力(虽然有损预测的有效性);

(4) 在不同的计算机上运行同一个软件。

最后,模型如果在违反它的假设、含有错误的数据或参数估计不对,甚至条件都不正确时,仍然是强壮的,则这种情况是十分理想的。

7.1.5 简捷性

一个模型在下面三个方面应该是简捷的:

(1) 在收集为模型所要求的数据方面,它应该是简单而且经济的,这一点是最重要的。如果不是如此,我们将不使用该模型。

(2) 模型在概念上应该是简单易懂的。软件工程师们没有足够的数学背景知识去理解模型及其它的假设。只有简单的模型才能使他们能够判定什么时候模型是可应用的,超过什么样的限度模型可能偏离应用的现实性。关于模型的参数,应该有易于理解的解释。这样就使得软件工程师们在数据不足或不是很有用时,对参数进行估计时更加可行。模型中含的参数个数也在很大程度上影响到模型的简捷性。应该在一个共同的基础上对参数的数目进行比较。

(3) 作为实际管理和工程化的工具,模型应易于用程序来实现。程序应运行快速而且代价低廉,除了初始的输入数据以外,不应该要求其它的人工干预。

简捷性应是对所有软件可靠性模型的共同要求,只有简单明了的模型才会有广阔的应用前景。

§ 7.2 测试策略与模型的选择

某些研究者试图寻找一个比其它任何模型都要好的模型,但这样一个模型是否存在仍是有疑问的。一般的作法都是根据不同的情况,选用不同的模型。但究竟怎样来选用不同的模型,却又是一个困难的问题。

这一节我们要将测试中的一些重要策略分为几种情况进行讨论,以便很好地选择适用的软件可靠性模型。对于模型的选用,将各种不同的测试策略作为选用模型的标准之一,在很大程度上会增加所选模型的适用性。

我们首先从一种非常简单的情况入手进行讨论,这就是对整个输入空间采取“过一遍”的、有规律的测试方法。这在许多实际情况中无法做到,但可以采用系统的方法从输入空间中有选择地选取测试数据,对软件进行测试。显然,它将导致一种十分特别的故障模式,即在测试期间,软件系统的故障密度是一个常数,它的随机性仅来自软件中错误的随机分布。为了提高测试效率,人们总试图采用系统测试的方法。其次,一种完全随机的测

试情况,是按某种完全随机的方式在输入空间中选取测试数据,用以对软件进行测试。在实际运用中,还有一种我们称之为混合状态的方式,即对软件进行测试所用的输入数据既不是完全系统地选择的,也不是完全随机选择的。为描述这一过程,我们需要更多关于测试集和输入空间本身、以及它们之间的关系的信息。

在一般的情况下,对于选取的模型,都有些参数要进行估计。关于它们,一般在测试开始之前,就已掌握了某些先验信息,因此贝叶斯方法对于软件可靠性分析,是一种非常有希望的方法。

7.2.1 一种理想化的实际情况

通常总是采用某种确定的方式,对软件系统进行测试,即试图遍历输入域中的全部数据。基于此,我们简单地假定,选取的测试数据是从输入空间中系统地加以选取的。虽然在大多数情况下,要做到这一点是不可能的,但人们总是试图选用那些和能提供更多关于软件可靠性信息的输出所对应的输入数据,对软件进行测试。这是由于使用这样的数据,还可以给我们提供某些关于其它未予选中的那些输入数据的信息。换句话说,选取的这类测试数据集,可以尽可能多地‘覆盖’输入空间。关于这种情况,测试是确定的,但测试的结果,或输出的正确性,则是随机的。显然,此时的故障密度是常数。并且,在通常情况下,采取这种测试策略,测试过程都是均匀进行的。

经过上述方式的测试之后,软件的可靠性将依赖于用于测试的数据个数,也依赖于从测试数据所获取的那些关于没有进行测试的其它输入数据的信息。可以将这类测试数据的每一个的覆盖率,作为一个实际的测度。

假设第 i 个测试数据有覆盖率 C_i,为简单计,不妨设:对于所有的 i, i= 1, 2, …, n, 有 $C_i = c$($c \geq 1$,常数),在这里 n 表示选取的测试数据总数。如果对于第 i 个测试数据,软件不出错,则由它所覆盖的 c 个数据,也不会使软件出错。这样实际将输入空间分为两部分:一部分由选取的及由它们覆盖的数据(总共 nc 个)组成;一部分则由未被覆盖的数据组成,设输入空间的总数据个数为 N,则这一部分含有 N - nc 个数据。

又设使用 n 个数据,共查出 n_e 个错误,且全部被排除,于是软件中的剩余错误个数我们可以设为 n_e'个,而它们全部属于未被覆盖的输入部分。上述可示于图 7.1 中。

图 7.1 输入空间被分成两部分

下面讨论软件可靠性估计问题。关键在于估计 n_e', 关于它的一个简单估计是:

$$n_e' = n_e(N - nc)/(nc).$$

因此, 软件可靠性的估计就是:

$$R = 1 - \frac{n_e'}{N} = 1 - n_e(N - nc)/(Nnc).$$

设对于所使用的每个测试数据 i, 它们都有一个覆盖率 C_i, 如 $C_i \neq c$, 则被覆盖的输入数据个数就是 $\sum_{i=1}^{n} C_i$ 个。上面关于 R 的表达式是简单的, 如果 $\sum_{i=1}^{n} C_i = N$, 则使用 n 个测试数据, 覆盖了整个输入空间, 且在排除了查出的全部 n_e 个错误之后, $n_e' = 0$, 于是, $R = 1$, 如果 $\sum_{i=1}^{n} C_i n \ N$, 则有:

$$R \approx 1 - n_e \Big/ \sum_{i=1}^{n} C_i.$$

如果 $C_i = 1, (i = 1, 2, \cdots, n)$, 则:

$$R \approx 1 - n_e/n.$$

这就是 Nelson 模型, 每个输入数据只覆盖自己。

从以上的讨论不难看出, c 实际上是一个平均覆盖率。事实上, 对 i= 1, 2, …, n, 要取每个覆盖率的难度, 比取平均覆盖率要大得多。

假设所使用的 n 个测试数据, 它们每一个的覆盖区域都不是互相离散的, 也即是说, 它们之间都存在着共同覆盖区域, 则情况就变得十分复杂。下面我们讨论其中最简单的一种情况: 设 D_i 为 i 个测试数据上的覆盖域, 依次序 $D_1, D_2, \cdots, D_n$ 个覆盖域的前一个都与它随后的一个覆盖域存在着一个共同覆盖域 d_i, 且 $d_i = D_i \cap D(i+1) \neq$ $(i = 1, 2, \cdots, n-1)$, 此时使用的 n 个测试数据的总覆盖率就是: $\sum_{i=1}^{n} C_i - \sum_{j=1}^{n} ©d_j ©$, 其中 $©d_j ©$ 表示覆盖域 d_j 的覆盖率。其它的情况更为复杂, 但理论上的处理与此相类似。

7.2.2 完全随机的测试策略

这种情况所使用的测试数据, 是以完全随机的方式从输入空间中选取的。假设错误在查出之后, 立即被完全排除。显然, 此时我们就有一个递减的故障密度。也就是说, 在一给定的固定长时间范围内, 故障数将随机地递减。

可以证明, 如果连续的故障间隔呈指数分布, 则作为时间函数的故障数, 它的递减方式是马尔可夫过程型的。由 Jelinski-Moranda 模型所描述的, 就是马尔可夫过程类型, 它假设故障时间呈指数分布, 且含有与软件中剩余错误数成正比例的一个参数。

在这里, 关键的问题是: 如果按日历时间或执行时间计算的测试是均匀进行的, 则指数分布的假设并不重要, 重要的在于测试数据是完全随机地从输入空间选取这一事实。虽然它是许多模型的一个基本假设, 但不少模型的实际应用者却并不真正地采用, 即许多测试数据并非是以一种“完全随机”方式选取的, 它们或多或少带有测试者的主观倾向。这样势必影响软件可靠性分析的质量。

7.2.3 混合测试策略

在实际的测试过程中,有着许多极为复杂的情况。对软件的测试既不是完全系统进行的,也不是完全随机进行的,而是它们混合在一起,一段时间内作系统方式的测试,一段时间内作随机方式的测试,我们定义它为混合测试。

对于混合测试,只要记录下每个数据是怎么选取的,我们就可以很容易作相应的处理。使用故障率加法模型,则整个故障率可以估计如下:

$$\lambda = p \times \lambda_s + q \times \lambda_r$$

其中,λ为软件系统的总故障率;

λ_s为系统选取方式估计出的故障率;

λ_r为随机选取方式估计出的故障率;

$p = n_s/(n_s + n_r)$是系统选择测试数据的比例,n_s是系统选取方式选出的测试数据个数,n_r是随机方式选出的测试数据个数;

$q = 1 - p$。

注意,在系统方式测试期间,λ_s是常数,它的值可以由平均故障时间的倒数很容易估计出来,λ_r可以用最大似然法估计。

从理论上讲,这样的处理是没有问题的,困难在于收集高质量的故障数据。没有数据就没有分析的基础。软件工程处于发展阶段,要解决的问题很多,但为了在实际应用上提高软件可靠性分析的准确度,软件工程师们应更加关注软件工程数据的收集工作。

7.2.4 非均匀测试

测试的全过程并不是均匀进行的,在某段时间内,对软件系统的测试更加集中,有时要集中一段进行强化测试,但也有均匀进行测试的时间段。

对于均匀进行的测试,Musa 在他的执行时间模型中已有所描述,而且这时应用这一模型,通常都能获得较好的结果。在他的模型中强调了 CPU 时间比日历时间更能反映问题的本质,是比日历时间更好的测度。虽然按日历时间计算的测试大多都不是均匀进行的,但按 CPU 时间进行计算,则可以认为是均匀的。但这样一来,收集 CPU 时间数据,当然比只收集日历时间数据要麻烦一些。

另外一个实际问题是:进行测试的客观实体——软件系统本身,也不是自身时间独立的,通常我们在一个软件系统并未完全编码完毕前就开始对组成它的模块或子系统进行测试,然后在整个系统完成之后,又继续测试。实际上,对大多数大型软件系统,我们都是这样做的。另外,在测试期间,对规格说明书进行修改变动也是经常的。

对于非均匀测试及非自身时间独立问题,都可以用非齐次泊松过程(NHPP)类模型来进行描述。这一类模型有 Goel-Okumoto 模型(1979),以及 Yamada-Osaki 模型(1983),它们都已获广泛应用,并在许多情况下都得到许多好的结果。虽然究其根源,NHPP 类模型都是作为硬件可靠性模型的一类,但软件的测试环境条件的非均匀性,为它们在软件可靠性分析中的应用提供了方便的条件。

上面我们的讨论限于软件的测试阶段,至于复杂度测度,Seeding 模型以及其它一些

技术,都未涉及。参数估计是另一个问题,贝叶斯方法在这方面是一个很有希望的技术。

§ 7.3 比 较 技 术

在选择软件可靠性模型时,首先考虑的就是模型的预测精度。有许多统计工具,对于客观地评价预测精度,是很有效的。

在这一节中,我们介绍一些行之有效的模型的比较技术。为简便计,我们仅限于故障间隔时间方式,且对于观察到的 m 个故障间隔时间,令 i= m+ 1,即 k= 1。其它的情况可以此类推。

在一软件系统的测试或操作运行期间,我们必将面对一个不断增加的数据集合。对集合所能允许的尽可能大的 i,取前面 i 个点,以预测(i+ 1)点的情况。为了比较模型的优劣,就有一个就平均水平而言,如何判定哪个模型给出的预测结果最好的问题。也就是说,就平均水平而言,哪个模型所产生出的被观察的点最接近中位数(或均值,如果它们存在的话)。

7.3.1 预分析技术

在拟合选定的模型之前,可以应用某些简单的预分析技术,进行先行处理。

(1) 故障数据的图形表示

用于初始评价的有用的图形表示,可以直观地帮助我们进行分析。它们有:

- 累积错误个数的图形: c(x) 。
- 用于故障计数数据的图形: 在相继时间段中的故障率,由 k_i/u_i 来估计,其中 k_i 是在第 i 个时间段内的错误出现数, u_i 是第 i 个时间段的长度。
- 用于故障间隔时间数据的图形: 在等长的相继时间段内的故障率,由[c(x_i) - c(x_{i-1})] / u来估计,其中 x_i 是直到第 i 个时间区间开始处的累积时间, u 是不变的区间长度。

上面所有的量都相对于累积时间 x 标出。

(2) 试探性数据分析技术(EDA)

对于故障数据图形的检查和试探性数据分析数字化技巧的应用,能揭示出数据中的矛盾。其中特别有用的有:

- 循环相关;
- 故障在短期内的大量出现;
- 标出故障率升高的时间段。

它们可以用于揭示出操作轮廓一致性的缺乏、故障相互独立性的缺乏、因不良修复引入而并非是软件中原有的错误、时间域的不正确定义、或数据收集中的问题等。

试探性数据分析涉及许多技巧,并可进行很多其它的检验,例如对变化趋势的检验。

(3) 等张回归

“等张”的意思就是“考虑到被估计的量的次序关系”。如果数据集合终止于一个无故障运行的时间段内,则在加上“1/2 个错误”到最后的数据点上之后,在表示 $c(x)$ 与 x 的关系的图形上可作出直线段的包络。这一包络在任一时刻的斜率就是瞬时故障率的一个估计,但因它带乐观倾向、多“噪声”、或不稳定,所以最后的故障率不宜用它来估计。

(4) 幂曲线拟合

将相继时间段内估计的故障率 k_i/u_i 和累积错误数 $c(x)$,相对于 x,在重对数纸上作图时,它们倾向于成一直线。这给出下面的关系:

$$m(x) = ax^b.$$

其中,$m(x)$ 为在时刻 x 处累积错误数的期望,$\ln a$ 是截距,$\ln b$ 是在 $\ln a$ 上的 $\ln[c(x)]$ 的回归线的斜率。在 $b<1$ 时,可靠性增长;$b=1$ 时,可靠性为常数;$b>1$ 时,可靠性下降。a 和 b 可以在图上很容易估计,或应用标准线性回归公式以估计它们的值。

对于早期数据,回归法是十分敏感的,除非将某加权因子应用于较后的数据点上,才能克服这一点。例如在开始阶段故障率增加的一个短时间段内,就可能产生 $b>1$ 的估计结果。

7.3.2 U-图

假设某用户在他的软件排错期间,记录下故障间隔时间数据:$t_1, t_2, \cdots, t_{i-1}$,并且关于下一次故障时间 T_i 进行了可信的可靠性估测,令:

$$F_i(t) = P_r[T_i < t]$$

为 T_i 的真实但未知的累积分布函数。以数据和特定的估测系统为基础,可以获得一个累积分布函数的估计,用 $F_i(t)$ 表示。

根据第 i 个故障间隔时间 t_i,利用这一预测系统,能产生关于 T_{i+1} 的估计 $F_{i+1}(t)$。对于某个 $n>0$,使得在 $i+1, i+2, \cdots, i+n$ 步上每次产生一个估计:$F_{i+1}(t), F_{i+2}(t), \cdots, F_{i+n}(t)$。为了彻底了解该估测系统的性能,可以将它们与实际的结果进行比较。如果在估计的结果和真实结果之间存在严重不一致的地方,就要设法消除用户对预测系统是否有用的疑虑心理。

我们在这里要介绍的 U-图和下面将要介绍的 Y-图是用来量测预测性能的两个过程。它们都建立在统计量

$$u_i = F_i(t)$$

的基础之上,根据 $F_i(t)$ 的定义,u_i 是随机变量 T_i 小于实际的观察值 t_i 的概率。

容易证明,如果 $F_i(t)$ 确是 T_i 的真实的累积分布函数,则 u_i 是一个满足均匀分布 $U(0,1)$ 的随机变量的一个实现。

于是,产生出的 $n+1$ 个 u 值$(u_i, u_{i+1}, \cdots, u_{i+n},\ n>0)$ 就是独立恒同分布 $U(0,1)$ 的 $n+1$ 个随机变量的实现。因此,如果能找到$\{u_i\}$是靠近独立恒同分布 $U(0,1)$ 的,则$\{F_i(t)\}$

也就是靠近$\{F_i(t)\}$的。

U-图过程就是表示$\{u_i\}$的样本累积分布函数接近 U(0, 1)的累积分布函数的接近程度的。U(0, 1)的累积分布函数即为过原点的斜率为 1 的直线, n+ 1 个 u 值的样本累积分布函数是定义在区间(0, 1)上的阶梯函数, 从 0 开始增长, 其增量为每步增加 1/(n+ 1)的 n+ 1 个次序统计量。其具体作图方法即为: 将 n+ 1 个 u_i 的值按升序分类, 用$\{u_j\}$表示分类以后的序列($j = 0, 1, 2, \cdots, n$), 把点$(u_j, j/(n+1))$描在图上, 并作一条 45 线。

如 u_i 是均匀分布的, 则这些点将落在(除开一个小的随机噪声外)通过原点的单位斜率线上。预测中的任何偏差将从这条线上的系统变差上反映出来, 且任一点到线上的垂直距离的最大值(柯尔莫哥洛夫——斯米尔诺夫距离)就是预测的分布与真实分布相距多远的一个测度。

下面的图 7.2 示出了一个 U-图的例子。

图 7.2　U-图的例子

7.3.3　Y-图

Y-图要求对$\{u_i\}$作如下的进一步变换:

$$x_j = \ln(1 - u_j), \quad j = i, i+1, \cdots, i+n,$$

$$y_j = \sum_{l=i}^{j} x_l / \sum_{l=i}^{i+n} t_l, \quad j = i, i+1, \cdots, i+n-1.$$

当 u 值都满足独立恒同均匀分布 U(0, 1)时, y 值是 n 个满足独立恒同均匀分布 U(0, 1)的次序统计量。Y-图表示$\{y_i\}$的样本累积分布函数与 45 线的接近程度。

作图方法与 U-图作图法一样, 将点$(y_j, j/(n+1))$描在图上并作出 45 线。这时单位斜率线上产生的偏差表示预测中的趋势。

另外, 对于 x_j 的供选择的检验手段就是: 应用一般关于趋势的拉普拉斯测试; 或构造出 u_i 对于 i 的散布图, 并观察在图形所占区域中点的任一不规则分组。

7.3.4　PL 检验

PL(Prequential Likelihood)检验与 T_i 的估计的概率密度函数有关。

设 T_i 的真实概率密度函数为:

$$f_i(t) = F_i'(t),$$

而它的估计为:

$$f_i(t) = F_i'(t).$$

PL 就定义为 $f_i(t_i)$, 即在实际观察的值 t_i 上的估计的预测密度。

在进行 $n+1$ 次预测之后, 进一步定义 PL 为:

$$PL_{(n+1)} = \prod_{j=i}^{i+n} f_j(t_j),$$

即 $n+1$ 个 PL 的简单乘积。

我们最感兴趣的是: 在我们使用两个不同的预测系统 A 和 B 时的两个 PL 的比值: PLR, 定义它为:

$$PLR = \frac{PL^A_{(n+1)}}{PL^B_{(n+1)}} = \prod_{j=i}^{i+n} f^A_j(t_j)/f^B_j(t_j) \quad ,$$

其中, f^A 表示由预测系统 A 产生的预测密度, f^B 表示由预测系统 B 产生的预测密度。

PLR 可以用于对两个预测系统进行比较: 如果在同一数据集合上应用预测系统 A、B 分别进行 $n+1$ 次预测, 并计算相应的 $PL^A_{(n+1)}$ 和 $PL^B_{(n+1)}$, 如 $PLR \to \infty, (n \to \infty)$, 则预测系统 A 优于预测系统 B; 如 $PLR \to c(n \to \infty, c > 0)$, 即如果 PLR 随着预测次数无限增加而趋于一个有限的极限, 则预测系统 A 和 B 是等价的, 它们各自产生的预测结果也将是等价的。

就是在样本大小有限时, PLR 也能用于比较。在预测系统 B 产生更大偏差的情况, 或 B 系统所使用的模型在把握基本的倾向方面失去作用时, PLR 将大于 1, 从而反映出 B 系统中的不足。无论如何, 这一信息可以很容易地由 U-图和 Y-图得出, 但 PLR 能查出预测系统中不足的一个方面, 而 U-图和 Y-图却不能够。

7.3.5 模型的适应

应该满足什么样的条件, 工作才是适应的呢?如果某个预测系统只是在偏差方面存在问题, 而偏差又是稳定的, 并且在将来的预测方面也仍保持一致性, 则这对于模型的适应而言, 将是一种理想的情况。将来的偏差与过去的偏差保持一致性, 是很重要的, 否则, 过去就不能用作推断将来的基础。

形式化地说, 就是我们要求对于每个 i, 下面的表达式:

$$F_i(t) = G_i(F_i(t))$$

关于某个函数:

$$G_i: (0, 1) \to (0, 1)$$

成立。进而言之, G 函数的集合应保持不变。

条件 $F_i(t) = G_i(F_i(t))$ 简单地描述 F_i 和 F_i 之间的函数关系是不成问题的, 但是要保持这些函数不变, 是一个很强的要求。但是, 如果它是对一个实际发生的情况的好的近似, 则为获得更靠近真实情况的适应性工作就是可能的。当我们在一个数据集合上分析某预测系统的预测质量时, 如果有一个好的 Y-图, 则我们就可以相信已经把握住了基本的趋

势。如果伴有下面的情况发生,则在偏差方面的稳定不变就可以认为是相当适宜的:U-图显示出显著的偏差,并且以后连续作出的 U-图都相当地稳定。

当我们把上述的情况作为一个真实情况的可接受的逼近时,剩下的问题就是如何确定 G 函数。由于 G_i 更正 F_i 中的错误,并且因为这些错误是稳定产生的,所以我们可以将由当前的 U-图中提取的信息作为基础,对 G_i 函数进行估计。设 G_i^* 是 G_i 的估计,并且就象它是真实的 G_i 一样,用来修改 F_i。

考虑某预测系统,它在第 i 步以前产生了 m 个预测结果。如果我们想要从现在开始(第 i 步)进行对该预测系统的适应性检验,就应该采取下述步骤:

(1) 在$(u_{i-m}, u_{i-m+1}, \cdots, u_{i-1}; m>0)$的基础上,考察 Y-图,以保证数据中的基本趋势已经合理地被把握住了;

(2) 在 u 值的同一集合的 U-图基础上,获得 G_i^*;

(3) 产生出原始预测累积分布函数 $F_i(t)$;

(4) 定义出适应的预测累积分布函数:

$$F_i^*(t) = G_i^*(F_i(t))$$

以及适应的预测概率密度函数:

$$f_i^*(t) = F_i^{*\prime}(t) = g_i^*(F_i(t))f_i(t),$$

其中,

$$g_i^*(x) = G_i^{*\prime}(x).$$

从理论上讲,如果 Y-图确实是非均匀的,就不应该作完第 1 步以后再往下作,因为这时发生预测结果不好的原因已不仅是由于偏差的存在。从此以后,我们就应将由最初的预测系统产生的预测结果作为原始的预测结果。

在第(4)步,得出的 $F_i^*(t)$ 是一个真实的预测。适应性预测累积分布函数已不依赖于 T_i 的实现 $t_i, t_{i+1}, \cdots$。

适应性过程似乎将噪声引入了预测,导致比用 PLR 判别出更坏的模型行为。通过使用受限的最小平方参数样条来修匀适应性预测,这一问题就可获解决。

第八章　软件规划管理与可靠性

从软件产品开发者(软件开发小组、公司或软件生产厂家等)的立场出发,主要关心的问题是减少开发成本,保证软件质量,按时交付使用或及时投放市场。从用户的立场出发,主要关心的问题是软件价格要合理,保证按他们的要求设计、开发、交付软件产品。双方的要求都涉及到经济效益问题,而软件规划管理工作对于双方的要求,都能起到保证的作用。在软件规划管理工作中,软件可靠性所起的作用是不容忽视的。在这一章里,我们要围绕经济效益问题对软件的规划管理工作中软件可靠性所起的作用展开讨论,并进而介绍软件可靠性模型在确定终止测试时间和最佳投放时间方面的应用。

§ 8.1　软件规划的经济效益

有一个好计划,对于任何工作都是一个良好的开端。然而,再好的计划也要准备修改变动。有人说:“一个计划只是为了作为修改变动的公共基础。每个人都应了解,计划应该做得改起来方便。”这是很有道理的。

从开发者的立场出发,在一开始就能让用户以某种积极的方式进行参与,是很有好处的。人们对有关的事情都有一种积极的‘参与意识’。在确定软件可靠性指标时,用户的参与,会使他们对以后开发出来的软件质量,感觉放心和信任,并最终赞同在软件可靠性与其它质量指标之间作出的折衷方案。

作好计划的第一步,也可能是很重要的一步,就是对将要开发的项目进行彻底的分析。为此,就应让管理人员和负责开发的软件工程师们就他们的要求对项目充分发表意见,另外还应向那些对可能的应用领域有丰富经验和知识的专家咨询。从这两方面来的意见,要作实际可行性的仔细分析。每一条意见或许都要对应于一种或几种可以采取的行动,而行动就意味着要花费代价,即,对每种意见的满足都必将对应着一定的成本,而预期的将来的收益必须能给以抵偿。

对于本书讨论的软件可靠性技术的各种应用,有五类人员会在开发的不同阶段涉及到,我们将其中主要方面列于表 8.1 中。

对于管理者,软件可靠性技术可以帮助他们在下列各方面发挥作用:

- 管理包括软件在内的项目的状态;
- 预测交付日期;

表 8.1　与软件可靠性有关的人员与任务

任　　务	各　类　工　作　人　员				
	管理者	系统工程师	质量保证工程师	测试者	排错者
系统分析与裁剪过程					
数据收集:					
故障数据收集					
使用资源统计					
参数估计					
运行程序					
系统研究					
监督管理					

· 判定可以对运行系统中的软件进行修改的日期;
· 确定应用于开发的软件工程技术;
· 确定是否接收下一层开发者交付的软件产品。

管理者应具备足够的经验和知识,在系统工程师协助下决定修改计划和预算、系统及部件的可靠性指标、资源分配等。最后还应监督其它各类人员汇总信息。

系统工程师应彻底了解软件可靠性技术,他们选用、调整模型,定义系统故障,并不断解释它们。他们要不断在可靠性指标、资源、进度之间作出折衷,研究局部可靠性与系统可靠性之间的关系,估计故障成本,并直至对提供的服务进行估价。总之,他们所起的作用是为管理提供研究和解释。

质量保证工程师负责数据收集、处理,接收系统工程师的指导,确定估测模型的初始参数值,对测试者和排错者给以指导。他们要考核软件的当前故障密度。

测试者负责查错并记录一切有关信息,排错者负责排错并记录一切有关排错的信息。由他们收集的数据,直接关系到估测精度,以至最后的决策,在计划数据收集时就应充分考虑他们的意见,并使他们随时保持与以上三类人员之间的通讯。

下一步工作是将全部工作任务列出一个表,它应包括所有要求实现的应用和项目的全部特性。再下一步是分配任务,在这方面,表 8.1 是有指导作用的。

下面要考虑完成可靠性各有关统计量的估计所要使用的工具,并预测以后可能会发生的故障行为。

对各类人员的培训工作也应列入计划。

关于软件可靠性估测技术领域的发展,要求更多的实际应用,需要更多的数据、更多的开发项目,以测试和完善自身。更多的实际应用也会为软件可靠性理论研究提供更多的实际背景并提出理论研究的新课题。

软件可靠性理论的实际应用对于经济效益的贡献主要有以下 5 个方面:

(1) 估计正在运行的软件发生故障的代价

发生故障以后,要恢复系统的代价主要有:重新安装、清理数据库、恢复现场。即使恢复过程自动进行,但仍要对系统提供的服务进行中断处理,于是影响到收益。甚至由于发生故障而失去已有的用户,就更加大了损失。下面举例说明对损失的估计方法。如:某系统故障密度是 0.025 次/1CPU 小时,系统每周工作 40CPU 小时,发生一次故障后的恢复工作需 4 人-小时/每次故障,每个工时费为 $50,每月以 4 周计,则每月发生一次故障的代价为 $800,一年以 12 个月计,则损失将为 $9600/1 次故障/1 年。

(2) 为提供的服务计价

先估算出故障成本,加上其它成本,根据经济上的考虑和市场条件,乘上适当的利润率,即可估算出提供服务的价格。

(3) 对于维护阶段的管理,提供合理的目标

对于所谓"版本更新",我们将获得一个分段为常数的故障密度系列。在维护阶段有时改错频繁,故障密度的变化表现出和系统测试时十分相近,呈现出的总趋势是围绕某个值附近不断振荡,很长时间不能稳定。究其原因,是对软件加入新特性,使故障密度增大,随后的排错活动又使故障密度下降。于是使管理者处于互相矛盾的焦点:一方面是要求加入新特性,一方面是要求系统有尽可能高的可靠性。解决办法是确定合理的故障密度值。若估计的系统故障密度大于它,就应多考虑可靠性问题;若低于它,可适当考虑加入新特性。

(4) 帮助打开"市场窗口"

"市场窗口"是指这样的一段时间,其间软件开发者能推出新产品并获取利润。关键在于开辟"市场窗口"的最后期限。过早,则产品可靠性低,影响声誉,会失去许多客户;过迟,则竞争对手会占领市场,大量低价销售同类产品,将他们置于极端困难的境地。在某种意义上讲,决定开辟"市场窗口"的最后期限,是十分关键的。

(5) 有助于开发者在可靠性和资源、成本上作出折衷

故障密度与可靠性是一个问题的两种不同提法。故障密度的降低,即意味着可靠性的增加;反之,可靠性增加,则故障密度下降,或初始故障密度与故障密度比值增加。该比值增加,就意味着要求使用的有限资源增加,因此要增加开发成本。如可用的有限资源不变,则该比值增加,势必要求延长开发时间。期间的折衷,直接关系到软件可靠性估测技术的应用。

§ 8.2 软件测试终止时间与最佳投放时间

上一节中讲到的软件可靠性技术对经济效益贡献的(4)、(5)两个方面,就是这一节要讨论的主题。

软件可靠性增长模型不仅可应用于对软件的可靠性估测,还可以应用于软件系统寿命周期的成本分析、软件测试终止时间与最佳投放时间的确定。可以这样说,开发软件可靠性模型的一个重要目的,就在于为制订各种决策提供一个统一的分析框架。在实际应用中非常关心的一个重要问题就是何时能够终止软件的测试并将它交付用户使用或

投放市场。

软件的质量很大程度上依赖于花在对它测试的时间的长短。测试阶段越长，软件的质量越好，可靠性越高。一般说来，软件测试对软件开发的总成本有着很大的影响，因为改正查出的错误要求大量人力、时间、设备和工具。如果在交付使用以后的软件操作期间，还有大量错误存在于软件之中，则维护费用也要随之增加。另一方面，测试的延迟会引起用户的不满，不仅限于增加开发成本，也增加了用户因不能及时用上软件而产生的经济损失。于是就增加了要求及早终止测试和投放市场的压力。这些要求往往是互相矛盾的，特别在投放时间成为关键问题时，例如，由于军事上的要求或开辟“市场窗口”的需要，进度就是十分关键的因素。

从实质上讲，测试终止时间和投放时间是一个问题，测试一旦终止，也就意味着产品可以投放市场了。对于它们的确定，可以有成本上的考虑，也可以有可靠性指标上的考虑，还可以将成本与可靠性两种因素综合起来一并予以考虑。下面我们分别就这三种方式进行深入的讨论。图 8.1 示出在作出决策时可以遵循的一种途径。

图 8.1　决策途径

8.2.1　可靠性指标

给定可靠性指标 R_0，现在要考虑的是为达到 R_0 所要求的测试时间。

选定 NHPP 的 G-O 模型，下面来讨论之，设已知的用在测试阶段的时间为 t，在再运行时间 x 时刻后，软件的可靠性由下式给出：

$$R(x\mid t) = \exp[-N_0(e^{-t} - e^{-(x+t)})],$$

或者：

$$R = \exp[-m(x)e^{-t}], \tag{8.2.1}$$

其中：

$$m(x) = N_0(1 - e^{-x}).$$

对于这一问题的一种通常的解决方法是：在时刻 X 预测的软件可靠性达到值 R_0 时，就停止测试。于是问题就变成解(8.2.1)式以求 t：

$$R_0 = \exp[-\ m(x)e^{-\ t}]$$
$$= \exp[-\ N_0(1-\ e^{-\ x})\ ¡¤e^{-\ t}].$$

对上式两边取对数,并解之,得:

$$t = (1/\ )\ \ln[N_0(1-\ e^{-\ x})] -\ \ln\ln(1/R_0)\ .$$

在讨论 G-O 模型时,我们已研究了参数 N_0 和 $\ $ 的估计方法。根据它,我们可以利用测试时收集的数据予以估计。于是,就可以在给定 x 的值时,决定为达到可靠性指标 R_0 的测试时间 t 的值。

例　设通过参数估计,有:

$$\hat{N}_0 = 1348,\quad \hat{\ } = 0.124,$$

并给定:

$$R_0 = 0.7,\quad x = 0.1(\text{周}).$$

则:

$$t = (1/0.124)\{\ln[1348(1-\ e^{(-\ 0.124\times\ 0.1)})] -\ \ln\ln(1/0.7)\}$$
$$= 30.976 \approx 31(\text{周}).$$

即,为达到 $R_0 = 0.7$ 的指标,需要测试 31 周的时间。

为了考察 t 对 $R(x©t)$ 的影响,图 8.2 示出可靠性 $R(x©t)$ 与 $m(x) = 5(5)50$ 的关系。在图中我们可以看出,在测试时间 t 增加时,如保持 x 固定,于是对于固定的 $m(x)$, $R(x©t)$ 很快就增加并趋向于 $R(x©t) = 0.95$,然后 $R(x©t)$ 的增加就变得十分缓慢。它也即表示出再要增加软件的可靠性,就要花费相当长的测试时间才能使软件可靠性获得微小的改善。从经济的角度来看,一味增加测试时间,收效并不明显。

图 8.2　$R(x©t)$ 与 $m(x) = 5(5)50$ 的关系

如果我们选取将软件中的错误分为两类的 Yamada-Osaki 的 NHPP 模型,则讨论方式与上述相类似:

$$R(x©t) = \exp[-\ N_0\sum_{i=1}^{2} P_i(e^{-\ i^t} -\ e^{-\ i^{(t+\ x)}})].$$

对于给定的 R_0 和 x,由于上式是一个单调递增的函数,一定存在一个正的 T,t= T,使得满足:

$$R_0 = \exp\left[-\sum_{i=1}^{2} e^{-\ i^T} m_i(x)\right],$$

其中,

$$R(x \mid 0) = \exp[-m(x)],$$

$$R(x \mid \infty) = 1.$$

J. D. Musa 则从另外一个角度来考虑这一问题。

如果软件终止测试的日期规定得很严格,不容变动,则软件的故障密度目标就是可以考虑的了,即或者根据可应用的资源及经济条件以确定新的故障密度目标,或者在故障密度与这些条件之间作出折衷方案。实际上就是从执行时间的角度来考虑变动方式。

假定在软件测试的开始,根据软件本身的特性及程序设计环境,确定了故障密度指标。在测试过程中,我们可以得到 λ / λ_0 和执行时间 τ 之间的关系:

$$\tau = \frac{V_0}{\lambda_0} \ln \frac{\lambda_0}{\lambda} \quad (\text{基本执行时间模型}),$$

或:

$$\tau = \frac{1}{\theta \lambda_0} \left(\frac{\lambda_0}{\lambda} - 1\right) \quad (\text{对数泊松执行时间模型}).$$

例　使用表 8.3 中的关于 System T1 的故障时间数据,对上面两式中的参数进行估计,(此时使用测试压缩因子的值为 C= 1),可以得到:

基本执行时间模型:

$$\lambda_0 = 17.8 \text{ 次故障/CPU 小时},$$

$$V_0 = 142 \text{ 次故障}$$

对数泊松执行时间模型:

$$\lambda_0 = 39.9 \text{ 次故障/CPU 小时},$$

$$\theta = 0.0236/\text{次故障}$$

使用上面的两个式子,可以分别对于各种不同的故障密度改善因子确定出测试时间的长度(见表 8.2)。

表 8.2　故障密度改善因子与执行时间

故障密度改善因子	执行时间(CPU 小时)	
	基本执行时间模型	对数泊松执行时间模型
10	18.4	9.56
100	36.7	105.1
1,000	55.1	1060.9
10,000	73.5	19619.0

日历时间与执行时间的关系,将随模型的日历时间部分的参数变化而变化。该关系能用于根据给定的投放时间和可应用的资源以决定比值 λ / λ_0,并进而决定故障密度目标。

表 8.3　关于 System T1 的故障时间数据(单位:CPU 秒)

3	2676	7843	16185	35338	53443
33	3098	7922	16229	36799	54433
146	3278	8738	16358	37642	55381
227	3288	10089	17168	37654	56463
342	4434	10237	17458	37915	56485
351	5034	10258	17758	39715	56560
353	5049	10491	18287	40580	57042
444	5085	10625	18568	42015	62551
556	5089	10982	18728	42045	62651
571	5089	11175	19556	42188	62661
709	5097	11411	20567	42296	63732
759	5324	11442	21012	42296	64103
836	5389	11811	21308	45406	64893
860	5565	12559	23063	46653	71043
968	5623	12559	24127	47596	74364
1056	6080	12791	25910	48296	75409
1726	6380	13121	26770	49171	76057
1846	6477	13486	27753	49416	81542
1872	6740	14708	28460	50145	82702
1986	7192	15251	28493	52042	84566
2311	7447	15261	29361	52489	88682
2366	7644	15277	30085	52875	
2608	7837	15806	32408	53321	

注:测试终止出现在 91,208CPU 秒处。

8.2.2　成本指标

成本指标直接关系到软件供求双方的经济利益,它的重要地位是显而易见的。如果软件在投放以前的测试成本已超过了由于故障而产生损失的代价以及在投放之后对剩余错误的修改所必需的成本之和,那它就应该投放出去了。在软件的测试期间,迟早会出现这样一个时刻:再进一步测试下去的成本比由于投放软件所带来的损失和允许用户修改剩余错误的成本之和还要大。如果对软件继续测试下去,在可靠性方面的收效不大,而成本又如此之高,显然是不合算的。

下面我们先介绍一个成本模型。

设 t 表示时间变量,t^* 为最佳投放时间,D 表示投放软件的最后期限,C_1 表示投放之后每个错误的平均修复成本,C_2 表示投放之后每个错误造成的平均损失,$C_3 = C_1 + C_2$,C_4 表示执行每单位时间测试所花 CPU 时间的成本,C_5 表示软件已投放后使用人工所获收益,C_6 表示每单位时间在应用中运行软件的成本,C_7 表示测试期间每单位时间人工的成本,θ表示测试期间一个错误的期望发现时间,N_r 表示剩余错误数,B 表示在应用中运行软件时每单位成功运行时间的收益,C(t) 表示成本-收益因子,P(t) 表示由于延误投放而产生的处罚函数。

根据 Moranda 的几何型营养良化模型的假设,有:测试期间查出错误的时间呈指数分布,而且所有的错误查出时间都是独立的。因此有:

$$E\{\text{在区间}(0,t)\text{内的错误发现}\} = N_r(1 - \exp(-t/\theta))$$

如果决定在 t 时刻投放软件,则:

$$C_3(N_r - N_r(1 - \exp(-t/\theta))) = C_3N_r\exp(-t/\theta)$$

就表示投放后改正剩余错误的期望成本。

$$(C_4 + C_7)t$$

表示在区间(0,t)内的测试成本,并且如果 $t \leq D$,则:

$$(C_6 - C_5 - B)(D - t)$$

就表示在软件投放(D- t)时间后的净收益。

于是,当在 t 时刻投放软件时期望的成本-收益函数就是:

$$C(t) = \begin{cases} C_3N_r\exp(-t/\theta) + (C_6 - C_5 - B)(D - t) + (C_4 + C_7)t, & \text{如果 } 0 \leq t \leq D; \\ C_3N_r\exp(-t/\theta) + (C_4 + C_7)t + P(t), & \text{如果 } t > D. \end{cases} \tag{8.2.2}$$

最佳投放时间 t^* 满足: $\min\{C(t)\}$.

例 将(8.2.2)式的成本模型应用于下列数据:

	情况 1	情况 2
C_3($/月)	3500	6000
C_4($/月)	1000	1000
C_5($/月)	5000	5000
C_6($/月)	2500	2500
C_7($/月)	8000	8000
B($/月)	10000	10000
P(0)($)	0	10000
D(月)	3	2
N_r	30	50
θ(月)	1.2	1.6
t^*(月)	1.68	4.86

对于情况 1, $t_1^* = 1.68$(月)。在 t_1^* 时投放软件,可以节约 $11,000,软件中的剩余错误个数可望为:

$$N_r\exp(-t_1^*/\theta) = 7.4 \approx 8(\text{个}).$$

对于情况 2, $t_2^* = 4.86$(月)。为了减少整个的成本,在投放的最后期限之后 2.86 个月投放而不顾罚款 $10000,还可以比在 2 个月时投放要节省 $35000。

Yamada-Osaki 将 G-O 成本模型推广至他们的错误分类模型上去。他们定义:

C_{1i}:表示在测试期间改正 i 类错误的成本,

C_{2i}:表示在运行期间改正 i 类错误的成本,则成本模型变为:

$$C(T) = \sum_{i=1}^{2} C_{1i}m_i(T) + \sum_{i=1}^{2} C_{2i}\{m_i(t) - m_i(T)\} + C_3T,$$

$$\lambda_i(t) \equiv dm_i(t)/dt \quad (i = 1, 2).$$

假定:

$$C_{2i} > C_{1i} > 0 \quad (i = 1, 2), \quad C_3 > 0.$$

(1) 如果 $\sum_{i=1}^{2}(C_{2i} - C_{1i})p_{ii} > C_3/N_0$, 则存在唯一正解: $T = T_0^p$ 满足:

$$\sum_{i=1}^{2}(C_{2i} - C_{1i})\lambda_i(T) = C_3,$$

且:

$$T^* = \min(T_0^p, t).$$

(2) 如果 $\sum_{i=1}^{2}(C_{2i} - C_{1i})p_{ii} \leq C_3/N_0$, 则:

$$T^* = 0.$$

8.2.3 综合考虑

对于可靠性指标和成本指标单独予以考虑而确定的最佳投放时间,往往很难同时满足这两个指标。综合考虑的目的即在于要使选定的最佳投放时间能同时满足可靠性指标和成本指标。

权衡利弊,可靠性指标往往还是显得更重要些,因此有时为求一定的高可靠性,牺牲一点成本也是应该的。因此,对于综合考虑这二者,则一般的作法就是:在达到规定的可靠性指标之后,使软件开发的总成本达到最小值。我们将这一问题称为在可靠性指标约束之下的成本函数的优化问题。

下面我们先选取 Goel 和 Okumoto 的 G-O 模型为基础,讨论考虑到可靠性和成本两个指标时,怎样确定最佳投放时间的方法。然后介绍 Yamada 和 Osaki 基于软件错误分为两类的 NHPP 模型,确定软件最佳投放时间的方法。最后,以 Yamada-Osaki 模型为基础,分别给出以成本指标、可靠性指标,以及以成本—可靠性指标为综合考虑的具体数值例子。

为了确定最佳投放时间,首先也要建立成本模型。设:

C_1 为测试期间改正一个错误的成本,

C_2 为运行期间改正一个错误的成本($C_2 > C_1$),

C_3 为每单位时间的测试成本,

t 为软件寿命周期的长度,

T 为软件投放时间(即测试时间)。

因为 $m(t)$ 表示在 $(0, t)$ 区间内的错误数期望值,则在测试期间改正错误的成本为 $C_1m(T)$,而在运行期间改正错误的成本为 $C_2[m(t) - m(T)]$,测试成本则为 C_3T,则总成本可望为:

$$\begin{aligned} C(T, t) &\equiv C(T) \\ &= C_1m(T) + C_2[m(t) - m(T)] + C_3T \end{aligned} \tag{8.2.3}$$

我们的目的在于通过求 $C(T, t)$ 的极小值,求出最佳值 T^*。对(8.2.3)关于 T 微分,则:

$$\frac{d[C(t)]}{dT} = C_1 m'(T) - C_2 m'(T) + C_3.$$

令:
$$C_1 m'(T) - C_2 m'(T) + C_3 = 0,$$
则:
$$m'(T) = C_3/(C_2 - C_1) = \lambda(T). \tag{8.2.4}$$

根据 G-O 模型, 我们有:
$$\lambda(T) = N_0 e^{-T} = C_3/(C_2 - C_1).$$

注意, $\lambda(T)$ 是单调递减函数, 且 $\lambda(0) = N_0$. 如果 $N_0 \leq C_3/(C_2 - C_1)$, 则由于 $T \geq 0$, $d[C(T)]/dT > 0$, 所以(8.2.4)不存在有理解。见图 8.3。因此, 在这种情况下, $C(T)$ 的极小值在 $T = 0$ 处, 即 $T^* = 0$.

如果 $N_0 > C_3/(C_2 - C_1)$, 则(8.2.4)存在唯一解, 由下式给出:
$$T_0 = (1/\beta)\ln\frac{N_0(C_2 - C_1)}{C_3}.$$

对于 $0 < T < T_0$, 有 $d[C(T)]/dT < 0$; 对于 $T > T_0$, 有 $d[C(T)]/dT > 0$; 因此, $C(T)$ 的极小值在 $T_0 \leq T$ 时, 存在于 $T = T_0$ 处, 而对于 $T_0 > t$, 则存在于 $T = t$ 处。

注意, 如果最小期望成本超过了获得的运行收益, 则该软件也就不用进行测试了。

对于上述讨论总结如下:
$$C(T) = C_1 m(T) + C_2[m(t) - m(T)] + C_3 T$$

为使 $C(T)$ 达到最小, 需满足:
$$R(x \mid T) \geq R_0, \quad T \geq 0,$$
$$C_2 > C_1 > 0, \quad C_3 > 0, \quad x \geq 0, \quad 0 < R_0 < 1.$$
则:
$$T^* = \max(T_0, T_1),$$
其中,
$$T_0 = (1/\beta)\ln[N_0(C_2 - C_1)/C_3], \quad N_0(C_2 - C_1) > C_3;$$
$$T_0 = 0, \quad \text{否则} \tag{8.2.5}$$
$$T_1 = (1/\beta)\ln[m(x)] - \ln[\ln(1/R_0)], \quad R(x \mid 0) < R_0;$$
$$T_1 = 0, \quad \text{否则}. \tag{8.2.6}$$

可以有下面四种情况:

(1) 如果 $N_0 > C_3/(C_2 - C_1)$, 且 $R(x \mid 0) < R_0$, 则存在唯一大于等于零的 T_0 和 T_1 分别满足(8.2.5)和(8.2.6), 而且有:
$$T^* = \max(T_0, T_1).$$

(2) 如果 $N_0 > C_3/(C_2 - C_1)$, 且 $R(x \mid 0) \geq R_0$, 则:
$$T^* = T_0.$$

(3) 如果 $N_0 \leq C_3/(C_2 - C_1)$, 且 $R(x \mid 0) < R_0$, 则:
$$T^* = T_1.$$

(4) 如果 $N_0 \leq C_3/(C_2 - C_1)$, 且 $R(x \mid 0) \geq R_0$, 则:
$$T^* = 0.$$

图 8.3 $\lambda(T)$ 与 T 的关系

例　同上例,取:

$$N_0 = 1348, \quad = 0.124, \quad x = 0.1, \quad R_0 = 0.70,$$

而且令:

$$C_1 = 1, \quad C_2 = 5, \quad C_3 = 100, \quad t = 100.$$

应用上述方法,可求得:

$$T^* = 15.3 \quad \text{或} \quad T^* = 30.9.$$

但在 $T^* = 15.3$ **时,有:**

$$R(0.1 \cdot T^*) = 0.08,$$

因为, $N_0 > C_3/(C_2 - C_1)$ **且** $R(x \cdot 0) < R_0$**,则:**

$$T^* = \max(15.3, 30.9) = 30.9.$$

据此可计算出:

$$C(T^*) = 4555.$$

结果如图 8.4 **所示。**

Yamada **和** Osaki **将这一模型推广至他们的分类模型上去,于是确定最佳投放时间的问题就可以下面的形式予以表述:**

对 $C(T)$ **求极小**

使满足: $R(x \cdot T) \geq R_0, \quad T \geq 0,$

$$C_{2i} > C_{1i} > 0 \ (i = 1, 2),$$

$$C_3 > 0, \quad x \geq 0,$$

$$0 < R_0 < 1.$$

(8.2.7)

对于具体的成本函数 $C(T)$**,可以求出** T^***:**

(1) **如果** $\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_{ii} > C_3/N_0$**,且** $R(x \cdot 0) < R_0$**,则存在唯一的正** T_0 **和** T_1**,满足:**

$$\sum_{i=1}^{2} (C_{2i} - C_{1i}) \lambda(T) = C_3, \quad (T = T_0)$$

$$R(x \cdot T) = R_0 \quad (T = T_1)$$

且:

$$T^* = \max(T_0, T_1).$$

(2) **如果** $\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_{ii} > C_3/N_0$**,且** $R(x \cdot T) \geq R_0$**,则** $T^* = T_0$.

(3) **如果** $\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_{ii} \leq C_3/N_0$**,且** $R(x \cdot T) < R_0$**,则** $T^* = T_1$.

(4) **如果** $\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_{ii} \leq C_3/N_0$**,且** $R(x \cdot T) \geq R_0$**,则** $T^* = 0$.

对于上面的解,我们都假定:

$$t > \max(T_0, T_1).$$

图 8.4　**成本—可靠性最佳投放策略的确定**
($C_1 = 1, C_2 = 5, C_3 = 100, T_{LC} = 100$)

(t 为软件寿命周期的长度)

表 8.4 是在 N_0= 1,000, $_1$= 0.1, $_2$= 0.01, p_1= 0.9, p_2= 0.1, C_1= 1, t= 100 时,计算出来的最佳软件投放时间,它是以可靠性指标为考虑问题的出发点的情况。

表 8.4 软件最佳投放时间

C_2 \ C_3	10	20	30	40	50	100
2	22.8	15.5	11.3	8.3	6.1	0
3	30.5	22.8	18.5	15.5	13.2	6.1
4	35.3	27.2	22.8	19.7	17.4	10.2
5	39.0	30.5	25.9	22.8	20.4	13.2
6	42.0	33.1	28.4	25.2	22.8	15.5
7	44.7	35.3	30.5	27.2	24.8	17.4
8	47.2	37.3	32.3	29.0	26.5	19.0
9	49.5	39.0	33.9	30.5	28.0	20.4
10	51.6	40.6	35.3	31.9	29.3	21.7
20	74.6	52.7	45.6	41.3	38.3	29.9
30	100	63.4	53.1	47.8	44.2	34.9
40	100	76.0	60.1	53.2	49.0	38.7
50	100	91.9	67.6	58.5	53.3	41.8
60	100	100	76.5	63.9	57.5	44.5
70	100	100	86.9	69.9	61.8	46.9
80	100	100	98.1	76.7	66.4	49.2
90	100	100	100	84.4	71.3	51.4
100	100	100	100	92.7	76.9	53.6

从表 8.4 中可看出,操作运行维护成本(C_2)越高,软件投放时间越长;测试成本(C_3)越高,软件最佳投放时间就越短。也就是说,软件系统投放越晚,对于高成本的操作运行维护而言是节省成本的;而对于高成本的测试阶段而言,软件系统投放得越早,越节省成本。

表 8.5 示出以成本指标为考虑问题出发点的情况,它揭示出的规律与上述相同(N_0 = 1,000, $_1$= 0.1, $_2$= 0.01, p_1= 0.9, p_2= 0.1, C_{11}= 1, C_{21}= 10, C_3= 100, t= 100).

表 8.5

C_{22} \ C_{12}	1	2	3	5	10
10	21.7	21.6	21.5	21.3	20.9
20	22.6	22.5	22.4	22.2	21.8
30	23.5	23.4	23.3	23.1	22.7
40	24.6	24.5	24.3	24.1	23.6

图 8.5 显示出初始错误个数 N_0 在软件最佳投放时间问题中表现出的独立性(以成本指标为出发点)。图 8.6 则示出以可靠性指标为出发点时的 N_0 的独立性。

表 8.6 列出以可靠性指标为出发点的软件最佳投放时间(N_0= 1,000, $_1$= 0.1, $_2$= 0.01, p_1= 0.9, p_2= 0.1)。

图 8.5　考虑成本指标时的初始错误个数的独立性

($_1$= 0.1, $_2$= 0.01, p_1= 0.9, p_2= 0.1, C_1= 1, C_2= 10, C_3= 100, t= 100)

图 8.6　考虑可靠性指标时的初始错误个数的独立性

($_1$= 0.1, $_2$= 0.01, p_1= 0.9, p_2= 0.1, x= 2.0)

表　8.6

R_0 \ x	0.01	0.05	0.1	0.5	1	2
0.5	2.7	19.3	26.8	47.4	62.7	105.6
0.6	5.8	22.5	30.2	53.1	75.7	135.5
0.7	9.5	26.5	34.4	62.1	103.3	171.4
0.8	14.3	31.8	40.5	84.7	149.5	218.3
0.9	22.2	41.3	52.6	155.5	224.5	293.4

从表 8.6 中可以看出:高的可靠性指标就意味着更长的投放时间。

最后,看看以成本-可靠性综合指标为出发点的情况(取: $N_0 = 1,000$, $\alpha_1 = 0.1$, $\alpha_2 = 0.01$, $p_1 = 0.9$, $p_2 = 0.1$, $x = 0.2$, $R_0 = 0.7$, $C_{11} = 1$, $C_{12} = 2$, $C_{21} = 5$, $C_{22} = 10$, $C_3 = 100$, $t = 100$)。应用(8.2.7),可求出: $T_0 = 14.10$, $T_1 = 45.16$. 因为:

$$\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_i \alpha_i = 0.3608,$$

$$C_3 / N_0 = 0.1,$$

所以:

$$\sum_{i=1}^{2} (C_{2i} - C_{1i}) p_i \alpha_i > C_3 / N_0,$$

且:

$$R(0.2 \cdot T_0) = R(0.2 \cdot 14.10) = 9.54 \times 10^{-3} < 0.7 = R_0.$$

于是有:

$$T^* = \max(T_0, T_1) = 45.16.$$

示于图 8.7。

图 8.7 成本-可靠性综合指标

($N_0 = 1000$, $\alpha_1 = 0.1$, $\alpha_2 = 0.01$, $p_1 = 0.9$, $p_2 = 0.1$,
$C_{11} = 1$, $C_{12} = 2$, $C_{21} = 5$, $C_{22} = 10$, $C_3 = 100$, $t = 100$)

§ 8.3 根据软件的模块结构判定最佳投放时间

Y. Masuda 等人根据“在软件执行期间的任一时刻,只有 k 个模块中的一个在执行”的基本认识,开发出根据软件的模块结构判定最佳投放时间的模型。

模型中用到的符号及意义:

k　一软件系统所拥有的模块总数

T　测试期间的时间长度

α　在软件测试期间,软件系统的使用时间所占份额

α　在软件投放之后,软件系统的使用时间所占份额

β　在软件系统执行过程中,模块 j 的执行时间所占份额,且 $\sum_{j=1}^{k} \beta = 1$

N_j　模块 j 中的错误个数

X_j　在模块 j 中为查出一个特定的错误,模块 j 需要的执行时间

r_j　到目前为止,在模块 j 中查出的错误个数

$t_j(k)$　在模块 j 中,错误 k 被查出的时间,$k = 1, 2, \cdots, r_j$

$V(\tau)$　在测试期间,经过 τ 时刻,投放软件系统所获得的收益

$U(t)$　在 $t = 1$ 处的阶梯函数:$U(t) = \begin{cases} 1, & \text{如果 } t \geq 1; \\ 0, & \text{否则} \end{cases}$

$u(t) = 1 - U(t)$

t_D　软件系统的交货时间

W　软件系统的寿命

根据上述的基本认识,给定一段时间 T 以测试软件,就是假定了 $\alpha \cdot T$ $(0 < \alpha < 1)$ 是用于测试的,而在其它时间,$(1-\alpha)T$,软件并未被用到。在软件系统执行过程中,用于模块 j 的时间份额只占 β。有理由认为,测试期间的空闲概率,$1-\alpha$ 与软件系统在投放以后的使用过程中的空闲概率是不同的。为了区分它们,我们将后者表示成 $1-\alpha$。另一方面,β 又紧密依赖于软件系统的模块结构。

模型的假设:

1. 即使在软件系统交货之后,β_j $(j = 1, 2, \cdots, k)$ 仍然是不变的,就是说,测试的条件是对实际条件的模拟,而且软件系统的使用期是相当长的。

2. 模块 j 中的 N_j 个错误的每个错误,都有一个随机寿命 X_j,即,当在处理模块 j 的累积 CPU 时间达到 $X_j(w)$ 时,该错误即被查出。

3. X_j 服从带有为正的参数 (a_j, b_j) 的 Weibull 分布:

$$F_j(x) = P[X_j \leq x] = 1 - e^{-a_j x^{b_j}} \qquad (8.3.1)$$

当测试条件模拟真实条件时,参数 (a_j, b_j) 保持不变。当 $b_j = 1$ 时,$F_j(x)$ 为指数分布。

4. 所有的随机变量都是统计独立的。

5. 一旦查出错误,立即排错,且排错是完全的。

6. 测试期间,花在处理模块 j 上的 CPU 时间为平均值: $\beta\alpha T$.

在测试期间,收集到的故障数据为完全数据: $(t_j(1), t_j(2), \cdots, t_j(r_j))$, $j = 1, 2, \cdots, k$。对于这些故障数据,任务就是:

a. 找出估计值 $N_j, (a_j, b_j)$, $j = 1, 2, \cdots, k$.

b. 判断测试过程是否要继续下去?

下面讨论参数估计的方法。

1. 一个系统有 J 个分支,每个分支的寿命都是统计独立且恒同分布的,它们都服从 Weibull 分布,且具有累积分布函数:

$$F(x) = 1 - e^{-ax^b} \tag{8.3.2}$$

2. 在时间区间 $[0, S]$ 内, r 次观察时间有关系: $0 \leq x(1) \leq x(2) \leq \cdots \leq x(r) \leq S$。可以定义 $J \equiv N_j$, $S \equiv \beta\alpha T$, $x(i) \equiv \beta\alpha t_j(i)$,这样就能很容易地将结果应用于上面讨论的模型。下面分几种情况来讨论。

① 指数分布的情形: 这时有 $b = 1$, J 和 a 未知。

此时寿命分布具有一共同的概率密度函数:

$$f(x) = ae^{-ax} \tag{8.3.3}$$

先假定 J 和 a 已知,于是似然函数为:

$$L(J, a) = \begin{cases} a^r e^{-aS(y+J+r)} \prod_{i=1}^{r} (J - i + 1), & r \geq 1; \\ e^{-aJS}, & r = 0. \end{cases}$$

$$y \equiv \sum_{i=1}^{r} x(i)/S$$

于是最大似然函数为:

$$a = r/[S(y + J - r)]$$

$$J = \begin{cases} r, & 0 \leq y \leq r/\sum_{i=0}^{r} i^{-1}; \\ \infty, & r \geq 2 \text{ 且 } y \geq (r+1)/2; \\ k, & (\text{这时, } k \text{ 使 } m_{k,r} \leq y \leq m_{k+1,r} \text{ 成立}) \end{cases}$$

$$r \geq 2 \text{ 且 } r/\sum_{i=0}^{r} i^{-1} \leq x \leq (r+1)/2.$$

$$m_{k,r} \equiv \{1 - [(k-r)/k]^{1/r}\}^{-1} - k + r, \quad k \geq r.$$

如果 J 已知,则关于 a 的最大似然函数由 a 的表达式给出,其中以 J 代替 J 即可。

② Weibull 分布的情形: J 已知, a 和 b 未知。

这时寿命分布的共同概率密度函数与似然函数为:

$$f(x) = abx^{b-1}e^{-ax^b} \tag{8.3.4}$$

$$L(a,b)=\begin{cases}(ab)^r\prod_{i=1}^{r}(J-i+1)x(i)^{b-1}e^{-ax(i)^b}e^{-a(J-r)S^b}, & r\geq 1;\\ e^{-aJS^b}, & r=0.\end{cases}$$

$\ln L(a,b)$ 的平稳点满足:

$$a=r/g(b);\quad g(b)\equiv\sum_{i=1}^{r}x^b(i)+(J-r)S^b \tag{8.3.5}$$

$$1/b=-\frac{1}{r}\sum_{i=1}^{r}\ln x(i)+\frac{d}{db}\ln g(b) \tag{8.3.6}$$

解上述方程组,求 a 与 b 即可得出关于 a,b 的最大似然方程组,如:

$$\frac{d^2}{db^2}\ln g(b)\geq 0,\quad (b>0);$$

则 $g(b)$ 是具有对数凸性的。进一步有:

$$\frac{d}{db}\ln g(b)\big|_{b=0}>\frac{1}{r}\sum_{i=1}^{r}\ln x(i).$$

它可以保证(8.3.6)有唯一解。关于 a 的最大似然函数,可以通过设 $a=r/g(b)$ 从(8.3.5)得出。当 $r=0$ 时,可以推出 $a=0$,b 不定。

下面讨论软件投放时间的判定方法。首先引入在测试 T 时间之后,进一步测试 τ 时以后,再投放软件的获益目标函数:

$$v(\tau)=V_1(T+\tau)-V_2(\tau)-V_3(T+\tau).$$

其中, $V_1(t)$ 表示软件系统在时刻 t 时的值;

$V_2(t)$ 表示因未排除软件错误而引起的平均代价;

$V_3(t)$ 表示投放软件的时刻定在 t 时,软件测试的累积运行成本。

进一步假设:

1. $V_1(t)=\lambda e^{-\lambda_2 t}U(t_D-t)+\mu e^{-\mu_2 t}U(t_D-t)$,

其中, λ 和 $\mu_i(i=1,2)$ 是正的常数。要判定 $V_1(t)$ 的形式可能是困难的,但有理由假定 $V_1(t)$ 是 t 的减函数。进而言之,如果软件投放时间是合同规定的,则软件系统的值在超出投放时间 t_D 以后,就要掉价。

2. 如果在软件系统投放之后,在 W 期间内又查出模块 j 中的一个错误,则 C_j 的成本是额外附加的。W 可以表示成软件系统的寿命。

3. $V_3(t)$ 是一个线性函数, γ_1, γ 是一个正的常数。

于是在 T 时刻模块 j 中的错误为 (N_j-r_j) 个。设 $F_j(t)$ 由(8.3.1)式给出,其中参数 a_j 和 b_j 由 a_j 和 b_j 代替,则在时间段 $[T,T+\tau]$ 内发现 n_j 个错误的概率为:

$$P_{n_j}=\binom{N_j-r_j}{n_j}\frac{1}{\bar{F}_j(\beta\alpha T)^{N_j-r_j}}\{F_j(\beta\alpha(T+\tau))-F_j(\beta\alpha T)\}^{n_j}$$

$$\times\{\bar{F}_j(\beta\alpha(T+\tau))\}^{(N_j-r_j-n_j)},\quad n_j=0,1,\cdots,N_j-r_j.$$

$$\bar{F}_j(x)=1-F_j(x).$$

设在时间段 $[T,T+\tau]$ 内查出 n_j 个错,到时间 $T+\tau+W$ 为止查出 L_j 个错误的概率就变

成:

$$P_{l_j}(\eta_j) = \binom{N_j - r_j - n_j}{l_j} \bar{F}_j(\beta\alpha(T+\eta))^{-(N_j-r_j-n_j)} \{F_j(\beta(\alpha(T+\eta)+\alpha W)) - F_j(\beta\alpha(T+\eta))\}^{l_j} \{\bar{F}_j(\beta(\alpha(T+\eta)+\alpha W))\}^{(N_j-r_j-n_j-l_j)},\quad l_j = 0,1,\cdots,N_j - r_j - n_j.$$

因此,由于未被查出的错误造成的代价为:

$$V_2(\eta) = \sum_{j=1}^{k} C_j(N_j - r_j)\frac{F_j(\beta(\alpha(T+\eta)+\alpha W)) - F_j(\beta\alpha(T+\eta))}{\bar{F}_j(\beta\alpha T)}.$$

如果 $W=+\infty$, $V_2(\eta)$ 就简化为:

$$V_2(\eta) = \sum_{j=1}^{k} C_j(N_j - r_j)/\bar{F}_j(\beta\alpha T).$$

以在时间区间 $[0,T]$ 内对测试结果的观察为基础,可以估计出 N_j, a_j, b_j,然后计算 $v(0)$ 和 $v(\Delta)$(对某 $\Delta>0$),并比较结果。如 $v(0)\geq v(\Delta)$,则停止测试并投放软件;否则,继续测试到 $T+\Delta$。然后对 $[0,T+\Delta]$ 重复上述过程,直到软件投放时间被确定为止。

例　具有 5 个模块的一个软件系统,且 $\alpha=0.0549$, $\alpha=0.0337$,在下面的讨论中,除特别申明的以外,时间单位为周。

测试期间,在一天(24 小时)内,79 分钟 CPU 时间用于测试软件。每个模块所占 79 分钟内的时间分别为:

5.61, 4.54, 11.90, 34.00, 22.95

在投放之后,软件的使用时间由 79 分钟/天降为 48.5 分钟/天,β's 保持不变。

$\beta=0.0717$, $\beta=0.0574$, $\beta=0.1505$, $\beta=0.4301$, $\beta=0.2903$.

$t_D=20$, $W=50$, $\lambda=500$, $\lambda=0.01$, $\mu=300$, $\mu=0.01$.

C_j: $(C_1,C_2,C_3,C_4,C_5)=(2,2,2.5,3,3)$.

每周用于该软件测试的成本为 $\gamma=5$。取 $T_0=5$, $\Delta=1$,作判断如下。取 $(N_1,N_2,N_3,N_4,N_5)=(30,30,40,50,50)$,对于模块 j,产生出 N_j 个 Weibull 随机变量,如 $t_j(i)$, $1\leq i\leq N_j$(按升序), $r_j=\max(i\mid t_j(i)\leq\beta\alpha T)$,下面分 A、B 两种方法讨论参数值的估计。

方法 A:指数分布。

寿命分布全为指数分布, $b_j=1$。A_1:由上面讨论的指数分布的最大似然函数估计 a, N_j 的值。这时 N_j 未知,不妨令 $N_j=+\infty$。如果对于某个 j 正好如此,那么 $v(0)$ 和 $v(1)$ 就不被定义。在这种情况下,测试时间就要自动延长一周。A_2:给定 N_j,只有 a_j 可通过 $a=r/S(y+J-r)$ 来计算其值。

方法 B:Weibull 分布。

N_j 的值已知,用(8.3.5)和(8.3.6)式可估计 a 和 b 的值。然后接着用 $v(\eta)$ 的公式评价目标值 $v(0)$ 和 $v(1)$。如 $v(0)>v(1)$,于是软件投放于时刻 T,否则,令 $T=T+1$,重复评价过程。设这时投放时间用 T^* 表示。

对于一给定的 t,设 $l_j(t)$ 为满足 $\beta\alpha t<t_j(i)\leq\beta(\alpha t+\alpha W)$ 的 $t_j(i)$ 的个数。定义:

$$v_{act}\equiv V_1(t) - \sum_{j=1}^{k} c_j l_j(t) - V_3(t) \qquad (8.3.7)$$

则对于每个样本路径,在 T^* 时投放软件的实际收益为:

$$v^* = v_{act}(T^*) \tag{8.3.8}$$

给定产生出的随机数 $t_j(i)$ 的集合, $1 \leq i \leq N_j$, $1 \leq j \leq k(k=5)$. 于是:

$$v^{**} = v_{act}(T^{**}) = \sup\{v_{act}(T_0 + k\Delta): k = 0, 1, 2, \cdots\} \tag{8.3.9}$$

根据 Littlewood 的观点,软件错误的寿命分布属于那些有着递减故障率的分布类,据此我们下面将分析集中于有着递减故障率的 Weibull 分布。这时有 Weibull 分布的故障率为:

$$\eta(x) = \frac{f(x)}{F(x)} = abx^{b-1} \tag{8.3.10}$$

因此,可以断定一个 Weibull 随机变量是有着递减故障率的,当且仅当 $b \leq 1$。

(a_j, b_j), $(j = 1, 2, \cdots, 5)$ 的参数估计值见表 8.7。

表 8.7 参数估计值

case		模块				
		M_1	M_2	M_3	M_4	M_5
case 1	a	20	20	10	4	4
	b	1	1	1	1	1
case 2	a	10.78	10.78	6.41	3.22	3.22
	b	0.75	0.75	0.75	0.75	0.75
case 3	a	6.32	6.32	4.47	2.83	2.83
	b	0.5	0.5	0.5	0.5	0.5

case 1 对应于 $b_j = 1$ 的软件错误的指数型寿命。case 2 对应于 $b_j = 0.75$ 的递减故障率 Weibull 分布的寿命。case 3 对应于 $b_j = 0.5$ 的递减故障率 Weibull 分布的寿命。参数 a_j 的估计值,对于这三种情况都按 $E[X_1] = E[X_2] = 0.05$, $E[X_3] = 0.1$, $E[X_4] = E[X_5] = 0.25$ 的原则选定。每种情况都产生出 200 个样本。每个样本产生出软件投放时间 T^* 和实际收益 v^* (见式(8.3.8)),且产生最佳投放时间 $T^{**} = T_0 + k\Delta$ 以及随之得出的最佳收益 v^{**} (见式(8.3.9))。

§ 8.4 根据软件运行期间查错情形判定最佳投放时间

H. Ohtera 和 S. Yamada 发表了一个不仅考虑到测试,而且也考虑到运行期间的查错过程中,根据软件可靠性目标和平均无故障时间两个指标,来确定软件最佳投放时间的模型。

与以前的作法不同,他们提出在软件的测试期间和运行期间的查错过程是不相同的。他们规定,测试努力包括花在软件测试上的资源,如人力、CPU 小时、以及测试的条件与

数据等;而运行努力包括花在软件操作上的资源,如 CPU 小时等。

模型中用到的数学符号及意义如下:

t, s	分别代表测试与运行阶段的时间段
m(t), M(s)	分别代表在 NHPP 模型中的均值函数
w(t)	花费在测试时间 t 中的当前的测试努力
W(t)	$W(t)=\int_0^t w(x)dx$
p(s)	花费在运行时间 s 中的当前的运行努力
P(s)	$P(s)=\int_0^s p(x)dx$
a	初始错误的期望数, a> 0
a- m(t)	在测试终止时刻,软件中的剩余错误的期望数
r_1 r_0	分别代表每测试努力单位和每运行努力单位的错误查出率,且 0< r< 1, 0< r_0< 1
d(t)	每个错误的错误查出率: $d(t)=\frac{d}{dt}m(t)/[a-m(t)]$
X_k	(k- 1)与 k 次故障间的故障间隔时间, (k= 1, 2, …)
α β, m	测试努力函数 w(t)和 W(t)中的常数, α> 0, β> 0, m> 0.
R_0, S_0	在软件投放时刻的软件可靠性目标值和平均无故障时间
*	表示最佳投放时间

模型的假设如下:

1. 测试努力的时间独立行为满足 Weibull 分布,而在整个运行阶段,当前运行努力为一常数。

2. 在测试时间段(t, t+ Δt]内,查出的错误的期望与测试努力的总量之比,正比于软件系统中的剩余错误的期望。

3. 在运行时间段(s, s+ Δs]中查出的错误的期望与运行努力的总量之比,正比于软件中的剩余错误的当前期望。

4. 在测试阶段和运行阶段的查错情况,用 NHPP 模型描述。

测试阶段:

软件可靠性增长模型为:

$$\begin{aligned} &P_r\{N(t)=n\}=poim(n:m(t)), \quad n=0,1,2,\cdots \\ &m(t)=a\{1-\exp[-rW(t)]\}, \quad a>0, 0<r<1 \\ &W(t)=\int_0^t w(x)dx \end{aligned} \tag{8.4.1}$$

由实际的测试努力数据所表现出的开销模式多种多样,模型作者认为在测试阶段,Weibull 测试努力函数在描述测试努力行为方面的灵活性,可以将测试表示为:

$$w(t)=\alpha\beta m t^{m-1}\exp(-\beta t^m), \quad \alpha>0, \beta>0, m>0 \tag{8.4.2}$$

上式中参数 α表示进行软件测试到要求的测试努力的总量。当 m= 1 和 m= 2 时,要分别得出指数测试努力函数和 Rayleigh 测试努力函数。作为软件可靠性增长标志的每个错误

的错误查出率为:

$$d(t) = r \cdot w(t) \tag{8.4.3}$$

(8.4.2)式中的参数 α、β、m 可以用最小二乘法估计,(8.4.1)式中的参数 a、r 可以用最大似然法估计。

运行阶段:

在软件工程界,大家都了解,在用户运行阶段的查错情况与软件测试阶段是不相同的。但在为软件可靠性建模时,却并未明显地反映出这一事实。为了反映出运行阶段的查错过程的时间独立性行为,模型的作者假设,软件运行阶段也可被软件开发管理者作为一个查错过程来对待。于是,如同在上面对测试阶段所讨论的那样,在软件运行阶段的查错过程也可以用 NHPP 模型描述。

因为在软件测试的结束时刻 t,软件中剩余错误的期望是 $a - m(t)$,所以有:

$$\begin{aligned} &P_r\{N(s) = n\} = \mathrm{poim}(n: M(s)), \qquad n = 0, 1, 2, \ldots \\ &M(s) = [a - m(t)] \cdot \mathrm{expf}[r_0 P(s)], \\ &P(s) = \int_0^s p(x)\,dx, \end{aligned} \tag{8.4.4}$$

(注: $\mathrm{expf}(u) \equiv 1 - \exp(-u), (u \geq 0)$)

则在运行阶段花费的运行努力是:

$$\beta(s) = \delta, \quad \delta > 0 \tag{8.4.5}$$

总的说来,在测试阶段和运行阶段的每个错误的错误查出率的关系为:

$$r \cdot w(t) > r_0 \cdot \delta \tag{8.4.6}$$

于是可以假定,软件错误的查错率 $r_0 \cdot \delta$ 根据软件开发的经验,可以在软件运行阶段之前就已经被估计出来了。

图 8.8 表示出在测试阶段和运行阶段,每个错误的错误查出率与时间的关系。

图 8.8 在测试阶段和运行阶段每个错误的错误查出率的行为

下面谈谈关于软件最佳投放时间的问题。

1. 软件可靠性目标: R_0

G-O 模型可以用作解决最佳投放时间问题的评估标准。如图 8.9(a)所示, G-O 模型给出在时间段(t, t+ s]内第 k 个故障不出现的概率, 在这一给定的条件下, 在时刻 t, 当第(k- 1)个故障已经出现。于是基于上面讨论的 NHPP 模型的软件可靠性为:

$$R(s\mid t) \equiv P_r\{X_k > s\mid S_{k-1} = t\}$$
$$= \exp[-\ a\exp f_c(rW(t))\exp f(r_0\delta s)], \quad S > 0 \qquad (8.4.7)$$

注: f_c: S_f(关于连续随机变量, 有 Cdf{·}+ S_f{·}= 1)

$Cdf\{x\} \equiv P_r\{X \le x\}$, $S_f\{x\} \equiv P_r\{X \ge x\}$

其中,
$$S_{k-1} = \sum_{i=1}^{k-1} X_i.$$

由上式给出的软件可靠性与故障个数 k 无关, 并且给出在运行时间段(0, s]内不出现故障的概率。于是, 当当前可靠性 R(s|t)对于一段特定的运行时间 s 等于可靠性目标值 R_0 时, 对(8.4.7)式求解得出时间 t, 就给出一个最佳投放时间:

$$(t^*)^m = -\ \frac{1}{\beta}\ln\ \frac{1}{\alpha}\ln\ \frac{-\ \ln R_0}{a\exp f(r_0\delta s)}\ +\ 1 \qquad (8.4.8)$$

2. MTBF 目标: S_0

假定 MTBF 可以从软件可靠性增长模型中的均值函数计算出来。对于特定的运行时间 s, 由求解下面的(8.4.9)式, 求解 MTBF 方程可解出 x:

$$M(s+\ x) -\ M(s) =\ 1 \qquad (8.4.9)$$

方程(8.4.9)有一故障在运行时间段(s, s+ x]内出现, 即, 满足它的时间 x, 是在运行时刻 s, 为查出一个错误所要求的时间的期望(见图 8.9(b))。于是, 在 s= 0 的条件下的最佳投

图 8.9　作为最佳软件投放问题的评估标准的
软件可靠性与 MTBF

放时间由下面的作法给定: 当 MTBF 由解(8.4.9)式给定时的测试时间, t^*, 等于目标值 S_0($M(S_0)=1$), 该测试时间 t^* 即为最佳投放时间:

$$(t^*)^m = -\frac{1}{\beta}\ln\frac{1}{\alpha}\ln\frac{1}{a\exp f(r_0\delta S_0)} + 1 \qquad (8.4.10)$$

例　根据 Brooks 和 Motley[1980]引用的[B10]的 DS1, Yamada 等人估计出:

$\alpha = 2253.2$, $\beta = 4.5343\times 10^{-4}$, $m = 2.2580$, $a = 1397.6$, $r = 1.5791\times 10^{-3}$, $r_0 = 1.5791\times 10^{-2}$, $\delta = 1$, $R_0 = 0.9$, $s = 0.1$ 有:

$t^* = 40.47$(见图 8.10).

对于 $S_0 = 1$, 有

$t^* = 41.31$(见图 8.11)。

图 8.10　使用实际数据, 基于软件可靠性标准的最佳软件投放时间

图 8.11　使用实际数据, 基于 MTBF 标准的最佳软件投放时间

§ 8.5　根据测试与排错判定最佳投放时间

N. D. Singpurwalla 以如何在不确定的情况下进行决策的思想为基础, 提出两个判定在软件投放前应测试和排错多长时间, 并使软件的实用性达到最大的实用性函数(utility function): 它们分别以成本指标和软件可靠性指标作为控制。他指出根据软件可靠性的概率模型使用在软件故障数据的结果。对于单个状态测试的情况, 该最优化问题可以用非数值化的技术求解。

设 $\tilde{T}_1, \tilde{T}_2, \ldots,$ 表示在一正在变化的条件下进行测试的软件故障间的一系列运行时间(以下时间均指 CPU 时间)。设 $\tilde{T}_i$ 能用具有一参数 $\Lambda(N-i+1)$ 的指数分布来描述它, 其中 $\Lambda(x)$ 表示 x 为一未知的比例常数, N 是软件中未知的错误个数。于是有:

$$P(\tilde{T}_i \geq \tau \mid N, \Lambda) = \exp\{-\Lambda(N-i+1)\tau\}, \quad \tau \geq 0 \qquad (8.5.1)$$

又假设 Λ 满足含标参 μ 和形参 α 的 Γ-分布，N 满足含参数 θ 的泊松分布，且关于 N 和 Λ 的上述分布是独立的。

设对软件测试了 T 个时间单位，每个时间单位查出一个错误，并予以改正(完全数据)；在 T 时间单位之后，软件投放给用户使用。这一种情况称为单阶段测试。如软件在测试 T 时间单位之后，又接着测试了 T^* 个时间单位，然后投放使用，则称这种情况为双阶段测试。关于是否要再测试一段时间 T^*，然后投放的决策，依赖于 $M(T)$，$M(T)$ 表示在时间 T 内所查出的错误个数。对于 T 和 T^* 的选择，必须要先于观察到的 $M(T)$。因此，关于究竟是确定 T 的最佳值，还是 T 和 T^* 的最佳值，必须要由测试是单阶段的还是双阶段的而定。注意：单阶段测试可以看作一系列阶段测试的循环过程，每一阶段关于最佳测试时间 T 的判定，都由前一阶段的测试结果而定。由于对双阶段测试问题的讨论，要涉及到巨量的计算，所以，这里只限于对单阶段测试的讨论。

设 T 已被指定，$M(T)=k\geq 0$，$t_1, t_2, \ldots, t_k$ 为故障间的 CPU 运行时间，且任何时候都有 $k\geq 1$。令：

$$t^{(k)}=\begin{cases}(t_1,\ldots,t_k,T-\sum_{i=1}^{k}t_i), & k\geq 1\\ T, & k=0\end{cases}$$

给定 $t^{(k)}$ 和 k，则 N 和 Λ 分别在 q 和 λ 上的似然函数正比于：

$$\frac{q!}{(q-k)!}\lambda^{k}\exp\left\{-\lambda\left[\sum_{j=1}^{k}(q-j+1)t_j+(q-k)\left(T-\sum_{j=1}^{k}t_j\right)\right]\right\} \qquad (8.5.2)$$

令 $Y_i=\sum_{j=1}^{i}\tilde{T}_j$，则当 $M(T)\geq 1$ 时，可证：

$$t(T)=\frac{1}{T}\sum_{j=1}^{M(T)}(M(T)-j+1)\tilde{T}_j$$

和 $M(T)$ 一起对于 N 和 Λ 就是充分的。于是根据贝叶斯定律，N 的后验分布为：

$$P(N=q\geq M(T)\mid t(T),M(T)),\theta,\alpha,\mu)=P(N=q\geq M(T)\mid t(T),M(T)))$$
$$=\frac{We^{-\theta}\theta^{q}}{(q-M(T))!}\{\mu+t(T)T+(q-M(T))T\}^{-(\alpha+M(T))} \qquad (8.5.3(a))$$

W 为正规化常数。当 $M(T)=0$ 时，可得：

$$P(N=q\geq 0\mid T,0),\theta,\alpha,\mu)=P(N=q\geq 0\mid T,0))$$
$$=\frac{We^{-\theta}\theta^{q}}{q!}(\mu+qT)^{-\alpha} \qquad (8.5.3(b))$$

$M(T)$ 和 $t(T)$ 的样本分布，在 $k\geq 0$ 的条件下，可以写为：

$$P(M(T)=k\mid\theta,\alpha,\mu)\propto P(M(T)=k)$$
$$=\int_0^{\infty}\frac{\exp(-\theta(1-e^{-\lambda T}))}{k!}(\theta(1-e^{-\lambda T}))^{k}\frac{e^{-\mu\lambda}(\mu\lambda)^{\alpha-1}}{\Gamma(\alpha)}d\lambda \qquad (8.5.4)$$

$$P(t\mid M(T)=k,T,0)$$
$$=\int_0^{\infty}\frac{e^{-\mu\lambda}(\mu\lambda)^{\alpha-1}\mu}{\Gamma(\alpha)}\frac{(b\lambda)^{k}}{(k-1)!}Te^{-\lambda Tt}\sum_{j=0}^{k_0}(-1)^{j}\binom{k}{j}(tT-jT)^{k-1}d\lambda$$

其中，
$$k_0<t<k_0+1, k_0=0,1,\cdots,k-1, b=(1-e^{-\lambda T})^{-1} \qquad (8.5.5)$$

下面介绍在分析中用到的十分有用的决策树，见图 8.12。

图 8.12　单阶段软件测试的决策树

图 8.12 所示决策树中有两个决策点 D_1 和 D_2，以及两个随机结点 R_1 和 R_2。在结点 D_1 上，选定 T 的一个最佳值 T_0，而在结点 D_2 上只能采取一个行动，R：软件的投放。在随机结点 R_1 上，观察到足够的数据 $(t(T), M(T))$（如 $M(T) \geq 1$），或 $(T, 0)$（如 $M(T) = 0$）。在随机结点 R_2 上，各种不同的未知状态 $(N = M(T) + j, j = 0, 1, \cdots)$ 之一将被实现，而这些状态的每一种都会导出实用性为：

$$u[M(T) + j, R, (t(T), M(T)), T], \quad \text{当 } M(T) \geq 1,$$
$$u[M(T) + j, R, (T, 0), T], \quad \text{当 } M(T) = 0.$$

因为实用性表现得很象概率，所以我们必须要求上述的两个函数都是有界的。假设在 D_1 处，能对软件作 T 单位时间的测试，这将使从观察到的数据 $(t(T), M(T))$ 或 $(T, 0)$ 出发，得出 R_1 的结果。随后到达 D_2 结点，在这里唯一能采取的行动就是 R：投放软件。投放之后即到达 R_2，最终会遇到自然的各种未知状态中的一种：$N = M(T) + j, j = 0, 1, \ldots$，且产生出两个实用性函数中的一个：$u[M(T) + j, R, (\cdot, \cdot), T]$，其中 $(\cdot, \cdot)$ 表示 $(t(T), M(T))$ 或 $(T, 0)$。希望实用性函数具有下面的特性：随着 T 和 j 的增加，函数值下降。因为过长的测试时间意味着测试成本的增加，以及软件将冒更大的、被逐渐废弃的风险；且过大的 j 值也意味着遇到过程中出故障的机会会增加。然而，T 的值越大，其结果必然是 j 的值越小。因此，问题的关键就在于怎样选择一个合适的实用性函数。

为获取 T 的最佳值，从决策树的端点出发，按照期望的实用性函数的极大化原则，反向追溯到 D_1，可证明，T_0 是 T 的最佳值，其中 T 可导出下式：

$$\begin{aligned}
\sup_T \Bigg\{ & \sum_{k=1}^{\infty} \int_t \sum_{j=0}^{\infty} u[k + j, R, (t, k), T] \frac{W_1 e^{-\theta(t)} \theta(t)^{k+j}}{j!} \cdot (\mu + tT + jT)^{-(\alpha+k)} \\
& \cdot \int_0^{\infty} \frac{e^{-\mu\lambda} (\mu\lambda)^{\alpha-1} \mu}{\Gamma(\alpha)} \frac{(b\lambda)^k}{(k-1)!} T e^{-\lambda T t} \sum_{i=0}^{k_0} (-1)^i \binom{k}{i} (tT - iT)^{k-1} d\lambda \, dt \\
& \cdot \int_0^{\infty} \frac{\exp(-\theta(1 - e^{-\lambda T}))}{k!} (\theta(1 - e^{-\lambda T}))^k \frac{e^{-\mu\lambda} (\mu\lambda)^{\alpha-1}}{\Gamma(\alpha)} \mu \, d\lambda \\
& + \sum_{j=0}^{\infty} u[j, R, (T, 0), T] \frac{W_2 e^{-\theta} \theta^j}{j!} (\mu + jT)^{-\alpha} \\
& \cdot \int_0^{\infty} \exp(-\theta(1 - e^{-\lambda T})) \frac{e^{-\mu\lambda} (\mu\lambda)^{\alpha-1}}{\Gamma(\alpha)} \mu \, d\lambda \Bigg\} \qquad (8.5.6)
\end{aligned}$$

关于 t 的积分范围是 $k_0 < t < k_0 + 1, k_0 = 0, 1, 2, \ldots, k-1$。其中，

$$W_1 = \sum_{j=0}^{\infty} \frac{e^{-\theta}\theta^{+j}}{j!(\mu + tT + jT)^{\alpha+k}}^{-1},$$

$$W_2 = \sum_{j=0}^{\infty} \frac{e^{-\theta}\theta^{j}}{j!(\mu + jT)^{\alpha}}^{-1}.$$

实用性函数的选择

设在软件的运行期间(使用阶段)查出了 j 个错误,关于 j 有:

$$a_1 + a_2 e^{-a_3 j}, \quad (a_1, a_2, a_3 \text{ 为特定的常数})。$$

其中:a_1 表示具有大量(可能无限个)错误的程序的实用性,$a_1 + a_2$ 表示在程序中不含错误时的程序的实用性。不妨假设 a_1 是一个大的负数,$a_2 > 0$。这也就是假设了所有的错误引起的后果相同;这可能不尽合理。克服这一局限的方法之一就是将所有的错误分成不同的类,并对每一类设定不同的 a_2 和 a_3 的值。但是这样对于每一个类,都要求有一个精心制作的模型。设 C_1 表示在测试阶段查明一个错误的成本;$f(T)$ 为一非负的单调增函数,它表示测试成本加上直到 T 时尚未投放该软件的机会成本(the opportunity cost)。对于 $f(T)$ 的一个可能的选择是 $f(T) = dT^a$, $d > 0$, $a > 0$ 为特定的常数;或者,$f(T) = Z[Z + \ln T]$, $T < 1$ 或 $T \geq 1$, Z 是一个特定的常数。

在上述条件限制下,对软件测试了 T 个单位时间(CPU 时间),并改正了 $M(T) = k$ 个错误,然后投放,随之在运行期间又出现了 j 个错误,这时的整个实用函数为:

$$u[M(T) + j, R, (t(T), M(T)), T] = a^{-bT}[a_1 + a_2 e^{-a_3 j} - C_1 M(T) - f(T)]$$

其中,对于某个特定的常数 $b \geq 0$, e^{-bT} 是一个贬值系数。上面给出的函数,由以上讨论可知,是一个有界函数。有了 $u(T)$,对(8.5.6)式的计算就可进行了。

图 8.13、图 8.14 和图 8.15 分别示出几种具体情形下的总期望的实用性和 $u(T)$ 的行为方式。

实用性作为实际达到的可靠性的函数

上面提出实用性的动机,是以这样一个认识为基础的:实际达到的实用性是软件中剩余错误个数 j 的一个函数。这将忽略到 Λ 和 τ。Λ 表示残余错误个数的查出率(它也可能依次与软件的使用率有关);τ 表示软件的任务时间(mission time)。可以通过软件的可靠性函数来反映上述的特性。作为软件可靠性工程求解决策问题的一般的方法,将可靠性函数作为实用性函数的基础是十分有用的。

设实用性是 $R(\vartheta)$ 的函数,$R(\vartheta)$ 表示在 τ 时间内,一任务的可靠性函数。对特定的常数 $\alpha > 0$, $a_1 > a_0$, $l = 1, 2, \ldots$,在将实用性单独处理为 $R(\vartheta)$ 的一个函数时,有

$$u[R(\vartheta)] = \frac{a_1 - a_0}{1 - e^{-1/\alpha}} e^{-(R(\vartheta)-1)^{2l}/\alpha} + a_1 - \frac{a_1 - a_0}{1 - e^{-1/\alpha}} \qquad (8.5.7)$$

这样一个有界的实用性函数的一般形式示于图 8.16 中。

在图 8.12 中,在随机结点 R_2 处,给定数据 $(t(T), M(t))$ 或 $(T, 0)$,即测试时间 T 和任务时间 τ 我们将实际估计出一个可靠性的特定值,不妨写成 $(R(\vartheta)\mathbb{C}(\cdot, \cdot), T)$,于是它们将依次导出全实用性

图 8.13　总期望的实用性、u(T)和 T_0 的行为

$f(T)=T$　$(a=d=1)$, $b=0$, $a_1=-10$, $a_3=0.05$, $C_1=0.01$

$\alpha=2$, $\mu=1$, $a_2=110$　$\theta=5, 10, 15, 20, 25, 30$

取以上参数,对于不同的 θ值,u(T)与 T_0(使 u(T)达到最大值的点)的行为。

T_0 的值随 θ值的增加而下降。

图 8.14　总期望的实用性、u(T)和 T_0 的行为

$a=d=1$, $b=0$, $a_1=-10$, $a_3=0.05$, $C_1=0.01$, $\theta=10$,

$\mu=1$, $a_2=110$　$\alpha=2, 3, 4, 5$

取以上参数,对于不同的 α值,u(T)与 T_0 的行为。T_0 的值随 α值的增加而下降。

$$u[(R(\quad\quad,T),R,(\quad\quad,T]$$

图 8.15　总期望的实用性、u(T)和 T_0 的行为

$a=d=1, b=0, a_1=-10, a_3=0.05, C_1=0.01, \theta=10,$

$\mu=1, a_2=12, \alpha=1.5, 2, 3, 4$

取以上参数,对于不同的 α值,u(T)与 T_0 的行为。T_0 的值随 α值的增加而增加。

$= u[R(\vartheta\mathbb{C}(\cdot,\cdot),T] - C_1M(T) - f(T)$

在上面的讨论已涉及到,上面的式子事实上假定了 C_1 为在测试阶段,定位一个错误的成本,f(T)表示在 T 时刻,不投放软件的机会成本加上测试成本。

$(R(\vartheta\mathbb{C}(\cdot,\cdot),T)$可以通过剩余错误个数 j 和剩余错误的错误发生率 λ来指定。特别:

图 8.16　作为可靠性函数的实用性

$$(R(\vartheta\mathbb{C}(\cdot,\cdot),T) = \sum_{j=0}^{\infty}(R(\vartheta\mathbb{C}(\cdot,\cdot),M(T)+j,T)\cdot P(M(T)+j\mathbb{C}(\cdot,\cdot),T) \tag{8.5.8}$$

上面表达式中的最后一项可用(8.5.3(a))和(8.5.3(b))式来估计。因此有:

$$(R(\vartheta\mathbb{C}(\cdot,\cdot),M(T)+j,T)$$

$$\int_0^{\infty}(R(T)\mathbb{C}(\cdot,\cdot),M(T)+j,T,\lambda)P(\lambda\mathbb{C}(\cdot,\cdot),M(T)+j,T)d\lambda$$

因为 Λ被假设为具有标参 u 和形参 α的 Γ-分布,于是给定$(\cdot,\cdot),M(T)+j$ 和 T,Λ的后验分布也是一个 Γ-分布,具有标参

$$\mu\sum_{i=1}^{M(T)}(M(T)+j-i+1)\tilde{T}_i[\mu+T],$$

形参$(\alpha+M(T))$α当$(\cdot,\cdot)$是$(t(T),M(T))$或$(T,0)$时,且$(R(\vartheta\mathbb{C}(\cdot,\cdot),T,M(T)+j,\lambda$为 $e^{-\lambda\tau}$,则当 $M(T)\geq 1$ 时,有:

$$(R(\vartheta)\mid t(T), M(T)+j)=\int_0^{\infty} e^{-\lambda\tau} P(\lambda\mid t(T), M(T)+j, T)\,d\lambda$$

$$=\frac{\mu+\sum_{i=1}^{M(T)}(M(T)+j-i+1)\tilde{T}_i}{\mu+\sum_{i=1}^{M(T)}(M(T)+j-i+1)\tilde{T}_i+j\tau}^{M(T)+\alpha}$$

$$=\frac{\mu+j\sum_{i=1}^{M(T)}T_i+\sum_{i=1}^{M(T)}(M(T)-i+1)\tilde{T}_i}{\mu+j\sum_{i=1}^{M(T)}T_i+\sum_{i=1}^{M(T)}(M(T)-i+1)\tilde{T}_i+j\tau}^{M(T)+\alpha}$$

$$=\frac{\mu+Tt(T)+j\sum_{i=1}^{M(T)}T_i}{\mu+Tt(T)+j\sum_{i=1}^{M(T)}T_i+j\tau}^{M(T)+\alpha} \tag{8.5.9}$$

同样，当 $M(T)=0$ 时，有：

$$(R(\vartheta)\mid t(T), M(T)+j)=\frac{\mu+T}{\mu+T+j\tau}^{\alpha} \tag{8.5.10}$$

为了简化随后的计算，在(8.5.9)中的项 $j\sum_{i=1}^{M(T)}T_i$ 中的每个 T_i，都用它的先验期望来代替，以得出：

$$(R(\vartheta)\mid t(T), M(T)), M(T)+j)$$

$$\approx\frac{\mu+Tt(T)+\frac{\mu}{\alpha}\sum_{i=1}^{M(T)}(M(T)+j-i+1)^{-1}}{\mu+Tt(T)+\frac{\mu}{\alpha}\sum_{i=1}^{M(T)}(M(T)+j-i+1)^{-1}+j\tau}^{M(T)+\alpha} \tag{8.5.11}$$

将(8.5.10)和(8.5.11)式右边代入(8.5.8)中的相对应项，可得到 $u[R(\vartheta)\mid t(\cdot,\cdot), M(T)+j]$. 总之，$T_0$ 就是 T 的值，它在产生出(8.5.6)的结果时，有如下的变化：

① (8.5.6)式的 $u[k+j, R, (t,k), T]$ 由 $u[R(\vartheta)\mid t(t,k), k+j]-C_1k-f(T)$ 代替。

② (8.5.6)式的 $u[j, R, (T,0), T]$ 由 $u[R(\vartheta)\mid t(T,0), j]-f(T)$ 代替。

例　$u[R(\vartheta)]$ 由(8.5.7)式给定，$\tau=1$, $a_1=100$, $a_2=0$, $\alpha=0.1$, $l=1$，并设 $f(T)=T$, $C_1=0.1$：

① T_0 作为 θ 的函数($\mu=1$, $\alpha=2$)，

② T_0 作为 α 的函数($\mu=1$, $\theta=10$)。

结果列于表 8.8 中。

表 8.8　计算结果

					μ	α	θ
θ 的值	5	10	15	20	1	2	
T_0 的值	6	9	9	10			
α 的值	2	3	4	5	1		10
T_0 的值	9	5	4	3			

第九章　现 状 与 发 展

本章讨论软件可靠性研究的发展道路,以及目前研究的进展情况。

随着软件工程的深入发展,软件可靠性的研究也在不断向纵深发展。但是,工程实际中提出的问题也越来越多,越来越复杂。要回答并解决这些问题,需要持续不断的努力。

象硬件系统那样,在设计阶段,怎样进行软件可靠性设计? 目前软件工程方法多采用结构化的设计方法,对于组成软件系统的众多模块,怎么样来分配软件的可靠性? 随着科技水平的提高,生产活动中用到的电子设备不仅包括硬件,而且也包括软件。怎么样对既含硬件又含软件的复杂系统(目前有人将这种复杂系统称为 X-件)进行可靠性分析? 能对软件的随机测试过程直接建立软件可靠性模型吗?在进行软件测试的过程中,如果观察到的软件故障次数为零,又如何估计软件当前版本的故障概率? 计算机科学的不断进步,将人类的现代文明带进了一个全新的高科技时代。怎么样借用现代高科技的成果,以解决软件可靠性估测中的特殊问题? 专家系统的开发与应用、神经网络系统理论的发展,也可以用到软件可靠性的研究中来吗?

本章将对这一连串的问题,逐一地进行解答。

§ 9.1　软件可靠性研究的现状

软件可靠性理论的发展,到现在不过 20 年。在这期间,已发表了不少的软件可靠性估测模型。起初,模型的作者们的确花了不少的时间争论不休,各人都认为自己的模型是最好的,有着无可比拟的优点。有人用“模型战”来描述这一阶段的情况。但是,随着争论的深入发展,产生出了一系列客观的标准。而且由于论战的需要,模型作者们也使得在模型中所采用的统计方法越来越实际可用了。而且,许多经典的模型中的基本上一致的地方也被提取出来,并且得到了论证。

现在,论战早已过去,大家都能够平心静气地抛开对自己模型的偏爱,而客观地对这一领域的工作展开冷静而正常的研究和讨论。

下面几个方面的工作应该是将来要予以重点关注的:

要加强模型中使用的假设的验证工作。在模型中,用到了大量各种各样的假设,以描述软件的测试策略和过程、软件被使用的方式、软件中错误的分布和出错的过程,等等,虽然这些假设中的大多数有着长期的经验作为基础,但是,它们根本就没有被验证过。因此,对现有模型中使用的各种假设的验证和在这一基础上提出一些经过证实的共同假设,是

今后开发模型的一个关键。

要开发更多的软件测度。为了使软件可靠性模型更加优化,更加准确,软件可靠性理论应该将更多的知识包括进来,例如:关于软件开发过程的描述知识、关于设计审查报告、测试报告和问题报告的知识、关于软件系统安装定位以及怎样在那儿履行它的职能的描述知识、以及当前的位置和它的问题域的描述性知识,等等。一部分这样的知识如果能以一种更准确的方式进行量测,那么它们将只要用"是"或"不是"的方式予以表达即可。

要注重决策理论的研究。例如,关于综合性 Γ-分布的估计问题,对给定的有效数据,到底应该选用什么样的分布来进行估计呢?先验分布应该是系统所发生的故障时间服从的分布的一部分,这些分布可能是对数正态分布、指数分布、Weibull 分布,等等。怎样做到使用的先验分布能正确反映这些事实上的分布,从而保证估计的准确性呢?研究这些问题,无论对于理论上还是对于实际应用,都具有重要的意义。另外,对于下面一些事件,如:

- 预测的错误数过少/预测的错误数过多
- 投放时间过早/投放时间过晚
- 达到确定的故障率指标所需时间太长/太短

科学地设定成本函数,从而定义出有关这一类问题的决策的真实意义。如果能为软件可靠性模型定义出可利用性函数,则对于正确选择软件可靠性模型无疑会有所帮助,而且对于模型的比较、评价和改进都有好处。

关于贝叶斯估计的问题。我们的经验已经证明,故障时间的分布参数比起每一千行代码中的错误估计数,更趋于稳定。问题的症结可能在于:

- 故障时间分布参数是对软件使用的频度和方式的函数,比如可以定义出软件的执行频度、软件的执行路径出现的概率,等等,而它们至少可以部分地被看作人类共性的一个结果,并且因此使得分布参数相对地趋于稳定。而引入软件中的错误,则因人而异,不同的软件工程师,所犯错误是不相同的。
- 发生的软件错误个数,是问题领域、系统历史和寿命周期各阶段的函数,它受许多可变因素的影响,所以表现出更多的不稳定性。

基于上面的这些考虑,对于下述各种情况的测量就是很重要的:

- 在软件寿命的不同阶段所查出的每千行代码中的软件错误个数。
- 一连串安装——维护序列的净化效应。注意 $N_0(0)$ 是未知的。我们仅仅能观察到在每个位置 i 上出现的错误 $N_0(i)$,问题在于每个 $N_0(i)$ 都已知时(对于所有的 $i<j$),怎样估计 $N_0(j)$。
- 对与将要出现的错误数有关的问题域或位置之间的区别的量测。

即使对于这些问题的部分解,也将帮助我们选定一个关于 N_0 的先验分布,并因此极

大地提高对于 N_0 的估计精度。

软件可靠性研究,就目前的状况来看,还显得残缺不全。最明显的一点就是:用以测试软件可靠性模型、证明它们的估测精度、可应用性以及模型与模型之间的差别的数据库,至今还没有建立。这样造成的后果势必是理论与实际的脱离,模型的作者只顾理论上的问题,而忽视对模型实际效能的检验。为了建立一个完全的软件错误和可靠性数据库,下面的信息都是需要的:

- 对软件性能的描述所涉及到的机构、开发时间、结果、前后有关的软件项目;
- 简短但高水平的要求和规格说明书的摘要;
- 使用的程序设计语言、编译系统、操作系统、主要的工具以及运行环境;
- 软件的大小(资源、实体、注释和可执行的指令数)、开发方法、关键的进度标志和使用的人力、时间;
- 每个小时(周、月)在系统测试期间查出改正的错误个数,如果可能,在模块测试期间的数据也应给出;
- 为开发测试和模拟测试而进行的测试次数、水平,特别要给出出故障的时间和没有出故障的运行时间;
- 有关软件的工程开发过程中的一切数据。

为了建立软件可靠性数据库,要求做到:

- 最大限度地发挥软件寿命周期各阶段涉及人员的参与意识,主动收集数据;
- 尽可能方便、有效地使用各项目的定义,收集工具;
- 及时记录所有有关的信息并加强反馈;
- 管理和纪律、培训手段的配合;
- 大软件开发公司的骨干作用与小公司的配合协作,共同努力;
- 开展广泛的国际合作。

特别是当前对于分布式和并行计算的研究,极大地增加了软件系统的复杂性,如何为这样复杂的软件系统建立软件可靠性模型,已经为从事软件可靠性研究的学者们提出了一个重大的课题。

为了推进和协调软件可靠性的研究工作,目前国际上已正式成立了 IEEE/CS/TCSE 下面的 SRES(Software Reliability Engineering Subcommittee)。SRES 有权制定自己的方针、政策,每年召开一次年会,就软件可靠性工程的全部领域进行广泛的交流。另外,SRES 还可以制定自己的教育计划。

软件可靠性研究发展到今天,许多研究人员及学者发现,对它的发展过程进行一次反思,是大有好处的。因此,许多人又在进行总结与归纳的工作。整理软件可靠性使用的术语和名词、归纳软件可靠性研究中的基本概念、重新对它们下严格而科学的定义等的工作,正在有组织地、有条不紊地进行。也即是说,软件可靠性要成为一门具有坚实基础的学

科,首先对构筑基础的组成单元,必须进行一次全面的清理。

另一方面,人们在思考如何进行综合的工作,将现有模型的所有优秀特性都集中起来,发挥它们的优点,在软件可靠性工程的实用化方面,取得一些新的突破。因此,建造软件可靠性估测专家系统、综合应用现有软件可靠性模型等研究课题,也都被提了出来,并制订了计划,在一步一步地开展研究工作。相信在不久的将来,利用先进的人工智能技术,研制出来的实用化软件可靠性估测系统,将会发挥出它们特有的作用,将软件可靠性工程的研究与实用技术推向更新、更高的水平。

§ 9.2 软件可靠性的分配技术

可靠性分配要解决的问题就是在保证整个系统的可靠性达到规定目标的基础上,规定每个单个部件的可靠性应达到的可靠性指标。关于硬件的可靠性分配问题,已有许多成熟的技术可以应用,而关于软件的可靠性分配问题,则尚缺少成熟的技术。为此,F. Zahedi 和 N. Ashrafi 提出了一种进行软件可靠性分配的方法。她们的原则是:在最大限度地满足用户对系统的实用性的要求的基础之上,来确定每个单个部件应达到的可靠性指标。她们先设定一个实用性测度,以便综合考虑用户的要求,以及对每个单个部件的软件可靠性的优先选择权(preferences)。首先,确立一个系统划分为层次结构的方法,然后应用分析分层过程——AHP(Analytic Hierarchy Process),以推定需要的模型参数。模型的公式具有非线性规划的形式,它在模块和程序这两个层次上,考虑软件可靠性的各种技术和经济上的约束条件,使软件系统的实用性达到最大。求解该非线性规划问题,则可以确定在模块和程序级别上的软件可靠性的分配方案。

她们规定的一种统一的软件可靠性的层次划分方法,示于图 9.1 中。

图 9.1 软件可靠性的层次结构

首先按功能划分,然后进一步划分成程序,每个程序又划分成模块。再到下面,每个模

块又可以划分为若干子模块,但有一个约束条件:每个模块下面的子模块属于且仅属于该模块,而不能被其它的模块所调用。这一思想与面向对象的程序设计方法,以及可重用代码的思想都是一致的。

根据上述层次划分的方法,设系统被划分为 m 个互相独立的模块,记为 $M_1, M_2, \cdots, M_m$。模型对软件可靠性的分配,到模块级为止。下面的问题在于如何确定在 P 级的每个程序(以及在 M 级的每个模块)的相对重要性。显然,这里存在着两种不同的观点:(1)用户观点。用户只能看到 F 级,他不可能也不应要求他可以看到属于系统内部的 P 级与 M 级。因此,关于 P 级的每个程序(以及 M 级的每个模块)的相对重要性,不可能通过直接询问用户而得到。(2)程序开发者的观点。程序开发者只能看到软件内部的 P 级与 M 级,而对 F 级的了解则远远不如用户。因此,必须要找到一种方法,以确定 P 级的每个程序(以及 M 级的每个模块)的相对重要性。AHP 方法正好可以完成这一任务。AHP 方法分四步来解决这一问题:

① 设定划分层次的结构(这一步已在上面的讨论中完成)。

② 在每一层次由逐对比较的方法收集数据。

③ 使用本征值方法(或其它有效的计算方法),在每一层次上计算相对的权值。

④ 在得到分层结构除顶层外的各级低层上的元素的相对权值后的各个层次上,对相对权进行归并。

第一步已经完成,第二步要求以下述方式收集数据:在第二层,通过询问用户来逐对地比较这些功能,以便在总评定软件(U)时,可以判定功能的相对重要性。例如,假设软件有三个功能:F_1, F_2 和 F_3。从用户那儿可以引导出如下的输入矩阵:

$$A = \begin{array}{c} \\ F_1 \\ F_2 \\ F_3 \end{array} \begin{array}{c} \begin{array}{ccc} F_1 & F_2 & F_3 \end{array} \\ \begin{bmatrix} 1 & 2 & 4 \\ 1/2 & 1 & 3 \\ 1/4 & 1/3 & 1 \end{bmatrix} \end{array} \tag{9.2.1}$$

矩阵的第一行表示 F_1 相对于 F_1, F_2, F_3 的相对重要性:F_1 两倍重要于 F_2,四倍重要于 F_3。显见,主对角元上均为 1。但上面仍包含了不相容性:A_{23} 本应为 2,但实际上 $A_{23}=3$。这来自于人们的选择反应中的不确定性与误差,而 AHP 方法则顺应了这一不确定性。

本征值法在计算相对权时使用的输入矩阵为:

$$A\,W = \lambda_{max}\,W \tag{9.2.2}$$

其中,A 是通过对单个用户的逐对比较而获得的输入矩阵,而他们的优先选择方式正是我们力求了解的。本征值计算法就是要确定 W 和 λ_{max}。W 是一个层次关于它上一层的元素估计出的相对权,λ_{max} 为矩阵 A 估计出的最大本征值。有现成的软件工具求解式(9.2.2)。

如 A 是相容的,则已证 λ_{max} 等于矩阵 A 的维数。如(9.2.1)中的矩阵 A 是相容的(用 $A_{23}=2$ 代替 $A_{23}=3$),应用计算方法求解(9.2.2)式,将会得出:

$$W = \frac{4}{7}, \frac{2}{7}, \frac{1}{7},$$

$$\lambda_{max} = 3. \tag{9.2.3}$$

但输入矩阵为不相容的,所以得出的实际结果为

$$W = (0.558 \quad 0.320 \quad 0.122),$$
$$\lambda_{max} = 3.034,$$
$$\text{不相容率} = 0.017. \tag{9.2.4}$$

其本征值与 3 的距离即表示输入的不相容性的程度。

在层次结构的第二层,只需要逐对比较输入的矩阵,因为第一层上只有元素 U,我们只计算在完成功能 $F_1, F_2, \cdots, F_f$ 时的相对权。从第二级到第三级的联结,构成该软件的用户与软件工程师间的界面。在这里,软件工程师根据用户特定的要求,将软件划分为不同的程序,以完成规定的功能。

对于第三层,在实现功能 F_1 到 F_f 的过程中,程序可能担任不同的角色,而在实现不同的功能时,又可能需要不同的程序来完成。而它们在各自完成不同的功能时,所体现出来的重要性是不一样的。因此,我们需要产生出 f 个逐对比较的输入矩阵。

以功能 F_i 为例,我们有一个在完成它时的程序 $P_1, P_2, \cdots, P_p$ 的比较输入矩阵,$A^i(p \times p)$,写于右上角的 i,指明是为实现功能 F_i 而得到的关于程序 $P_1, P_2, \cdots, P_p$ 的输入矩阵。应用(9.2.2)式产生出一个 p 维的相对权向量 W^i。因为功能总数为 f,所以有 f 个 W^i ($i = 1, 2, \cdots, f$),称这些 W^i 为"局部相对权"。

怎样将这些局部权组合起来,在保证实现 U(第一层)时,得出程序级的各程序的相对权或重要性呢?即怎样组合第二层的局部相对权,达到在实现 U 时的"全局相对权"(第三层的)。在下面的讨论中,分别以 WF^i, WP^i, WM^i 表示第二层、第三层、第四层的局部相对权向量,以 AF^i, AP^i, AM^i 表示第二、三、四层的局部输入矩阵。右上角不标以字母 i,则表示是全局的。第三层的全局相对权可以计算如下:

$$WP = WF \begin{pmatrix} WP^1 \\ WP^2 \\ \vdots \\ WP^f \end{pmatrix} \tag{9.2.5}$$

它的展开形式为:

$$(wp_1 \quad wp_2 \quad \cdots \quad wp_p) = (wf_1 \quad wf_2 \quad \cdots \quad wf_f) \begin{pmatrix} wp_1^1 & wp_2^1 & \cdots & wp_p^1 \\ wp_1^2 & wp_2^2 & \cdots & wp_p^2 \\ & & \cdots & \\ wp_1^f & wp_2^f & \cdots & wp_p^f \end{pmatrix} \tag{9.2.6}$$

在第四层(模块层),可以重复上述过程,以得到模块 $M_1, M_2, \cdots, M_m$ 的全局相对权。对于每个程序,可得到一个 $m \times m$ 的方阵,而这样的输入矩阵一共有 p 个(因为有 p 个程序)。利用式(9.2.2),可获得 p 个 m 维的局部相对权向量,同样可获得第四层的全局相对权为:

$$WM = WP \begin{pmatrix} WM^1 \\ WM^2 \\ \vdots \\ WM^p \end{pmatrix} \tag{9.2.7}$$

其展开形式为:

$$(wm_1 \quad wm_2 \quad \cdots \quad wm_m) = (wp_1 \quad wp_2 \quad \cdots \quad wp_p) \begin{pmatrix} wm_1^1 & wm_2^1 & \cdots & wm_m^1 \\ wm_1^2 & wm_2^2 & \cdots & wm_m^2 \\ & & \cdots & \\ wm_1^p & wm_2^p & \cdots & wm_m^p \end{pmatrix} \tag{9.2.8}$$

上面的方法包含了很大的主观性因素,但 AHP 方法并不排除客观数据的使用,只要有客观的信息可资利用。

软件实用性测度

从用户的立场出发,一个软件的实用性一般可以认为是在这个软件中,用户能利用该软件,怎样可靠地实现各种不同的功能。换句话说,软件实用性是软件属性(功能)、以及这些属性的可靠性的函数:

$$U = h(F, R) \tag{9.2.9}$$

其中,U 表示实用性测度,F 和 R 分别表示软件功能的向量和这些功能的可靠性的向量。在这个一般的表达式中,h 可取任何的函数形式。当 h 为一线性函数时,我们可以定义实用性为:

$$U = \sum_{i=1}^{f} wf_i \times rf_i \tag{9.2.10}$$

其中,wf_i 表示功能 i 的全局相对权,rf_i 表示功能 i 的可靠性。(9.2.10)式表示用户可以将不同的权加于软件功能,因而软件功能可靠性也可以根据用户加于每个功能的相对重要性来加权。理论上,可以用(9.2.9)式或(9.2.10)式来计算软件的实用性。可以根据在功能测试中收集的故障数据来估计软件功能可靠性。类似地,我们还有:

$$U = \sum_{i=1}^{p} wp_i \times rp_i \tag{9.2.11}$$

其中,wp_i 表示程序 i 的全局相对权,rp_i 表示程序 i 的可靠性。

$$U = \sum_{i=1}^{m} wm_i \times rm_i \tag{9.2.12}$$

其中,wm_i 表示模块 i 的全局相对权,rm_i 表示模块 i 的可靠性。

软件可靠性分配模型

设软件中的全部模块都是独立的,程序 i 的可靠性可以写成组成它的所有模块的各模块可靠性之积:

$$rp_i = \prod_{m_j \in m_i} rm_j, \qquad (i = 1, 2, \cdots, p) \tag{9.2.13}$$

其中 m_i 表示用于构成程序 i 的所有模块的集合。将(9.2.13)式代入(9.2.11)式,有:

$$U = \sum_{i=1}^{p} wp_i \times \prod_{m_j \in m_i} rm_j \tag{9.2.14}$$

上式在给定系统与成本控制的条件之下,在判定模块可靠性的时候,构成实用性测度取极大值的方法。因此,软件可靠性分配模型有下面的结构:

$$\max_{rm_j} U = \sum_{i=1}^{p} wp_i \cdot \prod_{m_j \in m_i} rm_j,$$

应满足下列条件:

$$\begin{aligned} & rm_j \le u_j, \\ & rm_j \ge l_j, \\ & a_j + c_j \cdot rm_j \le \alpha \cdot v_j, \qquad (j = 1, 2, \cdots, m) \\ & \sum_{j=1}^{m} (a_j + c_j rm_j) \le C. \end{aligned} \tag{9.2.15}$$

在上式中,只有 rm_j 是未知变量,其它的全为模型的参数,下面讨论它们的估计。

出现在 $rm_j \le u_j (j = 1, 2, \cdots, m)$ 中的 u_j 是模块 j 的可靠性的可能的最高水平。理论上,u_j 可以取值 1,但在许多情况下,它是不现实的目标。称 u_j 为"可行的"可靠性水平。出现在 $rm_j \ge l_j (j = 1, 2, \cdots, m)$ 中的 l_j 是模块 j 的可靠性水平的最低允许限。称它为"最低可接受的"可靠性水平。它们构成系统控制。u_j 和 l_j 可以由软件工程师在软件开发的计划和设计阶段予以判定。如果在判定它们时无任何原则可循,也不妨设 $u_j = 1, l_j = 0$。

出现在 $a_j + c_j rm_j \le \alpha v_j (j = 1, 2, \cdots, m)$ 中的 a_j,表示对模块 j 加以可靠性为 rm_j 的约束之后的一般开支。c_j 为可变的成本开支,它是一种限界开支,后面还将讨论,它是具非线性形式的成本函数。α等于 1-(开发者的利润率),v_j 是模块 j 的售价。αv_j 为设计的模块 j 的完成成本,因此,它的成本一定不超过这一水平。

下面讨论模块售价应如何决定。设 V 表示用户愿意为购买一软件产品而付的价格,他为购买该软件能完成的功能而付钱。在用户对软件的评价过程中,我们已知模块的全局相对权,所以我们能导出模块 j 的一个隐式价格:

$$v_j = wm_j \cdot V, \quad (j = 1, 2, \cdots, m) \tag{9.2.16}$$

当软件被开发出来大批销售时,(9.2.16)式中的 V 将由 nV 代替,n 为销售的套数。因此,第三个约束条件说明分配给模块 j 的可靠性成本,不会超出其售价(可以为谋求利润率而作相应的调整)。在用户看来,一模块起着更重要作用时,它将得到更高的 wm_j 和更高的售价 v_j。假如其它条件均相同,而这一点就会导致对于那些在用户评价过程中有着更加重要性的模块,用户会愿意出更高的价格购买更高的模块可靠性。另一方面,如一模块更大、更复杂时,它的可变成本 c_j 也会更大,这导致在实现这一模块的配定的可靠性水平时,成本更高。因此,虽然 a_j 和 c_j 是模型参数,但可由软件工程师根据模块的大小、复杂度、以及其它影响模块成本的因素来估计它们的价值。

如成本控制取线性函数的形式(如(9.2.15)式),应考虑模块可靠性的财政上限,这是因为下面的事实。第三个控制条件能简化成:

$$rm_j \le \frac{\alpha v_j - a_j}{c_j}, \quad (j = 1, 2, \cdots, m) \tag{9.2.17}$$

当财政上限可能降至低于为某些模块而设的技术性下限时,上述这些情况就可能出现。此时,公式中描述的问题将没有可行的解。这时软件工程师可能考虑放弃这些模块的任何利

润,而取 α= 1。当这样做还不能解决技术与财政的矛盾时,则软件工程师可能采取或者改变技术规范,或者宁愿蒙受经济损失而开发出这些模块。

第四为整个项目的资源控制。C 为有效的财政资源,可靠性成本的总和必须小于它。经常有这种情况:软件开发的总经费远低于软件最后的总成本。这时的这一控制就成为问题了。

一旦模块可靠性分配完成,就可用(9.2.13)式进行程序可靠性分配。(9.2.15)式给出的是最低限度的控制条件,另外还可以增加如:人员控制、进度控制等条件。

(9.2.15)式有 3m+ 1 个线性控制条件和 1 个非线性的目标函数。这是一个非线性规划问题,要求一般形式下的局部最佳,即不能保证求出的解一定是最好的一组。但当目标函数是极大化问题中的一个凹函数(或者极小化问题中的一个凸函数),且控制函数为凸函数,问题就变为凹的(凸的)规划问题,它的局部最优值也就是全局最优值。

(9.2.15)式中的 3m+ 1 个线性控制条件,使它们满足凸性条件,目标函数包括 p 个离散的函数,形式为:

$$h_i = wp_i \prod_{m_j \in m_i} rm_j, \quad (i = 1, 2, \cdots, p) \tag{9.2.18}$$

因为 $wp_i \geq 0, rm_i \geq 0$ ($i = 1, 2, \cdots, p$),它们的乘积也大于(或等于)零。这时完全可以保证凹函数的和仍为凹函数的论断为真。如果我们能证明(9.2.18)式的每个 h_i 为凹函数,则目标函数必然满足凹性条件。对于本问题,这样做并非必要:因为 $rm_j \in [0, 1]$,所以,$h_i \geq 0$。于是在区间[0, 1]内,$h_i / rm_j \geq 0$ 也成立。不失一般性,不妨设:

$$h_i = wp_i rm_1^b rm_2^d \tag{9.2.19}$$

其中 b 和 d 为正整数,分别表示程序 i 调用模块 1 和模块 2 的次数。将 h_i 对 rm_1 求偏导,有:

$$\frac{h_i}{rm_1} = wp_i b rm_1^{b-1} rm_2^d \tag{9.2.20}$$

它在区间[0,1]中,关于 rm_1 和 rm_2 的值仍为正的。也即是说,在区间[0,1]内,h_i 是一个连续递增的函数,且在 1 处取得无约束极大值,在 0 处取得无约束极小值。又由于 h_i 是一个加性函数,当每个 h_i 在考虑的区间内是连续递增的,那么在该区间内,整个(9.2.15)式的目标函数也是连续递增的,且在 1 处取无约束的极大值,在 0 处取无约束的极小值。在这一区间内,它并没有任何局部极小值或局部极大值存在。于是,(9.2.15)式中的受限最优解不是一个局部极大解,因为在该区间内不存在关系到(9.2.15)式的点。因此,(9.2.15)式的解就是一个全局极大值。有许多算法求解这一问题,最常用的为梯度搜索法。例如 PC 机上的软件包 GAMS 就可以利用 Wolfe 的简化梯度法与拟牛顿算法,联合求解这一类问题。

(9.2.15)式的解可以得出程序级与模块级的可靠性分配指标 rm_j 和 rp_j。这些值要在编码前的计划和设计阶段得出。它们使得软件管理人员可以通过资源的分配来影响可靠性。这里说的资源分配是直接与程序和模块的可靠性相联系的。它提供了一个直接量测开发小组工作成效的尺度。与成本具有等效作用的可靠性,也提供了一个有效的评估工具。因此,软件可靠性分配对于软件开发的控制与管理,有着潜在的有力作用。

在编码之后,就要针对它们的软件可靠性指标进行测试。测试阶段同样可以利用可靠性分配技术。因为 rm_j 和 rp_j 都是开发指标,编码后的程序、模块都要使用软件可靠性模型加以估计。它们实际达到的可靠性与 rm_j 和 rp_j 的差距,可以帮助确定测试策略的修改与调整。

例　图 9.2 示出一结构化的分层软件系统。

图 9.2　层次结构的例子

为完成功能 1:

$$A^1=\begin{array}{c} \\ p_1 \\ p_2 \end{array}\begin{array}{c} \begin{array}{cc} p_1 & p_2 \end{array} \\ \begin{bmatrix} 1 & 4 \\ & 1 \end{bmatrix} \end{array},$$

$$wp^1=(wp_1^1 \quad wp_2^1)=(0.8 \quad 0.2),$$

不一致率 = 0　　(9.2.21)

为完成功能 2:

$$A^2=\begin{array}{c} \\ p_2 \\ p_3 \end{array}\begin{array}{c} \begin{array}{cc} p_2 & p_3 \end{array} \\ \begin{bmatrix} 1 & 2 \\ & 1 \end{bmatrix} \end{array},$$

$$WP^2=(wp_2^2 \quad wp_3^2)=(0.667 \quad 0.333), \qquad (9.2.22)$$

不一致率 = 0

为完成功能 3:

$$A^3=\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array}\begin{array}{c} \begin{array}{cccc} p_1 & p_2 & p_3 & p_4 \end{array} \\ \begin{bmatrix} 1 & 1 & 2 & 1 \\ & 1 & 1 & 2 \\ & & 1 & 1 \\ & & & 1 \end{bmatrix} \end{array},$$

$$WP^3=(wp_1^3 \quad wp_2^3 \quad wp_3^3 \quad wp_4^3)=(0.293 \quad 0.293 \quad 0.207 \quad 0.207),$$

不一致率 = 0.046. (9.2.23)

根据(9.2.21)、(9.2.22)、(9.2.23)和(9.2.4),并应用(9.2.6)式,得出程序的全局相对权为:

$$WP = (wp_1 \quad wp_2 \quad wp_3 \quad wp_4)$$

$$= (0.558 \quad 0.320 \quad 0.122)\begin{pmatrix} 0.8 & 0.2 & 0 & 0 \\ 0 & 0.667 & 0.333 & 0 \\ 0.293 & 0.293 & 0.207 & 0.207 \end{pmatrix}$$

$$= (0.48 \quad 0.36 \quad 0.13 \quad 0.03) \tag{9.2.24}$$

上面得出的即为用户评价软件时的程序的相对重要性。下面根据软件工程师的主观判断,得出完成每个任务时,模块间的相对重要性。

为完成程序1的任务:

$$A^1 = \begin{array}{c} \\ M_1 \\ M_2 \end{array}\begin{array}{c} \begin{array}{cc} M_1 & M_2 \end{array} \\ \begin{pmatrix} 1 & 3 \\ & 1 \end{pmatrix} \end{array},$$

$$WM^1 = (wm_1^1 \quad wm_2^1) = (0.75 \quad 0.25),$$

不一致率 = 0. (9.2.25)

为完成程序2的任务:

$$A^2 = \begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \end{array}\begin{array}{c} \begin{array}{ccc} M_1 & M_2 & M_3 \end{array} \\ \begin{pmatrix} 1 & 2 & 8 \\ & 1 & 5 \\ & & 1 \end{pmatrix} \end{array},$$

$$WM^2 = (wm_1^2 \quad wm_2^2 \quad wm_3^2) = (0.604 \quad 0.326 \quad 0.070),$$

不一致率 = 0.005. (9.2.26)

为完成程序3的任务:

$$A_3 = \begin{array}{c} \\ M_3 \end{array}\begin{array}{c} \begin{array}{c} M_3 \end{array} \\ \begin{pmatrix} 1 \end{pmatrix} \end{array},$$

$$WM^3 = (wm_3^3) = (1), \tag{9.2.27}$$

不一致率 = 0.

为完成程序4的任务:

$$A^4 = \begin{array}{c} \\ M_3 \\ M_4 \\ M_5 \\ M_6 \end{array}\begin{array}{c} \begin{array}{cccc} M_3 & M_4 & M_5 & M_6 \end{array} \\ \begin{pmatrix} 1 & 4 & 4 & 8 \\ & 1 & 1 & 2 \\ & & 1 & 2 \\ & & & 1 \end{pmatrix} \end{array},$$

$$WM^4 = (wm_3^4 \quad wm_4^4 \quad wm_5^4 \quad wm_6^4) = (0.615 \quad 0.154 \quad 0.154 \quad 0.077),$$

不一致率 = 0. (9.2.28)

应用(9.2.24)、(9.2.25)、(9.2.26)、(9.2.27)、(9.2.28)和(9.2.8)式,可得出模块的全局

相对权:

$$WM = (wm_1 \quad wm_2 \quad wm_3 \quad wm_4 \quad wm_5 \quad wm_6)$$

$$= (0.48 \quad 0.36 \quad 0.13 \quad 0.03) \begin{pmatrix} 0.750 & 0.250 & 0 & 0 & 0 & 0 \\ 0.604 & 0.667 & 0.070 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0.615 & 0.154 & 0.154 & 0.077 \end{pmatrix}$$

$$= (0.577 \quad 0.237 \quad 0.174 \quad 0.005 \quad 0.005 \quad 0.002) \tag{9.2.29}$$

假设软件的售价(或价值 V)为 $1,000,000,应用(9.2.16)式和(9.2.29)中给出的值,可计算出:

$$(v_1 \quad v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6) = (577 \quad 237 \quad 174 \quad 5 \quad 5 \quad 2) \tag{9.2.30}$$

其中的单位为 $1,000。

软件工程师小组决定模块可靠性的下、上限各为 0.5 和 1,利润率为 50%,于是,$\alpha = 1 - 0.5 = 0.5$,估计该软件的开发投资应为 $250,000,估计的成本函数是根据每个模块的大小和复杂度作出的。

现在已为软件可靠性的分配准备好了必要的数据。应用(9.2.15)式,有

$$\begin{aligned} \max_{rm_j} U = {} & 0.48(rm_1 \quad rm_2) + 0.36(rm_1 \quad rm_2^2 \quad rm_3) \\ & + 0.13(rm_3^4) + 0.03(rm_3^2 \quad rm_4^3 \quad rm_5 \quad rm_6), \\ & rm_j \leq 1, \\ & rm_j \geq 0.5, \\ & 50 + 25rm_1 \leq 0.5(577), \\ & 20 + 99rm_2 \leq 0.5(237), \\ & 10 + 18rm_3 \leq 0.5(174), \\ & 1 + 5rm_4 \leq 0.5(5), \\ & 1 + 5rm_5 \leq 0.5(5), \\ & 1 + 1rm_6 \leq 0.5(2), \\ & 83 + 25rm_1 + 99rm_2 + 18rm_3 + 5rm_4 + 5rm_5 + rm_6 \leq 250 \end{aligned} \tag{9.2.31}$$

上面约束条件中的(3)—(8)的其它数据为模块的成本函数。仔细观察一下(9.2.31)中的约束条件,可以发现该问题没有一个可行的解,因为对于模块 4,5,6 的财政约束条件 $rm_4 \leq 0.3$, $rm_5 \leq 0.3$, $rm_6 \leq 0$,这是不可能的。主要原因在于这些模块的重要性水平偏低,从而使得在决定它们的隐式售价时偏低得太多。为解决这一非实用性问题,软件工程师小组决定接受这样的处理方法:这三个模块无利润开发,因此,可以将这三个约束条件修正为:

$$\begin{aligned} & 1 + 5rm_4 \leq 5, \\ & 1 + 5rm_5 \leq 5, \\ & 1 + rm_6 \leq 2. \end{aligned} \tag{9.2.32}$$

最后,应用 PC 程序包中的 GAMS,完成可靠性分配如下:

$$(rm_1 \quad rm_2 \quad rm_3 \quad rm_4 \quad rm_5 \quad rm_6) = (1 \quad 0.995 \quad 1 \quad 0.8 \quad 0.8 \quad 1),$$

$$U = 0.976.$$

使用(9.2.13)式,对于 4 个程序的可靠性分配如下:

$$(pr_1 \quad pr_2 \quad pr_3 \quad pr_4) = (0.995 \quad 0.99 \quad 1 \quad 0.41).$$

§ 9.3 X-件(软件与硬件的结合体)的可靠性分析

现代化的电子开关系统,既有硬件,也有软件。如何评估它们的可靠性,问题当然更复杂、更难。目前,将这种既含硬件、又含软件的复杂系统,暂简称为 X-件系统(X-ware system)。J.C.Laprie 和 K.Kanoun 等人在这方面进行了许多研究,下面分几个方面来介绍他们的工作。

1. X-件系统的故障行为

A. 基本系统

一个最简单的功能模型是:从它的输入域 I 映射到它的输出空间 O。系统的一次执行过程即在于选择一系列的输入点。如果认为状态变量是 I 或 O 的部分,则 I 中的每个元素都被映射到 O 中的唯一一个元素上。

一次系统故障的起因包括:①由事先存在于系统内的内部错误的被激活所引起的。内部错误包括:由一硬部件的物理故障引起,通常称这一类的错误为物理错误;或影响到软件或硬件的设计错误。②外部错误的出现,一般是由于物理环境或人类的失误引起。

于是可以将输入空间划分为两部分:I_{f_i},有错误的输入子空间;I_{a_f},激活内部错误的输入子空间。系统的故障域 $I_F = I_{f_i} \cup I_{a_f}$。当输入轨迹与 I_F 相交叉,就会出错,由此引发故障。输入点的每一次选择,都存在使系统失效的可能性。设 p 表示由于输入点的选择而引起系统失效的概率,则:

$p = P\{$对于选择的一个输入点,系统发生故障 $\big|$ 对于以前选择的输入点,系统不发生故障$\}$

用 $R_d(k)$ 表示在包含 k 次输入点选择的执行期间,系统不发生故障的概率,则

$$R_d(k) = (1 - p)^k \tag{9.3.1}$$

$R_d(k)$ 称为离散时间系统可靠性。设 t_e 为以一次输入选择的输入点作为输入的执行期,t 表示自执行开始后逝去的时间,则

$$t = kt_e \tag{9.3.2}$$

令:

$$\lambda = \lim_{t_e \to 0} \frac{p}{t_e} \tag{9.3.3}$$

于是有:

$$R(t) = \lim_{t_e \to 0} R_d(k) = \exp(-\lambda) \tag{9.3.4}$$

$R(t)$ 称为连续时间系统可靠性,λ 即故障率。

定义:$p(j) = P\{$对于第 j 次选择的输入点,系统发生故障 $\big|$ 对于以前选择的输入点,系统不发生故障$\}$,

$t_e(j)$= 以第 j 次输入选择的输入点作为输入的执行期,

则等式(9.3.1)～(9.3.4)变为:

$$R_d(k) = \prod_{j=1}^{k}[1 - p(j)],$$

$$t = \sum_{j=1}^{k} t_e(j),$$

$$x(j) = \lim_{\forall j[t_e(j)\to 0]} \frac{p(j)}{t_e(j)}.$$

于是有:

$$R_d(k) = \prod_{j=1}^{k}\{1 - \lambda(j)t_e(j) + o[t_e(j)]\} \tag{9.3.5}$$

$$R(t) = \lim_{\forall j[t_e(j)\to 0]} R_d(k) = \exp\left[-\int_0^t \lambda(\tau)\,d\tau\right] \tag{9.3.6}$$

设 j_0 表示选择的输入点数,它使得:

$$p(j) = \begin{cases} 0, & j < j_0 \\ p, & j \geq j_0 \end{cases}$$

$u(j_0)$ 表示一概率,则定义:

$$u(j_0) = P\{p(j) = 0 \quad (j < j_0),\ p(j) = p\ (j \geq j_0)\}.$$

离散时间系统可靠性可写成:

$$R(t) = \lim_{t_e\to 0} R_d(k) = \exp(-\lambda t).$$

上面的公式,对于无论是内部或外部错误,物理的或设计引入的错误,都可应用。对于软件而言,随机性至少来自输入空间中将激活错误的那些输入轨迹。特别,现在已知大多数软件错误是在验证阶段以后的运行阶段才表现出来的,它们是"软"的错误,而且要重现它们的激活条件是极为困难的。因此,对它们的诊断和排除的困难,更增加了随机性。

就硬件而言,使用试验数据估计的电子部件的故障率,实际上是随时间的增加而下降的,但下降的速度很慢。对于软件而言,也存在类似现象:对于给定的固定操作条件,软件有着常数故障率;系统负载对故障率的影响很大。

从概率的观点出发,在离散时间(或在连续时间)的执行期间的故障概率,可以认为是一个随机变量,则系统可靠性产生于以下两个分布的混合:

① 在给定运行条件的情况下,对于给定的执行期间常数故障概率的条件下,无故障执行次数的分布(或关于给定的常数故障率的故障时间的分布);

② 在执行期间故障概率的分布(或故障率的分布)。

设 $g_d(p)$ 表示在执行期间的故障概率分布的概率密度函数,对于 p 的实现 p_i 取 G 个值($i = 1, \cdots, G$);$g_d(\lambda)$ 表示在执行期间的故障率分布的概率密度函数,对于 λ 的实现 λ_i 取 G 个值($i = 1, \cdots, G$),则离散时间系统可靠性的表示为:

$$R_d(k) = \sum_{i=1}^{G}(1 - p_i)^k g_d(p_i),$$

$$t = k\sum_{i=1}^{G} t_{ei} g_d(p_i),$$

$$\lambda = \lim_{t_{ei}\to 0} \frac{p_i}{t_{ei}} \quad (i = 1, \cdots, G).$$

其中, t_{ei} 和 λ 分别表示关于在运行条件 $i(i = 1, \cdots, G)$ 下, 实际的执行的执行时间和故障率, 于是有:

$$R(t) = \lim_{i[t_{ei}\to 0]} R_d(k) = \sum_{i=1}^{G} g_d(\lambda_i)\exp(-\lambda_i t),$$

它为混合型指数分布。

当 p 为一具有密度函数 $g_e(p)$ 的连续随机变量, λ 为一具有密度函数 $g_c(\lambda)$ 的连续随机变量时, 有:

$$R_d(k) = \int_0^1 (1-p)^k g_c(p)\,dp,$$

$$R(t) = \int_0^\infty \exp(-\lambda t) g_c(\lambda)\,d\lambda$$

无论对于分布 $g_d(\cdot)$ 或 $g_c(\cdot)$, 由分布混合的情形可知, 系统故障率并不随时间的增加而增加。例如, 具有广泛一般性的连续分布 Γ-分布, λ 就可作为一连续随机变量, 有:

$$g_c(\lambda) = \frac{b^a \lambda^{a-1}\exp(-b\lambda)}{\Gamma(a)}, \text{ 且 } R(t) = \left(\frac{b}{t+b}\right)^a \text{ 是 Parato 分布。}$$

下面考虑独立执行的系统的 M 个实例。关于物理错误,“独立”意指对于给定的相同的输入模式序列, 硬件的一系列集合以并行方式运行; 关于软件, 则指一系列的小系统对给定的独立的输入序列的执行。设 $M(k)$ 表示 k 次执行以后, 无故障的实例数; $M(t)$ 表示自试验开始到逝去一段时间 t 以后, 无故障的实例数。$M(0) = M$, 在第 j 次执行时于 τ 时刻出现故障($\tau \in [0, t]$), 则不再继续执行($j = 1, \cdots, k$)。各种不同实例的执行的独立性允许将这些执行看作伯努利试验, 于是离散时间可靠性为:

$$R_d(k) = \frac{E[M(k)]}{M(0)},$$

连续时间可靠性为:

$$R(t) = \frac{E[M(t)]}{M(0)},$$

$R_d(k), R(t)$ 的统计估计量就是:

$$\hat{R}_d(k) = \frac{E[M(k)]}{M(0)},$$

$$\hat{R}(t) = \frac{E[M(t)]}{M(0)}.$$

B. 由部件组成的系统

系统为结构化的树型结构, 递归关系终止于叶子——基本系统, 也即原子系统, 原子系统可以看成对它的内部结构不再可分。每一结点表示一个部件, 许多数目不等的结点组成一层, 而它们是根据关系“由…所组成”组织起来的。许多层又组成一个层次结构。图 9.3 给出一个非常简单的例子, 其中三个部件 S_1, S_2 和 S_3 可以分别看作应用软件、执行软

件和硬件。

图 9.3　系统的部件

(a) 对应于关系“由…组成”的系统模型

(b) 对应于关系“与…发生相互作用”的系统模型

关于服务的概念,系统和它的用户各自所起的作用都是固定的:系统提供服务,用户消费服务。因此,在系统与它的用户之间,存在一种自然的层次关系:用户使用系统的服务。

对于在一分解树的一给定层次上的部件的集合,关系“使用谁的服务”是“相互作用”关系的一种特定的形式,准确定义了部件的一个特殊类:当且仅当一层次上的全部部件可以通过关系“使用服务”建立层次结构时,这些部件可称为“层”(layers)。换一种完全等价的方式讲,一给定层次上的部件叫做层。如果:(a) 对于这一关系,这一层次上的任何两个部件都担当固定的角色,如:供方、买方,或平等的双方;(b) 其关系图是一个单个的分支树结构。反过来说,如果它们各自的供方与买方的位置可以互换,或者它们的相互作用并不为该关系所约束,那么它们就是部件。图 9.4 示出系统的“层”。

图 9.4　系统的层。对应于关系“使用由…提供的服务”的系统模型

第三类关系就是“被谁解释”。两个层之间的解释界面或层的集合的特性,由实体以及为控制这些实体的操作的“提供”来描述。于是一系统可看作解释器的层次结构,其中可将给定的一解释器看作一个“抽象的”实体的“具体的”解释。在下面的讨论中,我们将这一具体的解释本身,也看作是抽象的实体。

再看图 9.3 中的例子。硬件对应用软件和执行软件作出解释,而执行软件可以通过:(a)提供“监督调用”指令,或(b)借助于“特权”指令的保护,来作为硬件解释器的外延。据这一关系建立的系统模型,示于图 9.5 中。

2．单解释器系统的行为

设系统由 C 个部件(每个部件的故障率为 λ_i, i= 1, 2, ⋯, C)组成。关于执行过权的系统的行为模型, 用具有下列参数的马尔可夫链描述:

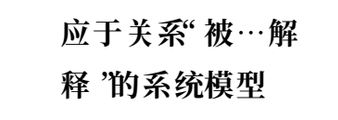


图 9.5　系统解释器。对应于关系“被…解释”的系统模型

S: 链状态数, 状态由在运行之下的部件定义。

1/ γ: 在状态 j (j = 1, 2, ⋯, S)中的平均逗留时间。

q_{jk}= P{系统由状态 j 到状态 k 的转换©一个或一系列部件的执行的开始或终止}, 且 $\sum_{k=1}^{S} q_{jk} = 1$。

一系统故障是由系统的部件的任一故障引起的, 在状态 j 中的系统故障率 ξ 就是在状态 j 中处于执行之下的部件的故障率之和:

$$\xi = \sum_{i=1}^{C} \delta_{ij} \lambda_i, \quad j = 1, 2, \cdots, S \tag{9.3.7}$$

其中, δ_{ij}, 当状态 j 中的部件 i 处于执行之下时, 其值为 1; 否则为零。

系统故障行为可以用有 S+ 1 个状态的马尔可夫链来建模, 其中系统在前 S 个状态中不发生故障(即处于执行之下的部件不出现故障); 状态 S+ 1 为故障状态, 即吸收状态。设 A= [a_{jk}], j= 1, 2, ⋯, S, k= 1, 2, ⋯, S, 为与无故障状态相关联的转移矩阵, 它有下列两个特点:

- 对角元 a_{jj} 等于- (r_j + ξ);
- 非对角元 a_{jk}, j ≠ k, 等于(q_{jk} · r_j)。

可以将 A 看作两个矩阵 A′, A″之和: A= A′+ A″

- A′的对角元等于- γ, 非对角元等于 q_{jk} · γ,
- A″的对角元等于- ξ, 非对角元等于零。

因此系统行为可以看作下列两个过程的重迭的结果:

- 参数 γ 和 q_{jk}(转移矩阵 A′)的执行过程;
- 由故障率 ξ(转移矩阵 A″)控制的故障过程。

一个自然的假设就是: 关于由执行过程向故障过程转换的故障率是很小的; 否则, 不满足这一假设的系统将没有什么实际的意义。这可以表示为:

$$\gamma \gg \xi \tag{9.3.8}$$

根据定义, 系统故障率 $\lambda(t)$ 为:

$$\lambda(t) = \lim_{dt \to 0} \frac{1}{dt} P\{\text{在区间}[t, t+dt]\text{内发生故障}\mid\text{在起始时刻与 } t \text{ 之间不发生故障}\}$$

设 $P_j(t)$ 表示系统处于状态 j 的概率, 则:

$$\lambda(t) = \sum_{j=1}^{S} \xi P_j(t) \Big/ \sum_{j=1}^{S} P_j(t) \qquad (9.3.9)$$

由(9.3.8)式可知,在发生故障前,执行过程趋向于平衡。向量 $\alpha = [\alpha_j]$ 表示平衡概率,它为: $\alpha \cdot A = 0$ 的解,其中有 $\sum_{j=1}^{S} \alpha_j = 1$。因此, $P_j(t)$ 在故障发生之前趋向于 α_j,且(9.3.9)式变成:

$$\lambda = \sum_{j=1}^{S} \alpha_j \xi \qquad (9.3.10)$$

$$= \sum_{j=1}^{S} \alpha_j \sum_{i=1}^{C} \delta_{ij} \lambda_i = \sum_{i=1}^{C} \lambda_i \sum_{j=1}^{S} \delta_{ij} \alpha_j \qquad (9.3.11)$$

设 $\pi_i = \sum_{j=1}^{S} \delta_{ij} \alpha_j$, (9.3.10)式可变成:

$$\lambda = \sum_{i=1}^{C} \pi_i \lambda_i \qquad (9.3.12)$$

它有一个简单的物理解释:

- α_j 表示在状态 j 不发生故障时所费时间的平均比例,因此, π_i 为当部件 i 处于执行之下,不发生故障的时间的平均比例,且满足 $0 \le \sum_{i=1}^{C} \pi_i \le C$.
- λ_i 是假定连续执行时部件 i 的故障率。

因此,可把 $\pi_i \lambda_i$ 看作部件 i 的等价故障率。

关于(9.3.12)式,进一步讨论如下。先分析硬件系统的情况:所有的部件都持续地发挥作用,因此要求所有的 π_i 都等于 1,于是有: $\lambda = \sum_{i=1}^{C} \lambda_i$。

再来分析软件系统。中心问题在于怎样估计部件的故障率。有两种基本的、但截然相反的作法:

- 利用无故障重复运行的结果,经过统计检验来估计部件的故障率;
- 利用故障数据,对每个软件部件使用一软件可靠性增长模型,以估计它们的故障率。

重要的在于强调在上面两种方法中的数据的代表性,要作到这一点的一个条件就是:数据应是在典型的环境中收集的,即环境应将相互作用的部件——实际的或模拟的——与其它的系统部件联系起来。如果这一条件得不到满足,就会在界面故障率(它表述在与其它部件相互作用期内的故障行为)与内部部件故障率之间产生差距。LittleWood 将一个部件故障率表示为下面的形式:

$$\lambda_i = \zeta_i + \gamma_i \sum_j q_{ij} \beta_j,$$

其中，ζ为内部部件故障率，ρ_{ij}为界面故障概率。由此产生 C^2 阶而不是 C 阶的估计复杂性。

对于不同的环境，怎样来计算呢？如果这一问题是指估计使用的一堆软件系统的某一软件系统的可靠性，那么，可以采用上面讨论的对于基本系统的处理办法，将 π 都作为随机变量来处理。如果是指在一给定的环境中已知其可靠性，怎样来估计另一环境中它的可靠性？设我们考虑的是序贯软件系统。定义表示执行过程的参数如下：

- $1/\gamma_i$：部件 i 的平均执行时间，$i = 1, 2, \cdots, C$.
- $q_{ij} = P\{$部件 j 开始执行 | 部件 i 结束执行$\}$，$i = 1, 2, \cdots, C, j = 1, 2, \cdots, C$，且 $\sum_{j=1}^{C} q_{ij} = 1$。

于是，有$(C+1)$个状态的马尔可夫链，状态 i 由部件 i 的执行来定义，且全部 π 缩减为 α（在考虑序贯软件的情况下，$\delta_i = 1$，其它的全为零）。于是有：$\lambda_i = p_i\gamma_i$，$i = 1, 2, \cdots, C$，p_i 是在部件 i 执行时的故障概率，因此，

$$\lambda = \sum_{i=1}^{C} \pi_i\gamma_i p_i = \sum_{i=1}^{C} \eta_i p_i \qquad (9.3.13)$$

其中，$\eta_i = \pi_i\gamma_i$，为在平衡时状态 i 的访问率。η_i 有着简单的物理解释：$1/\eta_i$ 为状态 i 的平均递归时间，即在状态 i 的两次无故障执行的平均时间段。式(9.3.13)能区别究竟是连续的时间执行过程，还是以执行为条件的离散的时间故障过程。如所有的 p_i 对于考虑的软件是内在的，且执行过程是独立的，那么对于一给定的环境，由已知的关于该环境的 η_i 和已知的 p_i，推导出关于该环境的软件故障率是可能的。在实践中验证这一假设的条件是：有可能找到将一软件分解为部件的适当方法。要知道，对于一软件，部件的概念是十分随意的，并且如果将一软件分解成的部件越多，则每一部件的状态空间就越小，对该部件提供的输入空间的满意的覆盖范围的似然率就越高。这一方法的限制在于划分的部件数越多，估计 η_i 的难度就越大。请注意，时间区间的划分（以及近似的可分解性）对于寻找适当的分解方法，能提供有力的帮助。

3．多解释器系统的行为

如果一系统被作为由解释器构成的层次结构，最高层解释器（它直接解释来自用户的要求）的与输入点选择有关的执行，就由它下面的直接后承所支持，依次类推，就可一直追溯到最底层的解释器。

设系统由 I 个解释器组成。第一个解释器处于层次结构的顶端，第 I 个解释器处于最底层。每个解释器都可能出错，且都可归因于错误的输入。与它的直接前继的解释器的与输入点选择有关的计算期间的任一解释器的故障，将导致其后的解释器发生故障。由此产生的故障传播将导致顶端的解释器发生故障，致使整个系统发生故障。据传统的可靠性理论，它们组成一个串联系统，于是可直接得出系统故障率等于每个解释器的故障率之和。如果 $\lambda_i(t)$，$i = 1, 2, \cdots, I$，表示解释器 i 的故障率，则：

$$\lambda(t) = \sum_{i=1}^{I} \lambda_i(t). \tag{9.3.14}$$

下面定义:

- $V_i(j_{i-1})$, $i = 1, 2, \cdots, I$ 为解释器 i 执行的输入点的个数, 它们对应于解释器 $i-1$ 的第 (j_{i-1}) 次选择, $V_1(j_0) = k$, k 为由用户为选定的部件运行所作的输入点选择的次数;
- $P_i(j_i) = P\{$解释器 i 在执行第 j_i 次选择输入点时发生故障©对于以前的所有输入点选择, 均不发生故障$\}$, $j_i = 1, 2, \cdots, V_i(j_{i-1})$, $i = 1, 2, \cdots, I$;
- $t_e(j_i)$, $i = 1, 2, \cdots, I$, 是解释器 i 关系到第 j_i 次输入点选择的执行时间,
  $j_i = 1, 2, \cdots, V_i(j_{i-1})$, $i = 1, 2, \cdots, I$.

于是式(9.3.1)～(9.3.15)变为:

$$R_d(k) = \prod_{j_1=1}^{k} [1 - p_1(j_1)] ¡¤ \prod_{j_2=1}^{V_2(j_1)} [1 - p_2(j_2)] \cdots \prod_{j_I=1}^{V_I(j_{I-1})} [1 - P_I(j_I)],$$

$$t = \sum_{j_1=1}^{k} t_e(j_1) = \sum_{j_1=1}^{k} \sum_{j_2=1}^{V_2(j_1)} t_e(j_2) = \sum_{j_1=1}^{k} \sum_{j_2=1}^{V_2(j_1)} \cdots \sum_{j_I=1}^{V_I(j_{I-1})} t_e(j_I),$$

$$\lambda = \lim_{"j_i[t_e(j_i)\to 0]} \frac{p_i(j_i)}{t_e(j_i)},$$

$$R_d(k) = \prod_{j_1=1}^{k} \{1 - \lambda(j_1)t_e(j_1) + o[t_e(j_1)]\} \cdots \prod_{j_I=1}^{V_I(j_{I-1})} \{1 - \lambda(j_I)t_e(j_I) + o[t_e(j_I)]\},$$

$$R(t) = \lim_{"j"j_i[t_e(j_i)\to 0]} R_d(k) = \exp - \sum_{i=1}^{I} \int_0^t \lambda(\tau)\mathrm{d}\tau = \exp - \int_0^t \sum_{i=1}^{I} \lambda(\tau)\mathrm{d}\tau.$$

于是式(9.3.14)得证。

下面设每个解释器由 C_i 个部件组成, $i = 1, 2, \cdots, I$。在执行中, 解释器 i 的某部件将用到解释器一个或多个部件提供的服务。于是可以定义如图 9.6 所示的解释器 i 的部件, 使用由解释器 $i+1$ 的部件提供的服务利用树。

图 9.6 解释器的部件间的利用树

对于每对毗连的解释器对, 都可能联系到一服务利用矩阵 $U_{i,i+1} = [U_{jk}]$, $j = 1, 2, \cdots, C_i$, $k = 1, 2, \cdots, C_{i+1}$。$U_{i,i+1}$ 是一个连通矩阵, 它的项 U_{jk} 为:

$U_{jk} =$ 1, 如解释器 i 的部件 j 在执行期间, 利用解释器 $j+1$ 的 k 部件提供的服务;
0, 否则。

定义故障率向量如下:

$\Lambda_i = [\lambda_j]$, $i = 1, 2, \cdots, I$, $j = 1, 2, \cdots, C_i$, λ_j 是解释器 i 的部件 j 的故障率。

$\Omega = [\omega_{i,j}]$，$i = 1, 2, \cdots, I$，$j = 1, 2, \cdots, C_i$，$\omega_{i,j}$是解释器 i 的部件 j 的总故障率，即执行所需解释器 $i+1, \cdots, I$ 的部件的故障率。它满足：

$$\Omega_i = \Lambda_i + U_{i,i+1}\Omega_{i+1}, \quad i = 1, 2, \cdots, I-1, \quad \Omega_I = \Lambda_I.$$

于是有：

$$\Omega_1 = \sum_{k=1}^{I} V_k \Lambda_k.$$

V_k 为从顶端解释器至解释器 k 的可达性矩阵：

① V_1 是$(C_1 \times C_1)$维的恒等矩阵；

② $V_k = U_{1,2} \cdot U_{2,3} \cdots \cdot U_{k-1,k}$，$k = 2, 3 \cdots, I$。符号 · 表示矩阵的布尔乘：一给定的部件的故障率只能乘一次。

将式(9.3.11)应用于层次结构的顶端解释器，就得到系统故障率：

$$\lambda = \sum_{j=1}^{C_1} \pi_{1,j}\omega_{1,j}$$

其中，$\pi_{1,j}$是顶端解释器的部件 j 的执行所占时间比例，当部件闲置时，用 $w_{1,j} = 0$ 表示。

现在考虑实际中一个很重要的特殊情况：系统由两个解释器组成，一个软件，一个硬件，且软件的部件以序贯方式执行，硬件则在执行期间它所有的部件要同时并行操作。设系统的运行条件不变。在下面的讨论中，分别用下标 S 和 H 代表软件和硬件。应用上面的关系，可以导出：

$$w_{S,j} = \lambda_{S,j} + \sum_{k=1}^{C_H} \lambda_{H,k},$$

$$\lambda = \sum_{j=1}^{C_S} \pi_{S,j} \cdot \omega_{S,j} = \sum_{j=1}^{C_S} \pi_{S,j} \cdot \lambda_{S,j} + \sum_{k=1}^{C_H} \lambda_{H,k} \qquad (9.3.15)$$

2．服务可恢复的 X-件系统的故障行为

可靠性的增长率(或故障强度的下降率)与排错和维护策略密切相关，这些策略可以是：每次故障之后的系统修改；在一个规定的故障次数之后的系统修改；以及保护性的维护措施(即对系统在未观察到故障的情况下的系统修改)。在下面的讨论中，称两次系统修改之间的系统状况为一个版本，如：自从 $j-1$ 次系统修改以后，在系统发生 a_j 次故障之后的第 j 次系统修改，可以称为版本 j 发生了 a_j 次故障。设：

- $X_{j,i}$表示版本 j 在第 $i-1$ 次故障之后的服务恢复和第 i 次故障之间的时间；
- Z_j 表示版本 j 在第 a_j 次故障之后的服务恢复和引入第 j 次修改的服务中断之间的时间，即引入版本 $j+1$；
- $Y_{j,i}$表示版本 j 在第 i 次故障之后的重新启动所花时间；
- W_j 表示引入第 j 个版本所需时间，修改可以离线完成；
- T_j 表示两次版本引入之间的时间。

于是有：

$$T_j = \sum_{i=1}^{a_j} (X_{j,i} + Y_{j,i}) + Z_j + W_j, \qquad j = 1, 2, \cdots$$

上面定义的各种时间区间之间的关系,示于图 9.7 中。

图 9.7　各种不同时间区间之间的关系

两次修改之间的故障数(a_j),与一系列因素有关:

- 系统的故障率;
- 错误的性质(如:诊断错误必需的时间、由该错误引起的后果,等等);
- 系统所处生命周期的不同阶段;
- 维护小组的效能,等等。

有三种极端的特殊情况值得注意,它们是:

① $a_j = 1, \forall j[Z_j = 0]$:它与两种情况有关,某些软件可靠性模型的假设,以及关键系统在一次危险的故障发生之后。这种情况要求仅在一次系统修改之后就恢复服务。

② $a_j = \infty, \forall j[Z_j = 0]$:它也与两种情况有关,对损坏的硬部件更换新的部件,以及软件在故障之后,更换输入模式,不作任何维护,就恢复服务。

③ $a_j = 0$:这种情况对应于保护性维护,如提高系统的正确性、适应性,以及完善系统的功能。它要求在上次维护后不发生任何故障,即引进新版本。

可靠性模型:

设所有的 $Y_{j,i}, W_j$ 和 Z_j 都等于零,即只注意故障过程,而既不考虑恢复服务的时间,也不考虑在一次故障和引入修改之间的可能要占的时间区间。令:

- $t_0 = 0$ 表示起始时刻(假设系统不发生故障);
- $n = 1, 2, \cdots$ 表示故障的次数,即第 n 次系统故障,事实上就是版本 j 的第 i 次故障。

于是有:

$$n = \sum_{k=1}^{j} a_{k-1} + i, \quad j = 1, 2, \cdots, \quad i = 1, 2, \cdots, \quad a_j, \quad a_0 = 0 \qquad (9.3.16)$$

- t_n, $n = 1, 2, \cdots$, 表示故障出现的时刻;
- $f_{x_j}(t), j = 1, 2, \cdots$, 表示无故障时间 $X_{j,x}$ 的概率密度函数 pdf;$sf_{x_j}(t)$ 表示 $X_{j,i}$ 的残存函数($X_{j,i}$ 被假定为对于给定的版本,都是统计恒同的);
- $f_n(t)$ 和 $\Phi_n(t)$ 分别表示故障出现时刻的 pdf 和分布函数,$n = 1, 2, \cdots$;

·N(t)表示在区间[0, t]内已发生的故障数, H(t)为 N(t)的期望: H(t) = E[N(t)]。

在假设条件"统计独立"下, 有:

$$\lambda_n(t) = \sum_{k=1}^{j-1} {}^{*a_k} f_{x_k}(t) * {}^{*i} f_{x_j}(t), \quad j = 1, 2, \cdots, \quad i = 1, 2, \cdots, a_j, \quad a_0 = 0 \tag{9.3.17}$$

其中符号"*"表示对合演算(convolution operation), $f_{x_k}(t)^{*a_k}$表示 $f_{x_k}(t)$的由它自身的 a_k-折迭对合(a_k-fold convolution), 且 $\sum_{k=1}^{j-1} f_{x_k}(t)$表示 $f_{x_1}(t), f_{x_2}(t), \cdots, f_{x_{j-1}}(t)$的对合。(9.3.17)式是以(9.3.16)式以它的第一项覆盖 j-1 个版本, 且第二项覆盖版本 j 的 i 个故障为条件而推出的。于是有:

$$P\{N(t) \geq n\} = P\{t_n < t\} = \Phi_n(t)$$

$$P\{N(t) = n\} = P\{t_n < t < t_{n+1}\} = \Phi_n(t) - \Phi_{n+1}(t)$$

$$H(t) = \sum_{n=1}^{\infty} nP\{N(t) = n\} = \sum_{n=1}^{\infty} n[\Phi_n(t) - \Phi_{n+1}(t)]$$

$$H(t) = \sum_{n=1}^{\infty} n\Phi_n(t) - \sum_{n=1}^{\infty} (n-1)\Phi_n(t) = \sum_{n=1}^{\infty} \Phi_n(t).$$

设 h(t)表示故障的发生率, 或 ROCOF, $h(t) = \frac{dH(t)}{dt}$, 因此有:

$$h(t) = \sum_{n=1}^{\infty} \lambda_n(t) = \sum_{j=1}^{\infty} \sum_{i=1}^{\infty} \sum_{k=1}^{j-1} {}^{*a_k} f_{x_k}(t) * {}^{*i} f_{x_j}(t) \tag{9.3.18}$$

因为这里并不考虑故障同时出现的问题, 于是故障过程就是正则的或有序的, 且 ROCOF 也就是故障强度。在系统发生了 n 次故障以后, 其条件可靠性就是与 n+1 次故障有关的残存函数。它定义为:

$$R_{n+1}(\tau) = P\{X_{j,1} + 1 > \tau \mid t_n\} = sf_{x_j}(\tau), \quad i < a_j;$$

$$R_{n+1}(\tau) = P\{X_{j+1,1} > \tau \mid t_n\} = sf_{x_{j+1}}(\tau), \quad i = a_j.$$

主要在系统的开发阶段, 我们才对之感兴趣, 因为这时主要关心的是下一次故障发生的时间。但在系统的运行阶段, 主要关心的是无故障时间区间[t, t+τ]。考虑下列互斥事件:

$$E_0 = \{t + \tau < t_1\};$$

$$E_n = \{t_n < \tau < t + \tau < t_{n+1}\}, \quad n = 1, 2, \cdots.$$

事件 E_n 表示在时刻 t 之前, 确实发生了 n 次故障, 而在区间[t, t+τ]内不出现故障。于是, 有:

$$R(t, t+\tau) = \sum_{n=0}^{\infty} P\{E_n\}.$$

事件 E_n 的概率为:

$$P\{E_n\} = P\{t_n < t < t + \tau < t_n + X_{j,i+1}\} \tag{9.3.19}$$

$$P\{E_n\} = \int_0^t P\{x < t_n < x + dx\} \cdot P\{X_{j,i+1} > t + \tau - x\}$$

$$= \int_0^t sf_{x_j}(t + \tau - x)\, \lambda_n(x)\, dx \tag{9.3.20}$$

于是, 可靠性具有下面的表达式:

$$R(t,t+\tau)=sf_{x_1}(t+\tau)+\sum_{j=1}^{\infty}\sum_{i=0}^{a_j-1}\int_0^t sf_{x_j}(t+\tau-x)\,\sum_{k=1}^{i}a_{k-1}+i(x)\,dx \tag{9.3.21}$$

它又可以写成:

$$R(t,t+\tau)=sf_{x_1}(t+\tau)+\sum_{j=1}^{\infty}\sum_{i=0}^{a_j-1}sf_{x_j}(t+\tau)*\sum_{k=1}^{j}a_{k-1}+i(t) \tag{9.3.22}$$

显然,实际上它是很难以利用的,但对于 $\tau\ll t$,由(9.3.18)和(9.3.22)式不难得出:

$$R(t,t+\tau)=1-h(t)\tau+o(\tau) \tag{9.3.23}$$

(9.3.23)式除了它的形式简捷以外,在实用上也是很有价值的,可应用于相对于系统寿命 t 而言,τ很小的场合。

上面的推导过程,构成更新理论的一种简单推广,以及对于非定常过程的更新过程概念的推广。在传统的理论中:

- 所有的 $X_{j,i}$都是随机恒同的,即,$f_{x_j}(t)=f_x(t)$,对所有的 j 成立;
- $H(t)$为更新函数,$h(t)$为更新密度。

图 9.8 故障率

设 $X_{j,i}$满足指数分布:$f_{x_j}(t)=\lambda\exp(-\lambda t)$。内部故障出现时间构成分段的泊松过程。这里对所有 λ的大小并未作任何形式的假设,但仍假定在 r 次修改之后,故障过程收敛于一个泊松过程。即:① 不再进行任何形式的修改,或②如果要作某形修改,它们也不会影响到系统的故障行为。设:$\lambda_1,\lambda_2,\cdots,\lambda_r$为这些故障率的序列(图 9.8)。

于是,故障强度 $h(t)$(见式(9.3.18))的拉普拉斯变换为:

$$h(s)=\sum_{j=1}^{r-1}\sum_{k=0}^{j-1}\left(\frac{\lambda_k}{\lambda_k+s}\right)^{a_k}\prod_{i=1}^{a_j}\left(\frac{\lambda_j}{\lambda_j+s}\right)^{i}+\frac{\lambda_r}{s}\prod_{k=1}^{r-1}\left(\frac{\lambda_k}{\lambda_k+s}\right)^{a_k}$$

省去关于 $h(t)$的冗长乏味的推导,可以得出:

- $h(t)$为一时间的连续函数,且 $h(0)=\lambda_1$,$h(\infty)=\lambda_r$。
- 如所有的 a_j 都是有限的,则可靠性增长的条件为:$\lambda_1\geq\lambda_2\geq\cdots\geq\lambda_j\geq\cdots\geq\lambda_r$;$a_j$ 越小,可靠性增长越快;如个别故障率有所增加,则与之对应的故障强度也增加。
- 如 $a_1=\infty$,又不作任何修改,则为一典型的更新过程,此时有 $h(t)=\lambda_1$。

上面关于故障强度与可靠性的变化关系,示于图 9.9 中。

要实际应用上面给出的模型,故障数据可以按两种不同的形式来收集:(a) 内部故障时间(即完全数据);(b) 单位时间内发生的故障数(即不完全数据)。一般而言,故障率模型更适于内部故障时间数据,而故障强度模型则更适于单位时间内发生的故障数的数据形式。

§ 9.4 恢复块结构技术

G.Pucci 应用恢复块结构技术(Recovery Block Structures- RBS),对软件可靠性等软件质量进行估计。

在设计软件时,提供有用的冗余,对软件根据共同的规格说明书作多种不同形式的设计,其冗余就构成一系列独立开发出的软件版本。将这些版本组织成一个系统的方法之一,就是恢复块(RB)技术。RB 由一个不同的程序的版本的集合(每个版本称为一个选择—— Alternate),以及一个称为验收试验(Acceptance Test —— AT)的查错例行程序组成。在第一个选择执行之后,验收试验对它的执行结果进行检验,如结果不正确,则该选择的状态被恢复到 RB 的入口,并开始执行另一选择。如此循环下去,直到某个选择能得出被验证为正确的结果,或再没有别的选择可资应用为止。最后,由一嵌入环境(Embedding Environment)提供更高层次的恢复过程。

图 9.9 故障强度与系统可靠性的变化关系

一个 RB 的执行可能受到不同类型的错误的影响。某部件(即选择或验收试验)中的错误,被视为一个与规格说明书中的要求不符合的行为的一种表象,且错误也能影响到选择或验收试验,而且它们能够在不同的部件中,由相同的输入激活。G.Pucci 根据错误对软件系统所产生的可观察到的后果,对它们进行分类。这样,就可将测试数据应用于模型参数的估计。

设一 RB 由 N 个计算模块组成,其中计算模块 i 由选择 i 的所有执行,以及随后的 AT 对选择结果的验证组成。控制的转移发生在查出错误的表象期间。对于每个计算模块 i, i= 1, 2, …, N, 可以分别有下列四种类型的事件发生:

① E_i^1(正确)。当选择 i 被激活后,产生正确的结果。结果为 AT 接受。

② E_i^2(出错)。当选择 i 被激活后,产生不正确的结果。结果为 AT 拒绝。

③ E_i^3(出错)。当选择 i 被激活后,产生正确的结果。结果为 AT 拒绝。

④ E_i^4(出错)。当选择 i 被激活后,产生不正确的结果。结果为 AT 接受。

它们被枚举了,并且互不相交。即:

$$P\{\bigcup_{j=1}^{4} E_i^j\} = 1, \quad P\{E_i^{j_1} \cap E_i^{j_2}\} = 0, \quad 1 \leq j_1 < j_2 \leq 4.$$

它们的特性用表 9.1 归纳如下:

表 9.1 四种事件的特性

事件类型	受影响的部件	是否被查出	恢复情况
1	无	——	——
2	选择 i	是	其它的选择

3	AT	是	更高层次的 RB
4	选择 i, AT	否	无

G. Pucci 建立的分析容错软件系统的故障过程的模型是离散状态、离散参数的马尔可夫链。它有状态 1, 2, …, N 和 2′, 3′, …, N′, 它们分别对应于 N 个计算模块在不同的出错条件下的执行。另外还有三个状态: 状态 UF, 表示一个未被查出的故障的出现; 状态 ATF, 表示由于 AT 中的错误引起的、被查出的故障的出现; 状态 AF, 表示由于选择中的错误引起的, 被查出的故障的出现。状态的转换模式示于图 9.10 中。

图 9.10　马尔可夫链的转换模式

所有可能的状态转换的集合(具非零概率的那些状态转换)是:

$$T = \{(i \to j): i = 1, 2, \cdots, N-1;\ j = 1, i+1, (i+1)', UF\}$$
$$\cup \{(N \to j): j = 1, ATF, AF, UF\}$$
$$\cup \{(i \to j): i = 2', 3', \cdots, (N-1)';\ j = (i+1)', UF\}$$
$$\cup \{(N' \to j): j = ATF, UF\}$$

由状态 i 向状态 1 的转换, 都是第 i 个计算模块的一次成功执行, 这一类转换所涉及的是类型 1 的事件。由状态 i 向状态 i+1 的转换, (i = 1, 2, …, N-1), 表示由 AT 在第 i 个选择中查出一个错误, 激活了第 i+1 个计算模块(类型 2 的事件发生)。在这种情况下, 最后的选择被查出发生故障, 马尔可夫过程达到状态 AF 的吸收。转换至状态 UF 的所有转换, 都与一未被查出的错误(类型 4 的事件)有关, 且对应的 RB 未被查出故障。状态 i 向状态(i+1) 的转换, 表示类型 3 的事件的出现, 即表示由选择 i 提供的正确结果被拒绝了。每个状态 i′, i = 2, 3, …, N, 表示在类型 3 的一事件发生之后的第 i 个计算模块的执行。由这些状态出发, 马尔可夫过程不能达到状态 1, 只能由状态 ATF 或 UF 吸收。

考察事件 A_i, "在首先的 i-1 次 AT 的执行过程中, 没有类型 3 的错误", i = 1, 2, …, N, 则集合 T 中的转换概率为:

- $i = 1, 2, \cdots, N$:

  $P\{i \to 1\} = p_{A_i} = P(E_i^1 \mid A_i)$ = 在先给定 AT 的一次正确行为的条件下, 由选择 i 提供的正确结果被接受的、选择 i 的成功激活的概率;

- $i = 1, 2, \cdots, N-1$:

  $P\{i \to i+1\} = P\{N \to NF\} = q_{A_i} = P(E_i^2 \mid A_i)$ = 在先给定 AT 的一次正确行为的条件下, 在选择 i 被激活时, 查出选择 i 的一次不正确行为的概率;

- $i = 1, 2, \cdots, N-1$:

  $P\{i \to (i+1)'\} = P\{N \to ATF\} = q^D_{AT_i} = P(E_i^3 \mid A_i)$ = 在先给定 AT 的一次正确行为的条件下, 在选择 i 被激活时, 由选择 i 提供的正确结果被拒绝的概率;

· $i = 1, 2, \cdots, N$:

$P\{i \rightarrow UF\} = q^{U}_{AT_i} = P(E_i \cap A_i)$ = 在先给定 AT 的一次正确行为的条件下, 由选择 i 提供的不正确结果被接受的概率;

· $i = 2, \cdots, N$:

$P\{i' \rightarrow UF\} = r^{U}_{AT_i} = P(E_i \cap A_i)$ = 在先给定 AT 的一次不正确行为的条件下, 由选择 i 提供的不正确的结果被接受的概率;

· $i = 2, \cdots, N-1$:

$P\{i' \rightarrow (i+1)'\} = P\{N' \rightarrow ATF\} = r^{D}_{AT_i} = 1 - r^{U}_{AT_i}$ = 在先给定 AT 的一次不正确行为的条件下, 由选择 i 提供的任何结果被拒绝的概率。

参数 $P_{A_i}, q_{A_i}, q^{D}_{AT_i}, q^{U}_{AT_i}, r^{U}_{AT_i}, r^{D}_{AT_i}$, 可以根据上面定义的 4 种类型的事件获得。下面讨论怎样将上面的模型应用于可靠性估计。

可以使用它们以获取 RBS 的关于将来行为的信息。第一个可靠性测度是: 在发生故障之前, RB 的正确执行次数的分布。即推导出离散的强度函数的表达式:

$$f(k) = P\{\text{在发生故障前的 } k \text{ 次成功执行}\}$$

为此, 要考虑两个不同的分布: $f_T(k)$: 在任何 RB 故障(无论查出与否)出现以前的执行次数; $f_U(k)$: 直到一个未被查出的故障出现为止的执行次数。f_T 和 f_U 都与吸收前对状态 1 的访问次数有关。设 π 表示在马尔可夫链中, 从状态 1 到状态 1 的状态间所有不同路径的集合:

$$\pi = \{\text{路径 } \pi \mid \forall k \in \{1, 2, \cdots, N\}) \wedge (\pi = (1 \rightarrow 2 \rightarrow \cdots \rightarrow k-1 \rightarrow k \rightarrow 1))\}.$$

π 表示从状态 1 到状态 1 的状态间所有被查出故障的不同路径的集合, 有:

$$\pi = \pi \cup \{\text{路径 } \pi \mid \forall k \in \{1, 2, \cdots, N-1\}) \wedge (\pi = (1 \rightarrow 2 \rightarrow \cdots \rightarrow k \rightarrow (k+1)' \rightarrow \cdots \rightarrow N' \rightarrow ATF))\} \cup \{1 \rightarrow 2 \rightarrow \cdots \rightarrow N \rightarrow ATF), (1 \rightarrow 2 \rightarrow \cdots \rightarrow N \rightarrow AF)\}.$$

设 p^{T}_{RB} 和 p^{U}_{RB} 分别表示在集合 π_T 和 π_U 中路径出现的概率之和, 则:

$$P^{T}_{RB} = \sum_{\pi \in \pi_T} \prod_{(i \rightarrow j) \in \pi} p_{ij}, \quad P^{U}_{RB} = \sum_{\pi \in \pi_U} \prod_{(i \rightarrow j) \in \pi} p_{ij}.$$

它们说明(参见 E. Inlar, Introduction to Stochastic Processes, Englewood Cliffs NJ: Prentice-Hall, 1975) f_T 和 f_U 为修改的几何分布:

参数为:

$$q^{T}_{RB} = 1 - p^{T}_{RB}, \quad q^{U}_{RB} = 1 - p^{U}_{RB},$$

$$f_T(k) = q^{T}_{RB}\,(1 - q^{T}_{RB})^{k}, \quad f_U(k) = q^{U}_{RB}\,(1 - q^{U}_{RB})^{k}.$$

因为:

$$p^{U}_{RB} = p^{T}_{RB} + \sum_{i=1}^{N} \prod_{j=1}^{i-1} q_{A_j}\, q^{D}_{AT_i} \prod_{k=i+1}^{N} r^{D}_{AT_k} + \prod_{i=1}^{N} q_{A_i} \tag{9.4.1}$$

$$p^{T}_{RB} = \sum_{i=1}^{N} p_{A_i} \prod_{j=1}^{i-1} q_{A_j} \tag{9.4.2}$$

于是:

$$\frac{q^{U}_{RB}}{q^{T}_{RB}} = 1 - \frac{\sum_{i=1}^{N} \prod_{j=1}^{i-1} q_{A_j}\, q^{D}_{AT_i} \prod_{k=i+1}^{N} r^{D}_{AT_k} + \prod_{i=1}^{N} q_{A_i}}{q^{T}_{RB}}$$

是在给定一次故障出现的条件下,RB 的故障属于未查出的错误引起的概率。于是显而易见,有:

$$1 - \frac{q_{RB}^{U}}{q_{RB}^{T}} = \frac{\sum_{i=1}^{N} \prod_{j=1}^{i-1} q_{A_j} \, q_{AT_i}^{D} \prod_{k=i+1}^{N} r_{AT_k}^{D} + \prod_{i=1}^{N} q_{A_i}}{q_{RB}^{T}}$$

表示故障属于不可容允的未查出的错误的概率。

假设所有孤立的选择都具有相同的故障概率,即:

$$P\{\text{选择 } i \text{ 产生的不正确结果}\} = q, \quad i = 1, 2, \cdots, N$$

又设所有四种类型的错误事件都互相独立,于是有:

$$q_{A_i} = qp_{AT}^{C}, \quad p_{A_i} = (1-q)p_{AT}^{C}, \quad (i = 1, 2, \cdots, N)$$

其中,p_{AT}^{C} 就是 AT 对一随机输入有正确行为的概率。由(9.4.1)式有:

$$\begin{aligned} p_{RB}^{T} &= (1-q)p_{AT}^{C} \sum_{i=1}^{N} (qp_{AT}^{C})^{i-1} \\ &= (1-q)p_{AT}^{C} \frac{1-(qp_{AT}^{C})^{N}}{1-qp_{AT}^{C}} \leq \frac{(1-q)p_{AT}^{C}}{1-qp_{AT}^{C}} \leq p_{AT}^{C} \end{aligned} \tag{9.4.3}$$

因此,在部件间的统计独立的情况下,RB 最多只能达到 AT 的可靠度。

从概率论的观点出发,可以将 RB 对于不同输入的执行看作一个具有 N + 1 个结局的独立试验的序列,出故障的概率为 q_{RB}^{T},i-成功的概率($i = 1, 2, \cdots, N$)为:

$$p_i = p_{A_i} \prod_{j=1}^{i-1} q_{A_j} \ .$$

注意: $q_{RB}^{T} + \sum_{i=1}^{N} p_i = q_{RB}^{T} + p_{RB}^{T} = 1$.

设 X_i, $i = 1, 2, \cdots, N$, 表示一个无穷试验序列,在第一次故障出现之前,出现 i-成功的次数。用 RB 的术语讲,就是 X_i 表示在任何 RB 故障出现之前,由选择 i 正确地处理的输入数,X_i 的联合分布是:

$$P\{X_1 = k_1, \cdots, X_N = k_N\} = \frac{(k_1 + k_2 + \cdots + k_N)!}{k_1! k_2! \cdots k_N!} q_{RB}^{T} \prod_{i=1}^{N} p_i^{k_i}.$$

它的概率母函数(probability generating function- pgf)为:

$$\begin{aligned} G(z_1, \cdots, z_N) &= E \prod_{i=1}^{N} z_i^{X_i} \\ &= \sum_{k_1=0}^{\infty} \cdots \sum_{k_N=0}^{\infty} \frac{(k_1 + \cdots + k_N)!}{k_1! \cdots k_N!} q_{RB}^{T} \prod_{i=1}^{N} (p_i z_i)^{k_i} \\ &= \frac{q_{RB}^{T}}{1 - \sum_{i=1}^{N} p_i z_i}. \end{aligned}$$

于是可得到 X_i 的临界分布的 pgf :

$$G(1, \cdots, z_i, \cdots, 1) = E[z_i^{X_i}] = \frac{q_{RB}^{T}}{q_{RB}^{T} + p_i - p_i z_i}.$$

它为参数:

$$q_i' = \frac{q_{RB}^T}{q_{RB}^T + p_i}$$

的修改几何分布的 pgf, 于是有 X_i 的期望为:

$$E[X_i] = \frac{p_i}{q_{RB}^T} = \frac{p_{A_i}}{q_{RB}^T} \prod_{j=1}^{i-1} q_{A_j} .$$

利用上面讨论的离散模型, 还可以计算其连续可靠性测度。为此, 设所有选择, 以及 AT 的平均执行时间分别为:

$$\frac{1}{\lambda_i},\ i = 1, 2, \cdots N, \quad \frac{1}{\lambda_T}$$

定义:

$$C > \sum_{i=1}^{N} E[X_i] \left(\frac{i}{\lambda_T} + \sum_{j=1}^{i} \frac{1}{\lambda_j} \right)$$

$$= \sum_{i=1}^{N} \frac{p_{A_i}}{q_{RB}^T} \sum_{j_1=1}^{i} q_{A_{j_1}} \left(\frac{i}{\lambda_T} + \sum_{j_2=1}^{i} \frac{1}{\lambda_{j_2}} \right)$$

表示在 RB 故障出现之前的正确执行的平均累积时间。于是 RB 的平均不发生任何故障的时间(Mean Time To Any Failure for the RB- $MTTAF_{RB}$)为:

$$MTTAF_{RB} = C + I_D + I_U .$$

其中, I_D 和 I_U 是引发故障的最后的输入的基值。它们分别为:

$$I_D = \frac{\frac{N}{\lambda_T} + \sum_{j=1}^{N} \frac{1}{\lambda_j} \prod_{i=1}^{N} q_{A_i} + \sum_{i=1}^{N} \prod_{j_1=1}^{i-1} q_{A_{j_1}} q_{AT_i}^D \prod_{j_2=i+1}^{N} r_{AT_{j_2}}^D}{q_{RB}^T};$$

$$I_U = \frac{\sum_{i=1}^{N} \left(\frac{i}{\lambda_T} + \sum_{j=1}^{i} \frac{1}{\lambda_j} \right) \prod_{j_1=1}^{i-1} q_{A_{j_1}} q_{AT_i}^U + \sum_{j_1=1}^{i} \prod_{k_1=1}^{j_1-1} q_{A_{k_1}} q_{AT_{j_1}}^D \prod_{k_2=j_1+1}^{i-1} r_{AT_{k_2}}^D r_{AT_i}^U}{q_{RB}^T}.$$

定义 $C > \sum_{i=1}^{N} E[X_i] \left(\frac{i}{\lambda_T} + \sum_{j=1}^{i} \frac{1}{\lambda_j} \right) + \frac{1}{\rho} + \frac{N}{\lambda_T} + \sum_{j=1}^{N} \frac{1}{\lambda_j} E[D]$, 其中, $\frac{1}{\rho}$为在一个被查出的故障出现之后, 为恢复 RB 的执行所需要的平均修复时间(MTTR)。同时, 可以将 RB 对于随机输入的执行, 处理成具有 N + 2 个结局的独立试验的一个序列: 未被查出的故障, 具有概率 q_{RB}^U; i-成功, 具有概率 p_i, i= 1, 2, …, N; 被查出的故障, 具有概率 $q_{RB}^D = q_{RB}^T - q_{RB}^U$。于是, 平均不发生未被查出的故障的时间(Meam Time To Undetected Failure- $MTTUF_{RB}$)为:

$$MTTUF_{RB} = C + I_U .$$

以上模型的参数值, 可由容错软件系统在它的开发期间的故障历史推出。故障历史可在软件测试阶段予以收集。

下面考察一个 RB 系统, 以及它的开发过程。为了将重迭出现的错误的影响降至最小, 编码是由不同的程序员小组完成的, 他们实现的部件都各不相同。由单个选择的故障历史, 可以推出:

$$p_{A_i} = P\{\text{选择 } i \text{ 关于随机输入的正确执行}\}$$

$\tilde{q}_{A_i}$ = P{选择 i 关于随机输入的错误多发行为}

p_{AT} = P{AT 的正确行为}

$\tilde{q}_{AT}$ = P{AT 的不正确行为}

由于对单个错误不相容部件之间不发生任何相互作用,所以这时可以将现存许多黑盒子类软件可靠性增长模型使用于部件测试阶段。对于任一部件 $C \in \{A_i, AT \mid i = 1, 2, \cdots, N\}$ 的这些软件可靠性增长模型的最后输出,通常都是一连续的可靠性函数:

$$R_C(t) = P\{T_C \geq t\}.$$

其中,随机变量 T_c 表示 C 的下一次故障的发生时间。于是,可以得到对于一随机输入的正确执行的概率 pC:

$$pC = R_C(t_C^m),$$

其中,t_C^m 是 C 的总执行时间,即为处理一单个输入,部件 C 所要求的时间。

针对单个部件,为收集适用的故障数据,一种典型的测试策略,可称之为操作剖面测试(operation profile testing),即根据适当的使用分布而产生的输入数据,对部件作测试。一旦由数据估计出这些参数的值,就可以使用上面讨论的 RBS 模型,这时有:

$$p_{A_i} = p_{A_i} p_{AT},\ q_{A_i} = q_{A_i} p_{AT},\ q^D_{AT_i} = p_{A_i} q_{AT},\ q^U_{AT_i} = r^U_{AT_i} = q_{A_i} q_{AT}.$$

注意,在应用上面的模型时,要保持不同的部件之间的独立性。例如,上面设定 $q^U_{AT_i} = r^U_{AT_i}$,就要求在部件测试期间,E_i^4 类的事件 A_i 必须是独立的。如果任意两个部件间的不正确行为的相关性存在,则这样的设定将产生乐观的估计。

部件测试之后,进行集成测试(integration testing),即将整个系统在某一输入的集合上进行测试。在这期间,可以收集直接用于参数估计的故障历史。只要对一次特定的运行,关于计算是怎样逐步演进的作持续不断的追踪,就可以观察到 RB 的一次执行过程中出现的错误与部件的错误事件的对应关系。虽然理论上可以收集到用于模型参数的估计的完整的故障数据集合,但事实上,这是一个十分艰难的任务。为获得足够有用的信息,要求的输入数据是大量的。在目前,尚缺乏有效的测试策略,以产生足够的输入数据,对模型参数进行估计。

但可以采用某些方法,对它求一些特解,以作近似估计。一种可能的方法就是将部件分割成一些相互独立的相关组(dependency groups),而组与组之间不存在任何的相关性。

作为例子,设一 RBS 有两个选择:A_1 和 A_2,它的一个可能的分割就是:$\pi = \{S_1, S_2\}$,$S_1 = \{A_1, A_2\}$,$S_2 = \{AT\}$。对它的分割是根据假设:选择的错误事件与 AT 的行为是独立的。将相同的输入同时给 S_1 的两个部件,并比较它们的结果。在测试之后,可得出参数估计:

p_{A_2} = P{A_2 正确 | A_1 不正确}

q_{A_2} = P{A_2 不正确 | A_1 不正确}

将这些值与对 AT 测试所获得的值作比较,就可获得 RBS 模型的参数估计值。如果在选择和 AT 之间存在依赖性,则模型给出的估计结果肯定是不精确的,但可获得 A_1 和

A_2 之间相关性的效果，这对于改进它们的质量，仍是有实际意义的。

§ 9.5 对测试过程的直接建模

T. Downs 等人针对现有大多数软件可靠性模型在测试阶段，将软件处理成黑盒子的情况，考虑对测试过程直接建模。为此，他提出一条引理，用以描述路径测试的基本情况。

引理：如果在每两次排错之间的时间区间内的执行轮廓不变，则在任一个这样的时间区间内的故障率 λ 可以表示成：

$$\lambda = - r \ln p \qquad (9.5.1)$$

其中，p 为被选定的一条路径无错执行的概率；

r 为每单位时间内测试的路径数（即路径测试率）。

1. 非均匀路径选择模型（NU）

假设：一个软件系统可以被划分成两部分：着重测试的部分和一般测试的部分，分别记为部分 1 和部分 2。于是可以建立测试轮廓（testing profile）如下：对于任何一次测试运行而言，一条路径在执行过程中经过部分 1 的概率为 p，经过部分 2 的概率为 1- p。

于是，在从软件中排除一个错误之前，应用上面的引理，可以得出故障率为：

$$\lambda = - r \ln[p(1 - \alpha_1)^{N_1} + (1 - p)(1 - \alpha_2)^{N_2}] \qquad (9.5.2)$$

其中，N_i 为部分 i 中的初始错误个数 $(i = 1, 2)$，

M_i 为经过部分 i 的路径数 $(i = 1, 2)$，

C_i 为在部分 i 中，每个错误影响到的路径数 $(i = 1, 2)$，

$\alpha_i = C_i / M_i$ $(i = 1, 2)$.

λ 中用作下标的数字 j，表示 λ 是 j 个错误被排除后的故障率。对于 λ，即表示一个错误也没有被排除时的故障率。

在许多情况下，因为（9.5.2）式中的 α_i 很小，所以可以得出（9.5.2）式的简略表达式为：

$$\lambda = k[N_1 + CN_2] \qquad (9.5.3)$$

其中，k 和 C 都是正的常数。

在 j 个错误被排除后，故障率变为：

$$\lambda_j = k\left[N_1^{(j)} + CN_2^{(j)}\right] \qquad (9.5.4)$$

其中，

$$N_1^{(j)} = N_1^{(j-1)}\left(1 - \frac{1}{N_1^{(j-1)} + CN_2^{(j-1)}}\right), \quad N_2^{(j)} = N_2^{(j-1)}\left(1 - \frac{1}{N_1^{(j-1)} + CN_2^{(j-1)}}\right) \qquad (9.5.5)$$

2．部分 2 中的错误个数远大于部分 1 中的非均匀模型(NUI)

大量的数据分析结果说明，在部分 2 中的错误个数远多于部分 1 中的，即 $N_2 \gg N_1$，这是因为部分 2 很少被测试。随着错误不断地从软件系统中被排除掉，即 j 不断增大，$N_1^{(j)}$ 下降的速度远快于 $N_2^{(j)}$。于是，可以假设 $N_2^{(j)} = N_2$，由此对(9.5.4)再简化，得：

$$\lambda_j = k[N_1^{(j)} + CN_2] = k[N_1^{(j)} + Q] \tag{9.5.6}$$

$$N_1^{(j)} = N_1^{(j-1)}\left(1 - \frac{1}{N_1^{(j-1)} + Q}\right) \tag{9.5.7}$$

其中，Q 是一个常数。

3．错误影响到路径的随机数的模型(RP)

在上面两个模型中，均假设在一给定的部分中的每个错误，都影响到同样多的路径数，这是一个过强的假设。这里假设任意一个错误所影响到的路径数是一个随机变量，为使数学上处理容易，下面的讨论中，只考虑到任何路径仅含有一个错误的情况，这就给由任一错误所影响到的路径数强加了一个限制。设在一软件系统中，有 N 个初始错误分布于 M 条路径上，并令随机变量 X_0 表示在排错之前，由任一错误所影响到的路径数。则对于 X_0 的限制就是，它的取值不能大于 gilb(M/N)。设 $m \equiv \text{gilb}(M/N)$，则

$$P_r\{\text{任一路径无错}\} = 1 - NE[X_0]/M \tag{9.5.8}$$

其中，$E[X_0] = \sum_{i=1}^{m} iP_r\{X_0 = i\}$。

根据引理，任何排错前的故障率为：

$$\lambda = -r\ln[1 - NE[X_0]/M] \tag{9.5.9}$$

于是有下面的递归关系成立：

$$P_i^{(k)} = \frac{P_i^{(k-1)}}{N-k}\left(N - k + 1 - \frac{i}{E[X_{k-1}]}\right) \tag{9.5.10}$$

其中，$E[X_{k-1}] = \sum_{i=1}^{m} iP_i^{(k-1)}$，$P_i^{(k)}$ 表示在 k 个错误被查出并从软件系统中被排除之后(在均匀测试条件下)，任一路径含有一个 i 类错误(一个错误影响到 i 条路径，则称该错误为“i 类错误”)的概率。

在 k 个错误被排除之后的故障率为：

$$\lambda = -r\ln\left(1 - \frac{N-k}{M}E[X_k]\right) \tag{9.5.11}$$

4．非均匀执行下的 RP 模型(NURP)

本模型是 NU 和 RP 两个模型的简单混合型。它假设一软件系统可划分为两个部分：着重测试的部分 1，和一般测试的部分 2，且任一错误所影响的路径数为随机变量。任一测试运行路径经过部分 1 的概率为 p，经过部分 2 的概率为 1- p。在未作任何排错之前，该软件系统的故障率为：

$$\lambda = -\ r\ln\ p\ \left(1-\frac{N_1E[X_{0,1}]}{M_1}\right) + (1-\ p)\ \left(1-\frac{N_2E[X_{0,2}]}{M_2}\right) \tag{9.5.12}$$

其中，$N_j(j=1,2)$为部分 j 中的初始错误个数，

$M_j(j=1,2)$为经过部分 j 的路径数，

$X_{0,j}(j=1,2)$为在未作任何排错之前，在部分 j 中由任一错误影响到的路径数，它是一随机变量。

注意到 $M_j(j=1,2)$是一个很大的数(特别当软件系统很大时更是如此)。(9.5.12)式可以简化为下面的形式：

$$\lambda \approx k\{N_1E[X_{0,1}] + CN_2E[X_{0,2}]\} \tag{9.5.13}$$

其中，

$$k = rp/M_1,$$

$$C = \frac{(1-p)}{p}\cdot\frac{M_1}{M_2}.$$

因此可以得出任一经过部分 $j(j=1,2)$的路径含有错误的概率为：

$$P_{f_j}(0) = N_jE[X_{0,j}]/M_j, \quad (j=1,2) \tag{9.5.14}$$

当第一个错误被找到时，它出现在部分 1 或部分 2 的概率分别为 $P_{l_1}(0)$，$P_{l_2}(0)$，它们是：

$$P_{l_1}(0) = \frac{pP_{f_1}(0)}{pP_{f_1}(0) + (1-p)P_{f_2}(0)} \tag{9.5.15}$$

$$P_{l_2}(0) = 1 - P_{l_1}(0) \tag{9.5.16}$$

利用(9.5.11)～(9.5.16)式，可以推出下面的递推关系式：

$$N_1^{(k)} = N_1^{(k-1)}\left(1-\frac{E[X_{k-1,1}]}{N_1^{(k-1)}E[X_{k-1,1}] + CN_2^{(k-1)}E[X_{k-1,2}]}\right)$$

$$N_2^{(k)} = N_2^{(k-1)}\left(1-\frac{CE[X_{k-1,1}]}{N_1^{(k-1)}E[X_{k-1,1}] + CN_2^{(k-1)}E[X_{k-1,2}]}\right) \tag{9.5.17}$$

其中，$N_j^{(k)}(j=1,2)$表示当从软件系统中排除 k 个错误之后，部分 j 中的错误个数。

$X_{k-1,j}(j=1,2)$表示当从软件系统中排除 k-1 个错误之后，部分 j 中任一错误所影响到的路径数。

于是，在排除掉 k 个错误之后，软件系统的故障率为：

$$\lambda = k[N_1^{(k)}E[X_{k,1}] + CN_2^{(k)}E[X_{k,2}]] \tag{9.5.18}$$

下面讨论 $E[X_{k-1,j}]$的计算过程：

设 $P_{i,j}$表示在测试的开始，任意一条路径经过部分 j 时含有一个 i 类错误的概率，则：

$$P_r\{\text{在部分 } j \text{ 中查出的第一个错误为 } i \text{ 类错误}\} = \frac{iP_{i,j}}{E[X_{0,j}]}.$$

因此，

$$P_r\{\text{在部分 } j \text{ 中查出了第一个错误且为 } i \text{ 类错误}\} = \frac{iP_{i,j}}{E[X_{0,j}]} - P_{l_j}(0).$$

于是，在第一个错误被排除后，在部分 j 中剩下的 i 类错误的统计期望为：

$$N_jP_{i,j} - \frac{iP_{i,j}}{E[X_{0,j}]}P_{l_j}(0) = P_{i,j}\ N_j - \frac{iP_{l_j}(0)}{E[X_{0,j}]} \tag{9.5.19}$$

又设:

$P_{i,j}^{(k)}$ 表示在 k 个错误被从软件系统中排除以后,任一经过部分 j 的路径含有一个 i 类错误的概率,则(9.5.19)式可重写为:

$$P_{i,j}\ N_j - \frac{iP_{l_j}(0)}{E[X_{0,j}]} = [N_j - P_{l_j}(0)]P_{i,j}^{(1)}.$$

给定

$$P_{i,j}^{(1)} = \frac{P_{i,j}}{N_j - P_{l_j}(0)}\ N_j - \frac{iP_{l_j}(0)}{E[X_{0,j}]}\ .$$

在 k 个错误被排除以后,我们有:

$$P_{i,j}^{(k)} = \frac{P_{i,j}^{(k-1)}}{N_j^{(k-1)} - P_{l_j}^{(k-1)}} N_j^{(k-1)} - \frac{iP_{l_j}(0)}{E[X_{0,j}]}.$$

于是(9.5.17)式中要求的统计期望为:

$$E[X_{k-1,j}] = \sum_{i=1}^{m_j} iP_{i,j}^{(k-1)}, \quad m_j > \mathrm{gilb}(M_j/N_j).$$

5. 在部分 2 中的错误个数远大于部分 1 中的错误数的 NURP 模型(NURPI)

为求完备,下面考虑非均匀执行,即:① 错误影响的路径是随机数,② 排错过程中,在一个部分中含有的错误数为常数(它的变化可忽略不计)。则在排除 k 个错误后,软件系统的故障率为:

$$\lambda_k = k[N_1^{(k)}E[X_{k,1}] + Q \tag{9.5.20}$$

$$N_1^{(k)} = N_1^{(k-1)}\ 1 - \frac{E[X_{k-1,1}]}{N_1^{(k-1)}E[X_{k-1,1}] + Q} \tag{9.5.21}$$

$E[X_{k-1,1}]$ 的意义同以上的讨论。

§ 9.6 测试中观察不到故障时的故障概率的估计

K.W.Miller 等人,开发出以软件的黑盒子模型为基础的理论分析法,解决下列三个问题:① 在随机测试过程中,当观察到的故障次数为零时,如何估计当前版本的故障概率;② 在使用分布(use distribution)与测试分布(test distribution)不匹配时,对估计的故障概率进行调整的方法;③ 在估计概率故障时,将随机测试的结果与其它信息结合起来的分析方法。

在下面的讨论中,"程序"可能指一个单个的过程(procedure),或一个模块的集合类(collection),或者整个的软件系统。

黑盒子模型假设一个程序可处理成一个数学函数,它有着明确的定义域和值域。程序将一单个的多维输入作为输入,而产生出单个的输出。必须得出结果的所有可能的输入的集合,构成程序的程序域(domain of program),相应的输出集合构成程序的值域。

如果定义域的基数很小,则穷举测试(exhansive testing)过程能对给定的某输入分布

(input distribution)准确地判定其故障概率。但是,对于大多数程序而言,穷举测试是不实际的。而事实上,程序正确性证明也是不可能的。于是,随机测试就是保证软件可靠性的重要技术手段。它可以单独使用,也可以与其它技术一起使用。

假定存在一套绝对正确的判定标准(oracle),它能准确地将任何输入/输出对按正确与错误来分类。例如,它能准确划分硬件故障、系统软件错误、二义性的规范说明,等等。下面讨论离散样本空间模型。

设函数 f 以及它的实现程序 F,F 将 f 的定义域分为两类:一类是由 F 能将定义域中的元素映射为正确输出的那些元素组成;另一类是由 F 将 f 的定义域中的元素映射为不正确输出的那些元素组成。函数 f 的作用就相当于是对程序 F 的判定标准。

我们采取这样的做法:如果一元素揭示出 F 中的一个故障,就将一个黑球放入一个罐中,如未揭示出 F 中的故障,则放入一个白球。显而易见,对于每个输入元素,都可能有一个或多个球放入罐中。将一个元素用于测试,就是判定对应的球的颜色。假定测试选择是可以重复的,即对应着以可放回的方式从罐中取球。

设 f 的定义域有限,域中的一个元素就相当于测试条件与数据(test case)。随机测试可在域中对元素进行随机选取用于测试,则有的元素就可能被多次选取。设 p(x)表示 F 的域中的一个元素 x 被选中用于测试的概率;w(x)为标识函数,它表示如果元素 x 揭示出 F 中的一个故障,则 w(x)= 1,否则,w(x)= 0。k 表示某元素被重复选中的次数(k 为整数)。对于 F 的域中的每个 x,就有对应于 x 的 kp(x)个球放入罐中,则 θ就表示罐中黑球所占的真正比例,有:

$$\theta = \sum_{x} p(x)w(x) \qquad (9.6.1)$$

上式表示以可放回的方式取球,取出的黑球所占的比例,而不是取出的球的个数。在一次取球的过程中,从罐中取出一个黑球的概率是 θ,则在一连串的取球过程中,取出 t 个白球的概率是$(1-\theta)^t$。在下面的讨论中,我们设执行了 t 次测试,且它们都没有揭示出一个故障(即所有被观察到的输出都是正确的)。对 θ的估计记为 $\hat{\theta}$。

Laplace 讨论了这样的估计问题。他推出公式 $\hat{\theta} = 1/(t+2)$。下面介绍其推导过程:取出的黑球个数为离散的随机变量,记为 X,于是有二项式模型:

① X 的可能取值为 x= 0, 1, 2, …, t;

② X 的概率密度函数(pdf)为:

$$g(x|\theta) = P_r\{X = x\} = \binom{t}{x}\theta^x(1-\theta)^{t-x}, \quad x = 0, 1, 2, \cdots, t.$$

在 x= 0 时,有:

$$g(0|\theta) = P_r\{X = 0\} = (1-\theta)^t.$$

则有:

$$\sum_{x=1}^{t} g(x|\theta) = P_r\{X > 0\} = 1 - (1-\theta)^t.$$

X 的期望值(即在 t 次测试中,黑球的期望数)为:

$$E[X|\theta] = \sum_{x=0}^{t} xg(x|\theta) = \theta.$$

关于 $E[X\mid\theta]$ 的方差为:

$$E[X^2\mid\theta] - E[X\mid\theta]^2 = \sum_{x=0}^{t} x^2 g(x\mid\theta) - \theta^2 t^2 = \theta(1-\theta)t.$$

在 β-模型中,忽略 Θ的固有离散性,并假设 Θ的先验分布是 $B(a,b)$,即

① Θ的可能值为 $0\leq\theta\leq 1$;

② Θ的 pdf 为 $f(\theta)$,有:

$$f(\theta) = \frac{\theta^{a-1}(1-\theta)^{b-1}}{B(a,b)}, \quad 0\leq\theta\leq 1;$$

③ $a>0, b>0$;

④ 正规化常数 $B(a,b)$ 是一个(完全的) β函数:

$$B(a,b) = \int_0^1 \theta^{a-1}(1-\theta)^{b-1}d\theta = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}$$

Θ的期望值为:

$$E[\Theta] = \int_0^1 \theta f(\theta)d\theta = \frac{a}{a+b}$$

Θ的方差为:

$$E[\Theta^2] - E[\Theta]^2 = \int_0^1 \theta^2 f(\theta)d\theta - \left(\frac{a}{a+b}\right)^2 = \frac{ab}{(a+b)^2(a+b+1)}.$$

Θ的先验 pdf 为 $f(\theta)$, $g(x\mid\theta)$ 为以 Θ的值为条件的 X 的似然函数, $f(\theta\mid x)$ 表示以 X 的观察值为条件的 Θ的先验 pdf。

依贝叶斯定理,在给定 x 的观察值时的 Θ的后验 pdf 为:

$$f(\theta\mid x) = \frac{g(x\mid\theta)f(\theta)}{\int_0^1 g(x\mid\theta')f(\theta')d\theta'}.$$

可证:

$$f(\theta\mid x) = \frac{\theta^{x+a-1}(1-\theta)^{t-x+b-1}}{B(x+a,t-x+b)}, \quad x=0,1,\cdots,t,\ 0\leq\theta\leq 1.$$

即 Θ的后验分布为 $B(x+a,t-x+b)$,其中 x 为在 t 次测试中观察到的黑球个数, a, b 为先验 Θ分布参数。注意到:如 t 次测试中观察到黑球个数为 x,则关于 Θ的期望有如下的变化:

$$\frac{a}{a+b} \to \frac{x+a}{t+a+b}.$$

Θ的方差有如下的变化:

$$\frac{ab}{(a+b)^2(a+b+1)} \to \frac{(x+a)(t-x+b)}{(t+a+b)^2(t+a+b+1)}.$$

选择 a 和 b 的值,没有简单的规则可循,但我们可作如下处理:

$$\hat{\theta} = E[\Theta\mid x] = \frac{x+a}{t+a+b} = \frac{t}{t+a+b}\frac{x}{t} + \frac{a+b}{t+a+b}\frac{a}{a+b}$$

它可以看作 x/t 估计与先验估计 $a/(a+b)$ 的线性组合。如 $t\gg a+b$,则 x/t 估计占支配地位;如果 $t\ll a+b$,则先验估计 $a/(a+b)$ 占支配地位。

注意到,如 t 次测试未能揭示一个故障(x= 0),则 Θ的后验分布为 B(a, t+ b),且其 pdf 为:

$$f(\theta \mid x=0) = \frac{\theta^{a-1}(1-\theta)^{t+b-1}}{B(a, t+b)}, \qquad 0 \leq \theta \leq 1.$$

此时有:

$$\hat{\theta} = \frac{a}{t+a+b} \tag{9.6.2}$$

其方差为:

$$\frac{a(t+b)}{(t+a+b)^2(t+a+b+1)} \tag{9.6.3}$$

如 a= b= 1,则 Θ的后验分布为 B(1, t+ 1),其 pdf 为

$$f(\theta \mid x=0) = (1+t)(1-\theta)^t, \qquad 0 \leq \theta \leq 1$$

于是得出 Laplace 的结果: Θ的期望值为:

$$\hat{\theta} = \frac{1}{t+2}.$$

方差为:

$$\frac{t+1}{(t+2)^2(t+3)}.$$

上述(9.6.2)式为 θ的当前贝叶斯(点)估计,而(9.6.3)式为该估计的方差。方差为 θ精度的置信度的一个测度,更小的方差对应于置信度的增加。

在开发软件的过程中,每个不同的版本都有一个对应的 θ值。每个单个的程序版本与一个固定的 θ值对应,但 θ的值是未知的。这就象在一个有着许多罐子的集类中选择某个罐子,而它们的每一个都有不同的黑球所占的比例。在测试前,我们无法知道 θ的先验值。在这种情况下, θ就可以作为一个随机变量来处理。因此,作如下先验假设, θ的总体分布为 B(a, b), a 和 b 是参数,要求它们的取值对于所有总体中的程序都能适合。

可以使用贝叶斯估计。因为输入域是有限的,所以 θ的可能值也是有限的,且贝叶斯定理可以直接应用于对 θ的 pdf 的估计。上面已讨论过,一个连续的 β分布对于由一般的贝叶斯定理获得的离散值,有着一种很强的关系。例如,当 a= b= 1,应用 β(1, 1)的 pdf ,对每个可能的 θ值进行定位,以获得这些离散值。然后,根据它们的和,对 pdf 进行规格化。θ= 0 是一个特例,因 β分布的 pdf 在零处无定义。但可以计算任意接近于零的置信区间,这对任何实际应用已足够了。期望值点估计 θ刚好可用到后验的 Θ分布已知这一事实。例如,可定义置信水平 c(如取 c= 0. 95,这是很典型的作法),于是定义了一个临界值 θ_c,使:

$$P_r\{\Theta \leq \theta_c \mid x_i\} = c.$$

求解 θ_c,有 $0 \leq \Theta \leq \theta_c$。落在其内的任何值,都有着 95% 的置信度。设 a= 1, x= 0,则可构造一个后验累积分布函数(cdf):

$$F(\theta \mid x_i) = P_r\{\Theta \leq \theta \mid x_i\} = \int_0^{\theta_c} f(\theta \mid x_i)\, d\theta$$

于是有:

$$F(\theta \mid \gamma) = 1 - (1 - \theta)^{t+b},$$

$$c = 1 - (1 - \theta)^{t+b}.$$

对于 θ, 我们有着一个 α 100% 的关于 Θ 的置信区间估计 $[0, \theta]$。

下面, 将 θ 作为随机变量 θ 的后验 β-分布的均值的"估计", 即对故障概率的估计。(9.6.2) 式中 a 和 b 的取值, 可以使用关于软件的先验信息, 这些信息会影响到对测试结果的解释。如果有意忽略所有的先验信息, 并且在关于 θ 的优先忽略的完全状态下进行计算, 则应取 a= b= 1, 这样就得出 Laplace 关于 θ 的方程, 这也体现了一种信任程度。在随机测试之前, 该软件有同等可能在 $[0, 1]$ 中取到 θ 的任何值。

下面讨论远离 a= b= 1 的情况。设某 θ 的均值为 μ 方差为 σ^2, 则为使 a 和 b 的取值能得到相同的均值与方差, 且产生一个先验 β-分布, 可取:

$$a = \frac{\mu^2(1 - \mu) - \sigma^2\mu}{\sigma^2},$$

$$b = \frac{(\mu^2(1 - \mu) - \delta\mu)(1 - \mu)}{\sigma^2\mu},$$

其中,
$$0 < \mu < 1,\ 0 < \sigma^2 < \mu(1 - \mu).$$

§ 9.7 神经网络在软件可靠性研究中的应用

因为所有现存软件可靠性模型, 都以某些理论假设作为基础, 而这些关于剩余在软件中的错误的性质、以及故障的随机行为的假设, 大多又各不相同, 所以由各个软件可靠性模型所表示的故障行为的类型, 差异很大。因此, 有证据可以证明: 这些模型的估测结果是很不一样的, 特别在测试阶段的初期, 更是如此。然而, 在早期的测试阶段, 关于软件产品的投放日期、以及所要求的附加测试努力等管理目标又要求模型有更好的估测能力。因此, 怎样正确选择适用的软件可靠性模型, 在软件可靠性估测过程中, 有着举足轻重的意义。

对于正确选择适用模型的问题, 可以有两种解决的途径: 一个途径是通过对广泛的、有代表性的故障数据集合的分析, 概括出现存软件可靠性增长模型的可应用性 (applicability); 另一个途径是通过开发出一个可适用模型的建模系统, 来针对不同的实际问题, 构造可适用的模型。但是, 为前一途径所要求的足够数据集合, 在目前尚不可能。即使有了大量的故障数据集合, 并概括出每个模型的可应用性, 可一旦面对新的故障数据集合, 仍不可能排除某个特定模型存在着某些应用方面的局限性的可能。这主要应归因于现存模型的外部参数依赖于软件开发环境这一事实。

N. Karunanithi 等人则通过上述另一途径来解决选模问题。他们将人工神经网络 (Artificial Neural Networks) 技术应用于软件可靠性估测过程, 并且他们报告称, 已获得了估测的一致性的好结果。

目前, 许多研究者都已发现, 许多软件可靠性模型都有各自不同的特点, 它们对于某

些故障数据集合,能给出好的结果,但对于另一些故障数据集合,则又不可能给出精确的估计。这一事实,也都逐渐地被模型的作者们所接受。我们将这一现象也划归到不一致性问题的范畴。之所以产生这样的问题,一方面,由于模型对软件系统的故障行为所作的假设固定不变,但软件系统却受到各种随机因素的综合作用,其结果就是软件的故障行为千变万化。用某一固定的故障模式去套形形色色的真实故障行为,其结果显然是不适配的。另一方面,在软件的测试期间,根据测试所反映的结果,人们肯定要不断地修订测试计划,改变测试策略,由此,收集到的故障数据所反映出来的软件系统的故障行为模式,必然也不会一成不变。由于这两方面的原因,必然导致不一致性的产生。而不一致性的问题,又严重地阻碍了软件可靠性技术的推广。

于是,现在有不少的研究者正在围绕不一致性问题展开研究工作。Littlewood 提出的综合考虑现存软件可靠性模型的不同优点,以产生一个适配的模型,以及我们正在从事的软件可靠性估测专家系统的研究与开发,都是从这一角度出发的。N. Karunanithi 等人把人工神经网络应用于软件可靠性研究,也是一种十分有意义的尝试。下面我们分别介绍 Karunanithi 和 T.M.Khoshgoftaar 等人在这方面的研究工作。

1. N.Karunanithi 和 Y.K.Malaiya 等人的工作

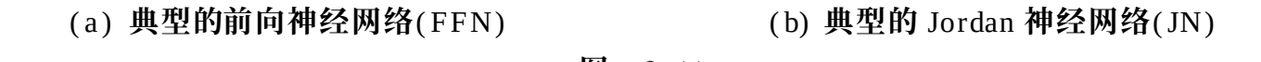
神经网络是基于现代对生物神经系统的认识而提出的一种高度抽象化的数学模型。一个人工神经网络至少应包括:① 大量被称为神经元(neurons)的简单处理单元,它们对给予的输入信息作局部的计算,并产生输出信息。② 大量加权的、神经元之间的联接,它们对网络的知识进行编码。③ 同步或异步的处理模式。④ 学习算法,它可以对网络的内部知识表示产生一种自动的开发方法,比如自动构造对应于特定问题的神经网络系统。

存在着各种各样的神经网络模型和学习算法。在他们将神经网络应用于软件可靠性估测的研究中,用得最多的是前向神经网络(FeedForward neural Networks —— 简称为 FFN),和 Jordan 的半递归神经网络(Jordan's Semi-recurrent neural Network —— 简称为 JN)。

一个典型的 FFN 由三种不同类型的层次组成:一个由神经元组成的输入层(input layer),它负责接收经过事先编码的、来自系统外部的输入信息;一个由神经元组成的输出层(output layer),它负责将来自所有输入层上的神经元和现有前面所有隐蔽层(hidden layer)上的神经元的加权信息,传送到系统的外部;以及一个或多个由神经元组成的隐蔽层,这些隐蔽层上的神经元不直接对来自外部的信息进行计算,它们的所有神经元负责接收来自所有输入层上的神经元和现有前面所有隐蔽层上的神经元的加权输入信息,并将这些信息转换成可作为输入信息的激活值(activation value),并传递到属于下一层上的神经元。对 FFN 模型的重要限制之一,就是它们的联接只能按向前的方向传播激活(activation)。JN 模型和 FFN 模型主要的区别在于:在输入层上的每个神经元都增加了一个输入神经元(输入单元)。在训练期间,它起内部输入单元的作用;在估测期间,它起与输出单元的反馈联接作用。它们的结构见图 9.11。

在输入层上的处理单元,接收来自外部世界的输入信息(如:累积执行时间),并且直接就将信息复制到它们的输出端。隐蔽层有一个或多个处理单元,它们接收来自所有输入

单元和所有前一级隐蔽单元(hidden units)的加权输入信息。多层的神经网络的处理能力,主要集中在隐蔽单元,它们能将外部输入信息转换成网络的内部表示。输出层的处理单元接收来自输入单元和所有隐蔽单元的加权输入信息,并将它们映射成输出信息(如:一个输出层处理单元将传递给它的输入信息,映射成一个累积软件错误)。



(a) 典型的前向神经网络(FFN)　　(b) 典型的 Jordan 神经网络(JN)

图　9.11

关于人工神经网络的构成,典型的作法是将同类型的处理单元用于组成隐蔽层和输出层。图 9.12 示出一典型的处理单元,以及它怎样计算作为输入的加权和(weighted sum)的和值(sum,以下一律用英文单词'sum'表示)。该处理单元应用它的激活函数(the activation function)将 sum 转换成一输出值。Karunanithi 等人使用具有逻辑激活函数(Logistic activation function)的 S 形单元(sigmoidal unit)来构成人工神经网络的输出层。S 形单元的输出(Output)由下式给出:

图 9.12　典型的处理单元

$$\text{Output} = \frac{1}{1 + e^{-\text{sum}}}$$

其中 sum 是来自隐蔽层的所有单元,包括输入层和一个有倾向性单元的加权输入信息。有倾向性单元(bias unit)是一个特别的输入单元,它的输出永远被固定为 1.0,它的作用是对所有的隐蔽单元和输出单元提供一个附加的输入。图 9.13 表示一个 S 形单元的激活函数。注意,S 形单元的输出被限制在区间[0,1]之中。在实际应用 S 形单元于输出层时,就必须将输出信息(如:累积软件错误数),按适当的比例,使用一个已知的极大值,将它的输出变换到[0,1]的范围之中。但在实际中,关于累积错误个数,并不存在这样一个极大值,那就应该预先设定一个合理的值。图 9.14 表示对这一问题的一种可能的解决方法。代替 S 形单元的使用,可以使用一线性单元。线性单元的输出(output)由下式给出:

$$\text{Output} = \begin{cases} \text{sum}, & \text{如果 sum} > 0, \\ 0 & \text{如果 sum} \leq 0. \end{cases}$$

这样一来,网络就可以给出任一个正值作为它的输出。但是,使用线性函数的一个困难在

图 9.13　S 形单元的输出特性　　图 9.14　线性单元的输出特性

于: 当 sum≤0 时, 它不能得出一个连续的导数。而为了保证学习算法的正确工作, 必须要求导数是连续的。因而一个简单的替代方法就是当 sum< 0 时, 使该线性函数成为一个(1/sum)类型的函数。这要求在 sum< 0 时, 线性单元的输出应很迅速地下降到零。当条件 sum≤0 成立时, 对输出附加一个可以忽略不计的值时, 这种类型的函数能够对于该激活函数的所有部分提供一个行为优良的、非零的导数。实际的激活函数可以由下式给出:

$$\text{Output} = \begin{cases} \text{sum}, & \text{如果 sum} > 0, \\ 1/(a - b\,\text{¡¤sum}), & \text{如果 sum} \leq 0. \end{cases}$$

其中, a 和 b 都是常数。这一经过修改以后的函数的导数为:

$$\frac{\text{dOutput}}{\text{dsum}} = \begin{cases} 1, & \text{如果 sum} > 0, \\ b/(a - b\,\text{¡¤sum})^2, & \text{如果 sum} \leq 0. \end{cases}$$

如果令 $b = a^2$, 则上面的导数在 sum= 0 时, 将减至 1.0。但关于 a, 还必需选择一个合理的值, 以保证在 sum< 0 时, 输出很快下降到零。Karunanithi 等人取: $a = 1.0 \times 10^3$, $b = 1.0 \times 10^6$。

FFN 的正常使用, 应先从对系统的训练开始。首先, 可以对网络的权使用一个随机值集合进行初始化。通常, 训练阶段要涉及一系列的迭代(也称"纪元"—epochs), 在训练的每一纪元期间, 网络由例输入-输出对(example input-output pair)的集合表示, 并且计算训练的与实际的输出之间的和平方误差(sum square error), 而后将误差反馈以调整网络权, 目的在于使误差越来越小。对权的调整要根据它们对误差所施加的影响大小来进行。当和平方误差降至一特定的容许限以下时, 训练过程就完成了。

处于监控之下并经过训练的 FFN 的许多实际应用, 都可以表示成一个映射, NN: $S^n \rightarrow R^m$。其中, 对于给定的应用问题, S^n 是一个 n 维的输入刺激空间, R^m 是相对应的 m 维输出响应空间, 一般映射的完成是通过使用一多层的网络达到的。训练过程可看作一操作, $T: I_k$ 砸 O_k, 其中 $(I_k, O_k) = \{(i, 0) \mid i \in S^n, 0 \in R^m\}$, 是从 (S^n, R^m) 中由采样得出的 k 个刺激—响应对的子集合。函数 T 是对 NN 的一个逼近。

估测问题可以抽象为一个映射 $P: I_l$ 砸 O, 其中 I_l 表示 l 个当前刺激样本的序列, O 表示对应于将来某一时刻的估计的输出。这里有: $I_k = I_l$, 且表示一 k= l 个相邻的刺激的序列, 而不是表示一个随机采样的集合。一旦使用一预先确定的误差容许限来训练该网络, 则在训练完成之后, 就可以使该网络在给以 $i_{k+d} \in S^n$ 作为刺激输入时, 估计输出的情况。输入 i_{k+d} 对应于在第 k 个时刻之后附加 d 个相连接的随机长度的时间区间以后的将

来的某个刺激。如果 d= 1, 则这种估计称为下一步的估计(the next-step prediction); 如果 d= n(≥2), 则称为长期估计。

对于一个给定的实际应用问题, 怎样构造一个适当的结构, 一直是当前人工神经网络研究的重要问题之一。估测的质量、以及为完成模拟所需要的资源, 都受到该结构的制约。最近, Fahlman 提出了一个动态建造多层神经网络的方法: Fahlman 的级联相关学习算法(Fahlman's Cascaded-Correlation Learning Algorithm), 用它构成的结构称为 Fahlman 级联相关学习结构。(读者可参阅: S. E. Fahlman and C. Lebiere, "The Cascaded-Correlation Learning Architecture", School of Computer Science, Carnegie Mellon University, Pittsburg, Tech. Rep. CMU-CS-90-100, Feb, 1990.)

在输入和输出层所用到的处理单元的数目, 要根据对输入信息和输出变量进行编码时, 实际使用的处理单元数而定。关于隐蔽层的处理单元, 则可以由 Fahlman 的级联相关学习算法进行动态的连接, 级联相关学习结构可以动态地将隐蔽层的结点加到已连好的网络上去, 所需的结点数则动态地依赖于数据集合的大小。在训练一开始, 训练总体的大小总是很小的, 因此, 网络的学习只需要很少的隐蔽单元就可以进行。随着训练总体大小逐渐地增加, Fahlman 级联相关学习算法就开始增加更多的处理部件到已连接好的网络上。正好与人们所期望的一样, 系统功能的实现, 随着时间的推移, 将得到不断改善。

在使用一个人工神经网络之前, 要用适合于它的方式对具体的应用问题进行编码。因为 FFN 中的状态变量限制在区间[0, 1]的范围之内, 所以应用问题的输入/输出变量, 也必须变换到这一范围中。对于那些输入/输出变量本身即为二进制的应用问题, 编码任务很好解决。但对于那些输入/输出变量的数值变动范围很大的问题, 如果不采取特别的编码技术, 就可能得出不准确的处理结果, 甚至使人工神经网络无法区分不同的输出值。

可以使用二进制方案对一非负整数 N 进行编码: 用 $\log_2(N+1)$ 位二进制数即可表示 N。如果用 m 位二进制数, 则可表示: $2^0, 2^1, 2^2, \cdots, 2^{m-1}$ 这些数。在数字空间与神经元状态空间之间, 存在着一一对应的关系。但采用这种直接二进制编码的弊端之一即在于: 如果在较高的数位上发生故障, 即使是一次单个的故障, 也会产生大的误差。这种情况极象在任何两种相关的模式之间, 无论何时都会产生一个大的 Hamming 距离(Hamming distance)的情况。

N. Karunanithi 等人采用 Gray 编码表示(Gray coded representation —— 关于它以及 Hamming 距离, 读者可参阅 R. W. Hamming, Coding and Information Theory, Printice-Hall, 2nd Ed., 1986.)来对具体的问题进行编码, 它很适用于输入模式是连接的数值的情形。Gray 编码表示有这样的性质: 任何两个数字式的邻近模式, 都有着确定为 1 的 Hamming 距离。由于使用的数据集合表示的是一个递增的数字值的序列, 故此在两个相衔接的值之间的 Hamming 距离, 就将以主扰动的形式出现。例如, 如果输入是 127 和 128, 它们是相邻的两个数, 则采用直接二进制编码所产生的 Hamming 距离为 8。任何接近这种大的 Hamming 距离的估计, 都会产生很大的误差。而采用 Gray 编码表示, 则可以减少由此而产生的误差。在输入不是连接的数字值时, 他们就采用标准的二进制编码技术。

不论采用什么样的编码方法, 都应保证输出层上的神经元的输出值不为零, 即为 1,

不能有任何中间的值出现。为此可采用诸如: 在 S 形激活函数中增大增益系数, 或在二次能量方程中加上一附加项等方法。仅仅在训练期间才要求这样做, 在估测期间就不必作如此的要求。在估测期间, 输出层上的每个神经元表示的是每一位的极大似然估计值, 而不是二进制数值本身。

在训练人工神经网络阶段, 使用执行时间作为输入, 而将观察到的累积错误数作为目标输出。训练集合的大小, 在从测试开始的 20% 的数据点, 到除最后一个数据点以外的范围内变化。在训练结束时, 输入一个附加的、未来的执行时间的区间段, 并考察人工神经网络对累积错误数的估计的精度。

为了将应用人工神经网络的估计结果与其它一些现有软件可靠性模型的估计结果进行比较, Y. K. Malaiya 等人提出了一个可估测性测度。它可以简单地概括如下: 设数据集合共有 m 个数据点, (t_i, μ), $i = 1, 2, \cdots, m$。其中, μ 是在第 i 个测试段结束时, 查出的累积错误数; t_i 为累积执行时间。令 μ_{tp}^{k} 表示在时刻 t_k 时的估计出的、到时刻 t_n 为止软件中的错误总数, 且有关系: $t_k < t_n \leq t_m$。利用在每个 t_k 时的估计值 μ_{tp}^{k}, 可以计算出该模型的估计误差:

$$\Delta_k = \frac{\mu_{tp}^{k} - \mu_t}{\mu_t}, \qquad k = 1, 2, \cdots, n-1.$$

于是该模型的平均估计误差则为:

$$A_{error} = \frac{1}{n-1}\sum_{k=1}^{n-1} |\Delta_i| = \frac{1}{n-1}\sum_{k=1}^{n-1} \left| \frac{\mu_{tp}^{k} - \mu_t}{\mu_t} \right|.$$

A_{error} 给出从 $t_0 (= 0)$ 到 t_{n-1} 这一段测试期间, 一个模型估计的结果到底有多好的一个量测标准。

他们为了进行对比, 使用数据 I (见: J. D. Musa, A. Iannino, and K. Okumoto, Software Reliability-Measurement, Prediction, Applications, McGraw-Hill, 1987.)、数据 II (见 N. D. Singpurwalla and R. Soyer, IEEE Trans. on SE., Vol. SE-11, No. 12, pp. 1456-1464, 1985.), 和数据 III (见: B. M. Anna-Mary, A Study of the Musa Reliability Model, M. S. Thesis, CS Dept., Univ. of Maryland, 1980.), 并选择出 5 个现有的软件可靠性模型: 对数泊松执行时间模型、L-V 模型、Crow 乘幂模型、Moranda 模型、Yamada S 形模型作为估计模型。将应用它们而分别得到的结果与人工神经网络的处理结果进行比较, 最后得出人工神经网络处理能获得优良的一致性, 特别是在测试的早期阶段的结论。分别见图 9.15—图 9.17。图中,“Neural Net”表示人工神经网络的估测结果,“log Poisson”表示对数泊松执行时间模型的估测结果,“Inverse Poly”表示 L-V 模型的估测结果,“Exponential”表示 Moranda 指数模型的估测结果,“Power”表示 Crow 乘幂模型的估测结果,“Delayed S-Shape”表示 Yamada S 形模型的估测结果。图上都是以估测结果的相对误差的百分比数来表示的。

2. T. M. Khoshgoftaar 等人的工作

Khoshgoftaar 等人在应用人工神经网络估测软件可靠性时, 使用时间不变的数据 (诸如: 软件复杂性等质量尺度), 以训练神经网络系统, 从而达到调整网络连接强度

图 9.15　人工神经网络估测结果与现有软件可靠性模型估测结果的比较(使用数据Ⅰ的情况)

图 9.16　人工神经网络估测结果与现有软件可靠性模型估测结果的比较(使用数据Ⅱ的情况)

图 9.17　人工神经网络估测结果与现有软件可靠性模型
估测结果的比较(使用数据Ⅲ的情况)

(connection strength,又称网络参数,即权)的目的。如果网络被训练得很好,而且用于训练网络的数据集合又是实际用于估测目的的真实数据,网络就会有很好的估测结果。在估测阶段,根据训练阶段产生的系统,对于每一输入模式,给定一个输出,以测试该网络系统。对于部分输入的模式,也可以采用这一方法,以得出与该部分输入模式对应的输出。他们使用的人工神经网络操作的两个阶段示于图 9.18 中。

人工神经网络学习算法允许它适应它本身的错误,也允许它从自身的错误中进行学习。每一个神经元就是一个简单的处理机,它可以对所有加权的输入求和,并且进行数学变换以产生一个输出。通常产生的数值(输出)总是传递至下一层上的神经元,它们又以同样的方式处理并产生一个输出。在他们的应用中,定义输出如下:

$$Net_j = \sum_{i=1}^{n} W_{ij} Out_i - \theta_j$$

$$Out_i = f(Net_i) = \frac{1}{1 + e^{-Net_i/T}}$$

其中,Out_i 是第 i 个神经元的输出;Net_j 是具有阈值 θ_j 的神经元 j 的加权和(weighted sum);n 是输入神经元的个数;W_{ij} 是神经元 i 与神经元 j 之间的连接强度(即权);T 是在对阶梯函数用 S 形函数逼近时,将取很小值处的温度(temperature)。训练阶段使用的是反向传递学习算法(back-propagation learning algorithm)。期望输出与实际输出之间的总误差 E,为:

图 9.18　人工神经网络操作的两个阶段

$$E = \frac{1}{2}\sum_{p}\sum_{i}(d_{p_i} - O_{p_i})^2,$$

其中 d_{p_i}是对应于第 p 次输入模式的神经元 i 的期望输出值, O_{p_i}是该神经元的实际输出值。在第 n 次迭代过程中, 对权由下列标准进行调整:

$$\Delta W_{ij}(n+1) = \eta\delta_j Out_i,$$

$$W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}(n+1),$$

其中, $W_{ij}(n+1)$是提供输入的神经元 i 和处于下一隐蔽层或输出层上的神经元 j 之间, 在纪元(epoch) n+1 时的权; η是学习率(learning rate); 对于输出层的神经元, 误差信号 δ 为:

$$\delta_j = Out_j(1 - Out_j)(d_{p_j} - O_{p_j});$$

对于隐蔽层的神经元 j, 没有特别的目标值, 误差信号 δ_j则根据所有那些直接与神经元(j)相连接的神经元(k)以及它们的权来递归决定的:

$$\delta_j = Out_j(1 - Out_j)\sum_{k}\delta_k W_{k_j}.$$

对于网络参数的调整, 分析其在权空间(weight space)中的变化趋势, 对于减少网络的训练时间, 以及提高训练过程的稳定性, 都是很重要的。为此有:

$$W_{ij}(n+1) = W_{ij}(n) + \Delta W_{ij}(n+1) + \alpha \Delta W_{ij}(n).$$

对权空间的修正,一直要迭代到 E 达到容许的限为止。

在他们的工作中,输入变量都是整数,输出变量是存在于软件模块中的总错误数,输出的值在 0 到 42 之间变化。采用编码技术,将它们变换到[0, 1]区间上去,最后还要将[0, 1]范围内的输出还原为真实的输出值。

隐蔽层的层数,以及每一隐蔽层上神经元的数目,部分地由输入和输出层决定。但它们所用神经元的数目越多,涉及的计算量就越大。但是这种情况有利的一面则在于:采用更多层次的隐蔽层结构,可以构造更高次的方程。

在对网络进行训练时,学习率可用于决定当产生的输出不等于期望的输出时,对权所作更正的程度。学习率越高,对权空间的改动就越多,但过小的学习率又将延长对网络的训练时间。因此,对学习率的大小进行适当的综合考虑,并且在不同的训练级别上考虑学习率大小的取舍,能获得较好的效果。

他们使用同一个输入数据集合,对不同的网络结构进行了试验。他们使用的输入变量有 8 种基本的尺度,而输出变量则只有程序模块中的总错误个数一种。这些试验的各种网络结构,从拥有一层隐蔽层,到拥有三层隐蔽层,而且每一隐蔽层上的神经元的数目也不相等。

当网络经过长时间的训练之后,或直到所有的样本都由网络学习过之后,人工神经网络的估测结果最好。在一段有限长的时间内,由网络学习所有的样本是不大可能的,因为对于相类似的输入模式,数据集合中仍可能含有不同的输出值,但它们又可能都是有意义的。也即是说,数据集合中可能含有远离拟合曲线的数据点。这些远离拟合曲线的数据点都有一个共同的倾向:它们都可能令神经网络和回归模型产生过大的偏差。可以预先确定一个特定的百分数,在占这一百分比的样本都被神经网络学习之后,再对软件的可靠性特性进行估测。这一特定的百分数可以根据样本的多少,或直观推断来决定。可以观察到这样一个事实:拥有一层隐蔽层的网络结构,根据性能标准(估测误差)可以很好地完成估测任务。它证明:在初始阶段,当少于 50% 的观察(程序模块)开始用于训练时,可以有较大的学习率,然后随着越来越多的观察被用于训练,可逐渐降低学习率,这样使用数据集合仍可使对网络的训练工作做得很好。这时,通过对拟合和估测质量的研究可以发现,数据集合给出非常乐观的结果。估测结果显示出:网络倾向于追踪全部数据集合的行为,且有时它的估测值高于实际值,但有时又低于它。数据集合本身就是一个随机行为的实例。以在训练网络时使用的数据中它能辨识的模式为基础,只要给定输入变量,网络估测的就是究竟会有多少个错误出现。因为训练数据仅仅是网络从外部世界接收的信息,它们就是网络推出结论的基础数据。

他们将人工神经网络技术应用于软件质量的估计,并且将结果与线性回归分析的结果进行比较。用于分析的数据是从-Ada 开发环境中开发的军用数据的命令和控制连接通讯系统(Command and Control of a Military Data Link Communication System—CCCS)获得的,组成这一大型 Ada 系统的模块共计 282 个。关于标准相关变量——每个模块中的错误数的数据,是根据问题更改报告(Problem Change Reports—PCR)收集的,对于每一次更改都应有一份 PCR,它们之间存在一个一一对应的关系。

在测定软件系统的某些质量指标时,经常用到线性回归分析技术。其中,许多软件复杂性尺度都是被用作独立变量,而相关变量只有一个,那就是软件中的错误个数。通过对独立变量的集合的一个子集合的选取,来进行线性回归分析,并由此尽可能多地解释相关变量的变化。这些独立变量的系数,是通过拟合样本数据的这些变量,利用最小二乘法估计出来的。已有一系列的方法可以用来选择合适的子集合,而又保证不引入附加的变化(噪声)。关于估计误差,可以利用平均相对误差来表示,同时它也被用作一个评价与复杂性尺度有关的模型的质量的一个测度,并据此来辨认哪些软件是错误多发性的。只要一个模型被用来进行拟合,就可以计算它的平均相对误差。

他们用于分析的软件复杂性指标有:

- 唯一的运算符数(Unique operator count)- η_1
- 唯一的运算对象数(Unique operand count)- η_2
- 总运算符数(Total operator count)- N_1
- 总运算对象数(Total operand count)- N_2
- Halstead 估计权序长度公式: $N = \eta_1\log_2\eta_1 + \eta_2\log_2\eta_2$
- Halstead 努力尺度—E
- 程序容量: $V = N\log_2(\eta_1 + \eta_2)$
- McCabe 循环复杂度—VG1
- 扩展的循环复杂度—VG2
- 过程调用次数—PROCS
- 注释行数—COM
- 空白行数—BLNK
- 代码行数—LOC
- 可执行的代码行数—ELOC

他们把对模型的评价分为两个部分来进行:模型拟合质量评价与模型估测质量评价。对于 CCCS 中的 282 个模块的数据,以随机方式选出 188 个模块的数据,用于进行模型拟合质量评价;剩下的 94 个模块的数据,用于进行模型估测质量的评价。评价结果分别列于表 9.2 和表 9.3 中。从表 9.2、表 9.3 中可以看出,人工神经网络分析技术的结果优于线性回归分析技术的结果,并且有着较线性回归分析技术更小的小标准误差(small standard error)。

表 9.2 CCCS 的模型拟合质量评价

建模技术	误差	
	平均相对误差	小标准误差
回归分析	0.6249	0.8167
人工神经网络	0.3333	0.2330

表 9.3　CCCS 的模型估测质量评价

建模方法	误差	
	平均相对误差	小标准误差
回归分析	0.5877	0.6248
人工神经网络	0.3980	0.2786

参 考 文 献

为使读者以后能进行更深入的了解和研究,下面列出有关软件可靠性模型及其应用的参考文献,所有文章均为已正式发表的英文论文。除一些有代表性的经典文献外,其它均为近年来发表的文章。我们在尽可能不重复的条件下,根据文章的主要内容进行了分类,以便于读者参考。

A: 综合性和经典性文献

[A1] R. E. Barlow and N. D. Singpurwalla, Assessing the reliability of computer software and computer networks: an opportunity for partnership with computer scientists, The Amer. Statist., 39 (2) 88-94, 1985.

[A2] A. Bendell and P. Mellor, Software reliability: State of the art report, Pergamon Infotech Ltd., pp. 267-442, 1986.

[A3] B. Bergman and M. Xie, On software reliability modelling, Proc. International Research Conf. on Reliability, Missouri, USA, May 17-19, 1988.

[A4] J. P. Cavano, Software reliability measurement: Prediction, Estimation, and Assessment, J. Systems and Software, 4, 269-275, 1984.

[A5] C. J. Dale, Date requirements for software reliability models, In Software Reliability: Achievement and Assessment, Ed. B. Littlewood, Blackwell, Oxford, pp. 144-153, 1987.

[A6] D. Drake and D. E. Wolting, Reliability theory applied to software testing, Hewlett-Parkard Journal, 4, 35-39, 1987.

[A7] A. L. Goel, Software reliability models: assumptions, limitations, and applicability, IEEE Trans. Software Eng., SE-11(12). 1411-1423, 1985.

[A8] L. N. Harris, Software reliability modelling-prospects and perspective, In Software Reliability: State of the Art Report, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 105-118, 1986.

[A9] M. H. Halstead, Elements of software science, Elsevier Press, New York, 1978.

[A10] V. I. Lipaev, Software reliability (a review of the concepts), Automation and Remote Control, 47(10), 1313-1335, 1987.

[A11] B. Littlewood and J. L. Verrall, A Bayesian reliability growth model for computer software, Applied Statistics, 22, 332-346, 1973.

[A12] B. Littlewood, How to measure software reliability and how not to, IEEE Trans. Rel., R-28 (2), 103-110, 1979.

[A13] B. Littlewood (editor), Software reliability: Achievement and assessment, Blackwell Scientific Publ., Oxford, 1987.

[A14] P. Mellor, Software reliability data collection: problems and standards, In Software Reliability:

State of the Art Report, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 165-181, 1986.

[A15] P. Mellor, Software reliability modelling: the state of the art, Information and Software Technology, 29(2), 81-98, 1987.

[A16] J.D. Musa, A theory of software reliability and its application, IEEE Trans. Software Eng., SE-1(3), 312-327, 1975.

[A17] J.D. Musa, Validity of the execution time theory of software reliability, IEEE Trans. Rel., R-28(3), 181-191, 1979.

[A18] J.D. Musa, Software reliability measurement, J. Systems and Software, 1, 223-241, 1980.

[A19] J. D. Musa, A. Iannino and K. Okumoto, Software reliability: measurement, prediction, application, McGraw-Hill, New York, 1987.

[A20] E. Nelson, Estimating software reliability from test data, Microel. Rel., 17, 67-74, 1978.

[A21] C. V. Ramamoorthy and F. B. Bastani, Software reliability-status and perspectives, IEEE Trans. Software Eng., SE-8(4), 354-371, 1982.

[A22] J. G. Shanthikumar, A general software reliability model for performance prediction, Microel. Rel., 21(5), 671-682, 1981.

[A23] M.L. Shooman, Probabilistic models for software reliability prediction, In Statistical Computer Performance Evaluation, Ed. Freiberger, W., Academic Press, New York, 485-502, 1972.

[A24] M.L. Shooman, Software reliability: a historical perspective, IEEE Trans. Rel., R-33, 48-55, 1984.

[A25] S. Yamada and S. Osaki, Reliability growth models for hardware and software systems based on nonhomogeneous Poisson processes: a survey, Microel. Rel., 23(1), 91-112, 1983.

[A26] S. Yamada and S. Osaki, An error detection rate theory for software reliability growth models, Trans. IECE of Japan, E68(5), 292-296, 1985.

[A27] S. Yamada and S. Osaki, Software reliability growth modelling: Models and applications, IEEE Trans. Software Eng., SE-11(12), 1431-1437, 1985.

[A28] M. Xie, SOFTWARE RELIABILITY MODELLING, World Scientific Publishing Co. Pre. Ltd., Singapore, 1991.

B: 关于非齐次泊松过程模型

[B1] W. K. Ehrlich and T. J. Emerson, Modelling software failures and reliability growth during system testing, Proc. 9th Int. Conf. on Software Eng., Chicago, USA, pp. 72-82, 1987.

[B2] A.L. Goel and K. Okumoto, Time dependent error detection rate model for software reliability and other performance measures, IEEE Trans. Rel., R-28(3), 206-211, 1979.

[B3] J. Kyparisis and N. D. Singpurwalla, Bayesian inference for the Weibull process with applications to assessing software reliability growth and predicting software failures, In Computer Science and Statistics, (16th Symposium on the Interface, Atlanta, G.A. 1984), North-Holland, pp. 57-64, 1985.

[B4] P.N. Misra, Software reliability analysis, IBM Systems Journal, 22(3), 262-270, 1983.

[B5] J. D. Musa and K. Okumoto, A logarithmic Poisson execution time model for software reliability measurement, Proc. 7th Int. Conf. on Software Eng., Orlando, pp. 230-238, 1984.

[B6] J. D. Musa and K. Okumoto, Application of basic and logarithmic Poisson Execution time models in software reliability, In Software System Design Methods, Ed. J. K. Skwirzynski, NATO ASI Series, Vol. F22, Springer, Berlin, pp. 275-298, 1986.

[B7] M. Ohba, Software reliability analysis models, IBM J. Res. Develop., 28(4), 428-443, 1984.

[B8] I. P. Schagen, A new model for software failure, Reliability Engineering, 18, 205-221, 1987.

[B9] S. Yamada, M. Ohba and S. Osaki, s-Shaped reliability growth modeling for software error detection, IEEE Trans. Rel., R-32(5), 475-478, 1983.

[B10] S. Yamada, H. Narihisa and H. Ohtera, Software reliability analysis based on nonhomogeneous error detection rate model, Microel. Rel., 24(5), 915-920, 1984.

[B11] S. Yamada, M. Ohba and S. Osaki, s-Shaped software reliability growth models and their applications, IEEE Trans. Rel., R-33(4), 289-292, 1984.

[B12] S. Yamada, S. Osaki and H. Narihisa, A software reliability growth model with two types of errors, R. A. I. R. O., 19(1), 87-104, 1985.

[B13] S. Yamada, H. Narihisa and H. Ohtera, Nonhomogeneous software errors detection rate model: data analyses and applications, R. A. I. R. O., 20(1), 51-60, 1986.

[B14] 徐仁佐, 刘莲君, 潘志宏, 熊忠伟, NHPP 模型拟合质量的改进, 自然科学进展——国家重点实验室通讯, 535-542, 第 1 卷第六期, 1991。

[B15] 徐仁佐, 吴新玲, NHPP 模型参数调整与 EM 算法, 计算机学报, 第 15 卷, 第 5 期, 1992. pp. 388-396。

[B16] Ming Zhao, Min Xie, NHPP Software Reliability Growth Models and Their Application, Proceedings of the First Beijing International Conference on Reliability, Maintainability and Safety (BICRMS'92), Beijing, China, October 12-15, 1992. pp. 497-509.

C: 关于马尔可夫过程模型

[C1] S. Bittanti, R. Scattolini and P. Bolzern, Parameter identification for software reliability growth models, Proc. 8th IFAC Symp. on Identification and System Parameter Estimation, Aug. 27-31, Beijing, China, 1988.

[C2] C. T. Gray, A framework for modelling software reliability, In Software Reliability: State of the Art Report, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 81-94, 1986.

[C3] T. F. Ho, W. C. Chan and C. G. Chung, A Module-structure software reliability model, Proc. 11th IASTED Conf. on Reliability and Quality Control, June 20-22, Lugano, Switzerland, 1989.

[C4] Z. Jelinski and P. B. Moranda, Software reliability research, In Statistical Computer Performance Evaluation, Ed. Freiberger, W., Academic Press, New York, 465-484, 1972.

[C5] W. Kremer, Birth-death and bug Counting, IEEE Trans. Rel., R-32(1), 37-47, 1983.

[C6] J. C. Laprie, Dependability evaluation of software systems in operation, IEEE Trans. Software Eng., SE-10(6), 1984.

[C7] G. J. Schick and R. W. Wolverton, An analysis of computing software reliability models, IEEE Trans. Software Eng., SE-4(2), 104-120, 1978.

[C8] F. W. Scholz, Software reliability modelling and analysis, IEEE Trans. Software Eng., SE-12(1), 25-31, 1986.

[C9] J. G. Shanthikumar, A general software reliability model for performance prediction, Microel.

Rel., 21(5), 671-682, 1981.

[C10] J. G. Shanthikumar and U. Sumita, A software reliability model with multiple-error introduction and removal, IEEE Trans. Rel., R-35(4), 1986.

[C11] T. Stalhane and B. H. Lindquist, A General Markov model for usage dependent software reliability, RELIABILITY'89, paper 5Ba/2, 1989.

[C12] U. Sumita and Y. Masuda, Analysis of software availability/reliability under the influence of hardware failures, IEEE Trans. Software Eng., SE-12(1), 32-41, 1986.

[C13] M. Xie and B. Bergman, On modelling reliability growth for software, Proc. 8th IFAC Symp. on Identification and System Parameter Estimation, Aug. 27-31, Beijing, China, 1988.

D: 关于统计数据的分析方法

[D1] T. Bendell, The use of exploratory data analysis techniques for software reliability assessment and prediction, In Software System Design Methods, Ed. J. K. Skwirzynski, NATO ASI Series, Vol. F22, Springer, Berlin, pp. 337-351, 1986.

[D2] L. H. Crow, Failure Patterns and reliability growth potential for software systems, In Sfotware System Design Methods, Ed. J. K. Skwirzynski, NATO ASI Series Vol. F22, Springer, Berlin, pp. 354-363, 1986.

[D3] L. H. Crow and N. D. Singpurwalla, An empirically developed Fourier series model for describing software failures, IEEE Trans. Rel., R-33(2), 176-183, 1984.

[D4] N. Davies, J. M. Marriott, D. W. Wightman and A. Bendell, The Musa data revisited: Alternative methods and structure in software reliability modelling and analysis, In Achieving Safety and Reliability with Computer Systems, Porc. SARSS'87, Ed. B. K. Daniels, Elservier Applied Science, London, pp. 118-130, 1987.

[D5] M. Drury, E. V. Walker, D. W. Wightman and A. Bendell, Proportional hazards modelling in the analysis of computer systems reliability, RELIABILITY'87, paper 5B/1, 1987.

[D6] H. S. Koch and P. J. C. Spreij, Software reliability as an application of martingale & filtering theory, IEEE Trans. Rel., R-32(4), 342-345, 1983.

[D7] D. R. Miller, Exponential order statistic models of software reliability growth, IEEE Trans. Software Eng., SE-12(1), 12-24, 1986.

[D8] S. M. Ross, Statistical estimation of software reliability, IEEE Trans. Software Eng., SE-11(5), 479-483, 1985.

[D9] N. D. Singpurwalla and R. Soyer, Assessing (software) reliability growth using a random coefficient autoregressive process and its ramifications, IEEE Trans. Software Eng., SE-11(12), 1456-1464, 1985.

[D10] R. Soyer, Applications of time series models to software reliability analysis, In State of the Art Report: Software Reliability, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 229-242, 1986.

[D11] P. Spreij, Parameter estimation for a specific software reliability model, IEEE Trans. Rel., R-34(4), 323-328, 1985.

[D12] L. A. Walls and A. Bendell, An exploratory approach to software reliability measurement, In State of the Art Report: software Reliability, Eds. A. Bendell and P Mellor, Pengamon Infotech

Ltd., 1986, pp. 209-227, 1986.

[D13] D. W. Wightman and A. Bendell, Proportional hazards modelling of software failure data, In State of the Art Report: Software Reliability, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., 1986, pp. 229-242, 1986.

[D14] D. W. Wightman and A. Bendell, The application of proportional hazards modelling, Reliability Engineering, 15, 29-53, 1986.

[D15] A. P. Dempster, N. M. Laird, and D. B. Rubin, Maximum Likelihood from Incomplete Data via EM Algorithm, (with discussion). J. Roy. Statist. Soc. Ser., B-39, 1977.

[D16] R. Chillarge, Wei-Lun Kao, and R. G. Condit, Defect type and its impact on the growth curve, Proc. 11-th ICSE, May 13-17, 1991. Austin, Texas, USA. 226-237.

E: 关于其它模型

[E1] B. G. Anderson, The role for entropy and information in software reliability, In Software Reliability: Achievement and Assessment, Ed. B. Littlewood, Blackwell, Oxford, pp. 172-191, 1987.

[E2] F. B. Bastani and C. V. Ramamoorthy, Input-domain-based models for estimating the correctness of process control programs, In Proc. of the Int. School of Physics, Eds. A Serra and R. E. Barlow, North-Holland, Amsterdam, pp. 321-378, 1986.

[E3] D. E. Brown, A method for obtaining software reliability measures during development, IEEE Trans. Rel., R-36(5), 573-580, 1987.

[E4] K. K. Govil, Incorporation of execution time concept in several software reliability models, Reliability Engineering, 7, 235-249, 1984.

[E5] T. F. Ho, W. C. Chan and C. G. Chung, An information theoretic approach to software reliability modelling, Proc. 11th IASTED Conf. on Reliability and Quality Control, June 20-22, Lugano, Switzerland, 1989.

[E6] X. Z. Huang, The hypergeometric distribution model for predicting the reliability of softwares, Microel. Rel., 24(1), 11-20, 1984.

[E7] P. Kubat, Assessing reliability of modular software, Operation Research Letters, 8, 35-41, 1989.

[E8] D. R. Miller and A. Sofer, A non-parametric approach to software reliability, using complete monotonicity, In Software Reliability: State of the Art Report, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 182-195, 1986.

[E9] M. Ohba, SPQL: Improvement of error seeding method, In Reliability Theory and Applications, Eds. S. Osaki and J. H. Cao, World Scientific, Singapore, pp. 294-233, 1987.

[E10] S. Osaki, Ohshimo, H. and S. Fukumoto, Effect of software maintenance policies for a hardware-software system, Int. J. Systems Sci., 20(2), 331-338, 1989.

[E11] A. L. Roca, A method for microprocessor software reliability prediction, IEEE Trans. Rel., R-37(1), 1988.

[E12] H. Sandoh and S. Fujii, Reliability growth analysis for discrete type softwares by quansi-error seeding, In Reliability Theory and Applications, Eds. S. Osaki and J. H. Cao, World Scientific, Singapore, pp. 319-327, 1987.

[E13] U. Sumita and J. G. Shanthikumar, A software reliability model with multiple-error introduction

& removal, IEEE Trans. Rel., R-35(4), 459-462, 1986.

[E14] Y. Tohma, K. Tokunaga, S. Nagase and Y. Murata, Structural approach to the estimation of the number of residual software faults baesd on the hypergeometric distribution, IEEE Trans. Software Eng., SE-15(3), 345-362, 1989.

[E15] S. T. Weiss and E. J. Weyuker, An extended domain-based model of software reliability, IEEE Trans. Software Eng., SE-14(10), 1512-1524, 1988.

[E16] C. Wohlin, Some new aspects on software reliability model comparisons, Proc. Second IEE/BCS Conference: Software Eng. 88, Liverpool, UK, 11-15 July 1988. 38-42, 1988.

[E17] M. Xie, A shock model for software failures, Microel. Rel., 27(4), 717-724, 1987.

[E18] S. Yamada and S. Osaki, Discrete software reliability growth models, applied Stochastic Models and Data Analysis, 1, 65-77, 1985.

[E19] S. Yamada and S. Osaki, A generalized discrete software reliability growth models and its application, In Reliability Theory and Applications, Eds. S. Osaki and J. H. Cao, World Scientific, Singapore, pp. 411-421, 1987.

[E20] R. Jacoby, Y. Tohma, Parameter value computation by least square method and software availability and reliability at service-operation by the hyper-geometric distribution software reliability growth model, Proc. 11-th ICSE, May 13-17, 1991. Austin, Texas, USA. 226-237.

[E21] Raymond Jacoby, Yoshihiro Tohma, Parameter Value Computation by Least Square Method and Evaluation of Software Availability and Reliability at Service-Operation by the Hyper-Geometric Distribution Software Reliability Growth Model (HGDM), Proceedings of the 13th ICSE, Austin, Texas, U.S.A. May 13-16, 1991. pp. 226-237.

[E22] Yoshihiro Tohma, Hisashi Yamano, M. Ohba and Raymond Jacoby, The Estimation of Parameters of the Hyper-geometric Distribution and Its Application to the Software Reliability Growth Model, IEEE Trans. on Software Engineering, Vol. 17, No. 5, May, 1991, pp. 483-489.

F: 关于贝叶斯分析方法

[F1] R. E. Barlow and N. D. Singpurwalla, Assessing the reliability of computer software and computer networks: an opportunity for partnership with computer scientists, The Amer. Statist., 39(2) 88-94, 1985.

[F2] V. M. Catuneanu and A. N. Mihalache, Improving the accuracy of the Littlewood-Verrall model, IEEE Trans. Rel., R-34(5), 418-421, 1985.

[F3] W. S. Jewell, Bayesian extensions to a basic model of software reliability, IEEE Trans. Software Eng., SE-11(12), 1465-1471, 1985.

[F4] H. Joe and N. Reid, On the software reliability models of Jelinski-Moranda and Littlewood, IEEE Trans. Rel., R-34, 216-218, 1985.

[F5] J. Kyparisis and N. D. Singpurwalla, Bayesian inference for the Weibull process with applications to assessing software reliability growth and predicting software failures, In Computer Science and Statistics, (16th Symposium on the Interface, Atlanta, G. A. 1984), North-Holland, pp. 57-64, 1985.

[F6] N. Langberg and N. D. Singpurwalla, A unification of some software reliability models, SIAM J. Sci. Statist. Comput., 6, 781-790, 1985.

[F7] B. Littlewood and J. L. Verrall, A Bayesian reliability growth model for computer software, Applied Statistics, 22, 332-346, 1973.

[F8] B. Littlewood and A. Sofer, A Bayesian modification to the Jelinski-Moranda software reliability growth model, Software Engineering Journal, 3, 30-41, 1987.

[F9] G. Liu, A Bayesian assessing method of software reliability growth, In Reliability Theory and Applications, Eds. S. Osaki and J. H. Cao, World Scientific, Singapore, pp. 237-244, 1987.

[F10] T. A. Mazzuchi and R. Soyer, A Bayes empirical Bayes model for software reliability, IEEE Trans. Rel., R-37(2), 248-254, 1988.

[F11] R. J. Meinhold and N. D. Singpurwalla, Bayesian analysis of a commonly used model for describing software failures, The Statistician, 32, 168-173, 1983.

[F12] A. E. Raftery, Analysis of a simple debugging model. Appl. Statist., 37(1), 12-22, 1988.

[F13] W. E. Thompson and P. O. Chelson, On the specification and testing of software reliability, 1980 Proc. Ann. Rel. & Maintain. Symp., pp. 379-383, 1980.

[F14] D. E. Wright and C. E. Hazelhurst, Estimation and prediction for a simple software reliability model, The Statistician, 37, 319-325, 1988.

[F15] H. Martz, R. Waller, Bayesian Reliability Analysis, John Wiley, 1982.

[F16] J. J. Deely, D. V. Lindley, "Bayes Empirical Bayes", J. American Statistical Association, Vol. 76, 1981, pp. 833-841.

G: 关于最佳投放策略

[G1] D. S. Bai and W. Y. Yun, Optimum number of errors corrected before releasing a software system, IEEE Trans. Rel., R-37(1), 1988.

[G2] P. A. Caspi and E. F. Kouka, Stopping rules for a debugging process based on different software reliability models, Proc. Int. Conf. on Fault-Tolerant Computing, pp. 114-119, 1984.

[G3] V. M. Catuneanu; C. Moldovan; I. Popentiu and M. Gheorghiu, Optimal software release policies using SLUMT, Microel. Rel., 28(4), 547-549, 1988.

[G4] P. A. Currit, M. Dyer and H. D. Mills, Certifying the reliability of software, IEEE Trans. Software Eng., SE-12(1), 3-11, 1986.

[G5] E. H. Forman and N. D. Singpurwalla, An empirical stopping rule for debugging and testing computer software, J. Amer. Statist. Assoc., 72, 750-757, 1977.

[G6] R. Kenett and M. Pollak, A Semi-parametric approach to testing for reliability growth, with application to software systems, IEEE Trans. Rel., R-35(3), 304-310, 1986.

[G7] E. H. Forman and N. D. Singpurwalla, Optimum time intervals for testing hypotheses on computer software errors, IEEE Trans. Rel., R-28(3), 250-253, 1979.

[G8] H. S. Koch and P. Kubat, Optimum release time of computer software, IEEE Trans. Software Eng., SE-9(3), 323-327, 1983.

[G9] P. Kubat, Assessing reliability of modular software, Operation Research Letters, 8, 35-41, 1989.

[G10] J. D. Musa and A. F. Ackerman, Quantifying software validation: When to stop testing? IEEE Software, 3, 19-27, 1989.

[G11] K. Okumoto and A. L. Goel, Optimum release time for software systems based on reliability and cost criteria, J. Systems and Software, 1, 315-318, 1980.

[G12] A. E. Raftery, Analysis of a simple debugging model, Appl. Statist., 37(1), 12-22, 1988.

[G13] S. M. Ross, Software reliability: The stopping rule problem, IEEE Trans. Software Eng., SE-11 (12), 1472-1476, 1985.

[G14] J. G. Shanthikumar and S. Tufekci, Application of a software reliability model to decide software release time, Microel. Rel., 23(1), 41-59, 1983.

[G15] M. Xie, A generalization of the JM-model, RELIABILITY'89, Paper 5Ba3, June 14-16, 1989.

[G16] S. Yamada, H. Narihisa and S. Osaki, Optimum release policies for a software system with a scheduled software delivery time, Int. J. Systems Sci., 15(8), 905-914, 1984.

[G17] S. Yamada and S. Osaki, Cost-reliability optimum release policies for software systems, IEEE Trans. Rel., R-34(5), 422-424, 1985.

[G18] S. Yamada and S. Osaki, Optimal software release policies for a non-homogeneous software error detection rate model, Microel. Rel., 26(4), 691-702, 1986.

[G19] S. Yamada and S. Osaki, Optimal software release policies with simultaneous cost and reliability requirements, European J. Oper. Research, 31, 46-51, 1987.

[G20] Y. Masuda, N. Miyawaki, U. Sumita and S. Yokoyama, A Statistical Approach for Determining Release Time of Software System with Modular Structure, IEEE Trans. on Reliab., Vol. 38, No. 3, 1989 August, pp. 365-372.

[G21] H. Ohtera, S. Yamada, Optimum Software-Release Time Considering an Error-Detection Phenomenon during Operation, IEEE Trans. on Reliab., Vol. 39, No. 5, 1990 December, pp. 596-599.

[G22] N. D. Singpurwalla, Determining an Optimal Time Interval for Testing and Debugging Software, IEEE Trans. on Software Engineering, Vol. 17, No. 4, April 1991. pp. 313-319.

H: 关于模型的比较

[H1] A. A. Abdel-Ghaly, P. Y. Chan and B. Littlewood, Evaluation of competing software reliability predictions, IEEE Trans. Software Eng., SE-12(9), 950-967, 1986.

[H2] C. J. Dale, Software reliability models, In Software Reliability: State of the Art Report, Eds. A. Bendell and P. Mellor, Pengamon Infotech Ltd., pp. 31-34, 1986.

[H3] W. K. Ehrlich and T. J. Emerson, Modeling software failures and relaibility growth during system testing, Proc. 9:th Int. Conf. on Software Eng., pp. 72-82, 1987.

[H4] A. Iannino, B. Littlewood, J. D. Musa and K. Okumoto, Criteria for software reliability model comparisons, IEEE Trans. Software Eng., SE-10(6), 687-691, 1984.

[H5] P. A. Keiller, B. Littlewood, D. R. Miller and A. Sofer, On the quality of software reliability predictions, Proc. NATO ASI on Electronic Systems Effectiveness and Life Cycle Costing, Springer, pp. 441-460, 1983.

[H6] A. M. Leone, Selecting on appropriate model for software reliability, 1988 Proc. Ann. Rel. & Maintain. Symp., pp. 208-213, 1988.

[H7] B. Littlewood, Theories of software reliability: how good are they and how can they be improved? IEEE Trans. Software Eng., SE-6(5), 489-500, 1980.

[H8] B. Littlewood, A. A. Abdel-Ghaly and P. Y. Chan, Tools for the analysis of the accuracy of software reliability predictions, In Software System Design Methods, Ed. J. K. Skwirzynski,

NATO ASI Series, Vol. F22, Springer, Berlin, pp. 299-355, 1986.

[H9] B. Littlewood, How good are software reliability predictions, In Software Reliability: Achievement and Assessment, Ed. B. Littlewood, Blackwell, Oxford, pp. 172-191, 1987.

[H10] J. D. Musa and K. Okumoto, A comparison of time domains for software reliability models, J. Systems and Software, 4, 277-287, 1984.

[H11] A. N. Sukert, Empirical validation of three software error prediction models, IEEE Trans. Rel., R-28(3), 199-205, 1979.

[H12] R. Troy and R. Moawad, Assessment of software reliability models, IEEE Trans. Software Eng., SE-11(12), 839-850, 1985.

[H13] M. Xie and O. kerlund, Applications of software reliability models: possible problems and practical solutions, Prec. SRE'89, Stavanger, Norway, Oct. 10-12, 1989.

[H14] Min Xie, Ren-Zuo Xu, Combining Test Strategy in Reliability Modelling, Proc. 3rd Symposium on Reliability Mathematics, Sep. 1989. Xi'an, China. 210-215.

I: 关 于 应 用

[I1] J. E. Angus, The application of software reliability models to a major C31 system, 1984 Proc. Ann. Rel. & Maintain. Symp., pp. 268-274, 1984.

[I2] P. G. Bishop and F. D. Pullen, Probabilistic modelling of software failure characteristics, In SAFECOM'88: Safety Related Computers in an Expanding Market, Ed. W. D: Ehrenberger, pp. 87-93, 1988.

[I3] J. B. Bowen, Application of a multi-model approach to estimating residual software faults and time betwen failures, Quality and Reliability Engineering International, 3, 41-51, 1987.

[I4] W. H. Farr and O. D. Smith, A tool for statistical modeling and estimation of reliability functions for software: SMERFS, J. Systems & Software, 8, 47-55, 1988.

[I5] K. K. Govil, Incorporation of execution time concept in several software reliability models, Reliability Engineering, 7, 235-249, 1984.

[I6] X. Z. Huang, The new applications of the hypergeometric distribution model on software reliability, Microel. Rel., 25(4), 713-714, 1985.

[I7] K. Kanoun and T. Sabourin, Failure analysis and operational reliability evaluation in automatic telephone switching software, Technology and Science of Informatics, 6(7), 497-512, 1987.

[I8] Y. Komuro, Evaluation of software products testing phase-application to software reliability growth models, In Reliability Theory and Applications, Eds. S. Osaki and J. H. Cao, world Scientific, Singapore, pp. 188-194, 1987.

[I9] G. A. Kruger, Project management using software reliability growth models, Hewlett-Pachard Journal, 39, 30-35, 1988.

[I10] T. S. Liu and T. H. Howell, Survey and application of software reliability models, Proc. Int. Phoenix Conf. on Computer and Communications, Ariz, pp. 200-206, 1984.

[I11] P. Mellor, Experiments in software reliability estimation, Reliability Engineering, 18, 117-129, 1987.

[I12] J. D. Musa and K. Okumoto, Application of basic and logarithmic Poisson Execution time models in software reliability, In Software System Design Methods, Ed. J. K. Skwirzynski, NATO ASI

Series, Vol. F22, Springer, Berlin, pp. 275-298, 1986.

[I13] L. Rydstr m and O. Viktorsson, Software reliability prediction for large and complex telecommunication systems, Proc. 22th Annual Hawaii Int. Conf. on System Sciences, Jan. 3-6, Hawaii, USA, pp. 312-319, 1989.

[I14] I. P. Schagen and M. M. Sallih, Fitting software failure data with stochastic models, Reliability Engineering, 17, 111-126, 1987.

[I15] M. L. Shooman, Reliability of process control software, In IFAC Reliability of Instrumentation Systems, The Hague, Netherlands, 21-30, 1986.

[I16] P. Spreij, Parameter estimation for a specific software reliability model, IEEE Trans. Rel., R-34 (4), 323-328, 1985.

[I17] M. Xie, Some stochastic models for software reliability analysis, Proc. 4th Int. Symp. on Applied Stochastic Models and Data Analysis, Dec. 7-9, Nancy, France, 1988.

[I18] R. Z. Xu, An empirical investigation: software errors and their influence upon development of WPADT, Proc. 9th Int. Computer Software & Applications Conf., Chicago, USA, Oct. 9-10, 1985.

[I19] M. Yaacob and M. G. Hartley, A Survey of microprocessor software reliability with an illustrative example, Int. J. Elect. Eng. & Edu., 18, 159-174, 1981.

[I20] S. Yamada, H. Ohetra and H. Narihisa, A testing-efort dependent software reliability model and its application, Microel. Rel., 27(3), 507-522, 1987.

[I21] M. Zaki and M. M. Ei-Boraey, Analysis of reliability models for interconnecting MIMD systems, J. Systems & Software, 8(2), 133-144, 1988.

[I22] M. R. Bastos Martini, Karama Kanoun, and J. Moreira de Souza, Software-Reliability Evaluation of the TROPICO-R Switching System, IEEE Trans. Reliab., 39(3), AUGUST 1990, 369-379.

[I23] J. M. Caruso, D. W. Desormeau, Integrating prior Knowledge with a Software Reliability Growth Model, Proc. 11-th ICSE, Austin, Texas, USA. 238-245.

J: 其它近期的参考文献

[J1] Xu Renzuo, Liu Lianjun, A Dynamical Software Reliability Prediction Modeling Method and Its Implementation. Proceedings of The First Beijing International Conference on Reliability, Maintainability And Safety (BICRMS'92), Beijing, China October 12-15, 1992, pp. 663-667.

[J2] T. Downs, P. Garrone, Some New Models of Software Testing with Performance Comparisons, IEEE Trans. on Reliab, Vol. 40, No. 3, 1991, August, pp. 322-337.

[J3] N. Karunanithi, D. Whitley, and Y. K. Malaiya, Prediction of Software Reliability Using Neural Networks, Proc. 1991 IEEE Int. Symp. on Soft. Rel. Eng., May 17-19, 1991, pp. 124-130.

[J4] N. Karunanithi and Y. K. Malaiya, The Scaling Problem in Neural Networks for Software Reliability Prediction, Proc. 1992, IEEE Int. Symp. on Soft. Rel. Eng., October, 1992, pp. 76-82.

[J5] T. M. Khoshgoftaar, A. S. Pandya and H. B. More, A Neural Network Approach for Predicting Software Development Faults, Proc. 1992, IEEE Int. Symp. on Soft. Rel. Eng., October, 1992, pp. 83-89.

[J6] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park and D. M. Nicol, Estimating the Probability of Failure When Testing Reveals No Failures, IEEE Trans. on Software Engineering, Vol. 18, No.

1, January 1992, pp. 33-43.

名 词 索 引

① 凡英文字母打头的一律视为一画。