高技术通讯 2014 年 第24 卷 第6 期:558 ~564

doi:10.3772/j.issn.1002-0470.2014.06.002

考虑不完美排错情况的软件可靠性过程仿真①

侯春燕② 陈 晨③ 王劲松 林 胜

(天津理工大学计算机与通信工程学院 天津 300384)

摘 要 针对传统非齐次泊松过程(NHPP)类软件可靠性分析方法存在忽略对软件测试中的故障排除过程的问题,提出了一种新的描述软件测试中的实际可靠性增长过程的仿真方法。该方法采用排队论建模软件测试中可能存在的故障排除行为。排队模型全面考虑了故障排除中可能存在的不完全排错,甚至是引入新故障的情况,并考虑了排错资源的局限性问题。以排队模型作为仿真模型,开发出仿真过程实现对软件可靠性过程的仿真。利用公开发表的一组软件失效数据对仿真方法进行的验证表明,与现有的仿真方法相比,该方法由于更全面地考虑了软件测试过程,因此取得更好的拟合结果。

关键词 软件故障排除,非齐次泊松过程(NHPP),不完美排错,软件可靠性增长模型(SRGM),排队论

0 引 言

软件可靠性是指在给定的时间内、给定的条件下软件不引起系统失效的概率。在过去的30 年中,人们对软件可靠性分析方法进行了大量研究,这些研究主要集中在对非齐次泊松过程(non-homogeneous Poisson process, NHPP)类软件可靠性增长模型的研究上。近年来,基于率的仿真方法开始用于软件随机失效过程的分析,因为这种方法能够灵活地跟踪软件动态失效过程。然而,目前的软件可靠性分析方法都忽略了对软件测试中的故障排除过程,对分析结果带来了不利影响,针对这一问题,本文提出了一种考虑了不完美排错情况的软件可靠性过程仿真方法,并通过实例分析了其有效性。

1 相关研究

Goel 和 Okumoto 最早对 NHPP 模型进行了研究,提出了经典的 G-O 模型[1],该模型至今在很多应用上都有良好的表现。后来提出的许多 NHPP 模型都是基于对 G-O 模型的改进。这些模型大部分继承了 G-O 模型的一个假设:检测到的故障能够立即完全修复或故障排除时间可以忽略不计。后来一些研究对该假设进行了部分改进,提出了考虑不完美排错(排除故障)情况的 NHPP 模型[2,3],但这类模型没有考虑故障排除时间延迟。实际上,从测试人员检测出软件故障到报告给软件开发人进行诊断、排错、验证这个过程中会有一定的时间延迟。传统的 NHPP 模型通常只考虑了软件测试中的故障检测过程而忽略了故障排除(排错)过程。在软件测试过程中,只有在故障完全排除后软件可靠性才能经历增长。传统 NHPP 模型对排错过程的忽略会导致模型过于乐观的估计结果,给软件开发者和用户带来不利的影响。

现在,研究人员已经开始讨论如何使用排队方法来解释软件测试中的排错行为。Dohi 等把有限故障和无限故障两类 NHPP 模型放到一个统一的建模框架中,通过引进一个无限服务员排队(infinite server queue, ISQ)模型描述软件排错行为,说明可以在经典的 NHPP 模型中考虑软件排错过程[4]。Huang 等引用一个实例系统 P1 来阐明故障排除过程不可以忽略这个问题,指出对于该大型的软件系统来说,检测到的故障需要花费数月时间去排除。他们分别使用 ISQ 模型和有限服务员排队(finite server queue, FSQ)模型描述软件的排错行为,推导

① 国家自然科学基金(651170301)和天津市科技支撑计划重点(13ZCZDGX02200)资助项目。
② 女,1980 年生,博士,讲师;研究方向:软件可靠性,软件测试;E-mail: chunyanhou@163.com
③ 通讯作者,E-mail: nkchenchen@nankai.edu.cn
(收稿日期:2013-10-25)

出新的软件可靠性增长模型来预测软件可靠性。与基于 ISQ 的模型相比,基于 FSQ 的模型考虑到了排错资源的约束性问题,但是由于模型过于复杂,最终没有得到求解[5]。Lin 等用基于率的仿真方法解决了这个问题。他们用排队论描述和解释软件开发中的故障排除行为。基于排队模型,开发出仿真程序实现对软件可靠性过程的仿真[6]。后来,Huang 等考虑到在排错过程中排错速率可能在某些特定的点发生变化,提出一个具有多个移动点的扩展的 ISQ 模型来预测和估计软件可靠性[7]。

从以上分析可以看出,考虑排错资源的局限性会导致 NHPP 模型难以求解。离散事件仿真为这个问题提供了一种解决方法。仿真方法的灵活性和动态性,能够使它放宽基于模型方法一些过于严格的假设,跟踪软件的动态失效过程。目前所提出的 NHPP 类软件可靠性增长模型将测试及故障排除阶段的总可靠性增长看作或近似为执行时间中的马尔可夫过程,或非齐次泊松过程,后者实际上也是马尔可夫过程。尽管不同模型在基本失效机制上的假设可能有很大差别,但在数学上仅是率函数的形式不同而已。因此可以采用基于率的仿真方法实现对软件可靠性过程的仿真。

近年来研究人员将基于率的仿真方法用于软件随机失效过程的分析。基于率的仿真将软件可靠性过程看作是由率函数控制的随机过程,通过率函数产生出随机过程的仿真数据,是传统解析模型的自然扩展。Tausworthe 等用纯生非齐次连续时间马尔科夫链(NHCTMC)来描述软件的随机失效过程,基于此提出一组仿真过程,该仿真算法可以应用于软件生命周期中的每个阶段[8]。后来,Gokhale 等针对软件测试阶段在仿真过程中考虑了故障排除过程中可能采用的各种排错策略[9]。Lin 等进一步发现软件测试中的故障排除时间不可以忽略,他们使用排队理论来描述故障排除行为,其中考虑了排错资源的约束性问题,在排队模型的基础上开发出仿真过程实现对软件的可靠性过程的仿真,但没有考虑软件测试中可能出现的不完美排错[6]。文献[10]和[11]将仿真方法用于描述了构件软件集成测试过程中应用可靠性随时间的动态变化性。

本文针对传统的软件可靠性分析方法所存在的问题,基于排队论,提出一种新的考虑不完美排错情况的软件可靠性过程仿真。

2 排队模型

排队论,也称随机服务系统理论。排队系统一般可描述为顾客到达请求服务,如果无法立即得到服务就排队等待,服务完成之后离开。每个排队系统包括一个或多个服务人员。软件测试中的故障排除过程描述的是排错人员对检测到的故障进行修复的过程,可以用排队系统进行建模。排队模型的顾客到达过程对应于软件测试中的故障检测过程,顾客服务过程对应于故障排除过程,排错人员相当于队列服务员。用排队模型建模软件测试中的故障排除行为如图 1 所示,其中考虑了不完美排错。

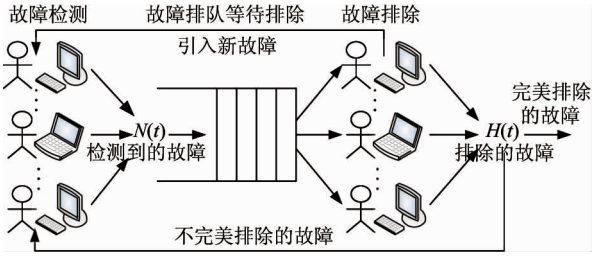

图 1 排队模型

下面对排队模型中的顾客到达规律和顾客服务规律分别进行分析。

设随机过程 $\{N(t), t \geqslant 0\}$ 表示排队系统的顾客到达过程,$N(t)$ 表示在长度为 t 的执行时间间隔内到达故障数,它依赖于软件的失效速率 $\lambda(t)$,则无论在 τ 时刻构件 j 处于什么状态,在 τ 与 $\tau+\Delta\tau$ 之间的充分小的时间区间内下式成立:

$$\begin{cases} P(N(\tau+\Delta\tau)-N(\tau)=1)=\lambda(\tau)\Delta\tau+o(\Delta\tau) \\ P(N(\tau+\Delta\tau)-N(\tau)>1)=o(\Delta\tau) \end{cases} \tag{1}$$

其中,函数 $o(\Delta\tau)$ 定义为

$$\lim_{\Delta\tau\to 0}\frac{o(\Delta\tau)}{\Delta\tau}=0 \tag{2}$$

函数 $o(\Delta\tau)$ 指示在时间区间 $(\tau,\tau+\Delta\tau)$ 内多于一次失效发生的概率值是可忽略不计的。

软件的失效过程满足 NHPP,t 时刻软件失效速率与软件中剩余故障数成正比。由于考虑了不完美排错,不完美排除的故障会对软件中剩余故障数产生影响。不完美排错存在两种情况:一种是原有故障仍有保留,另一种是引入新故障。这两种情况不是互斥关系。引入新故障是由故障排除行为引起的,不论故障排除成功与否。因此,t 时刻软件中剩

余的故障数可以表示为

$$x(t) = a - m_r(t) + m_d(t) \times \alpha \tag{3}$$

其中 α 表示引入新故障的概率。设 p 表示故障完全排除的概率,则有

$$m_r(t) = p \times m_d(t) \tag{4}$$

将式(4)代入式(3)中可以得到

$$x(t) = a - m_d(t) \times (p - \alpha) \tag{5}$$

因此,软件失效速率 $\lambda(t)$ 可以表示为

$$\lambda(t) = \frac{\mathrm{d}m_d(t)}{\mathrm{d}t} = b(t) \times x(t)$$
$$= b(t) \times (a - m_d(t) \times (p - \alpha)) \tag{6}$$

当故障查出率为常数,即 $b(t) = b$。根据初始条件 $m(0) = 0$ 求解微分方程(式(6)),得到

$$m_d(t) = \frac{a}{p - \alpha}(1 - e^{-b(p-\alpha)t}) \tag{7}$$

失效发生速率为

$$\lambda(t) = \frac{\mathrm{d}m(t)}{t} = abe^{-b(p-\alpha)t} \tag{8}$$

当 $p = 1, \alpha = 0$ 时,也就是完美排错的情况下,代入式(7)和(8),就得到经典的 G-O 模型。

当 $b(t) = \frac{b^2 t}{1 + bt}$, 求解式(6),得到

$$m(t) = \frac{a}{p - \alpha}[1 - ((1 + bt)e^{-bt})^{(p-\alpha)}] \tag{9}$$

失效发生速率为

$$\lambda(t) = ab^2(1 + bt)te^{-b(p-\alpha)t} \tag{10}$$

在完美排错的情况下,将 $p = 1, \alpha = 0$ 代入式(9)和(10),就得到 S-形 Yamada 模型。

当 $b(t) = \frac{b}{(1 + \beta e^{-bt})}$ 时,求解式(6),得到

$$m(t) = \frac{a}{p - \alpha} \times \frac{1 - e^{-b(p-\alpha)t}}{1 + \beta e^{-bt}} \tag{11}$$

失效发生速率为

$$\lambda(t) = \frac{abe^{-b(p-\alpha)t}}{1 + \beta e^{-bt}} + \frac{ab\beta e^{-bt}(1 - e^{-b(p-\alpha)t})}{(p - \alpha)(1 + \beta e^{-bt})^2} \tag{12}$$

在完美排错的情况下,将 $p = 1$, $\alpha = 0$ 代入式(11)和(12),就得到变形 S-形模型。

设随机过程 $\{H(t), t \geqslant 0\}$ 表示排队系统的顾客离开过程。$H(t)$ 表示在长度为 t 的执行时间间隔内排除的故障数。对于任意时刻 $t \geqslant 0$, 满足 $H(t) \leqslant N(t)$, 即排除的故障数滞后于检测的故障数。$H(t)$ 不仅依赖于故障检测过程 $\{N(t), t \geqslant 0\}$, 还依赖于排错人员的数目和排错速率。

故障从检测出到完成排除需要一定的时间延迟。这个延迟包括两部分:(1)分配排错人员的时间,其中包括由测试人员报告给排错人员的时间和由于排错人员数目有限故障等待的时间;(2)排错人员排除故障所花费的时间。在 NHPP 解析模型中详细考虑排错时间延迟将会导致模型难以求解,而仿真方法正是解决这类问题的较好方法。

指数分布是常用的服务时间分布假设。设排错人员的排错速率为 μ, 则对某一故障排错人员已经进行 t 时间的修复后,将在时间间隔 $(t, t + \Delta t)$ 内完成故障排除的概率为

$$P(t \leqslant T \leqslant t + \Delta t \mid T > t)$$
$$= \frac{g(t) \times \Delta t}{P(T > t)} = \frac{g(t) \times \Delta t}{1 - G(t)} = u \times \Delta t \tag{13}$$

其中 $G(t)$ 表示参数为 μ 指数分布函数,$g(t)$ 为其概率密度函数。

根据以上分析,排队模型中的顾客到达过程为 NHPP,服务员的服务速率为 μ, 服务规则为先来先服务。所有的服务员共享一个公用队列。该队列可建模为 G/M/N 队列模型。该队列模型是一个生灭过程模型,其生灭速率为

$$\begin{cases} \lambda(t) = b(t) \times (a - m_d(t) \times (p - \alpha)) \\ \mu(t) = \begin{cases} lu, & 0 \leqslant l < N \\ N\mu, & l \geqslant N \end{cases} \end{cases} \tag{14}$$

3 仿真过程

以第 2 节建立的排队模型为仿真模型,开发出仿真过程,实现对软件可靠性增长过程的仿真。仿真过程的实现基于如下假设:

(1) 对软件执行基于其运行剖面的黑盒测试,软件失效的发生是由软件中的故障引起的,软件系统的失效过程满足 NHPP,所有故障之间相互独立;

(2) 所有故障之间相互独立,软件在充分小的执行时间区间 $(t, t + \Delta t)$ 内发生一次失效的概率近似为 $\lambda(t)\Delta t$, 多于一次失效发生的概率可以忽略;

(3) 故障排除时间不可以忽略,已经排除的故障数滞后于检测到的故障数;

(4) 排错可能是不完全的,还可能会引入新故障,故障排除活动不会影响故障检测过程的继续进行,不完全排除的故障和引入的新故障会被故障检测过程检测到;

(5) 排错人员的数目是有限的,服务时间满足速度为 μ 的指数分布,在充分小的时间区间 $(t, t +$

Δt）内排错人员完成故障排除的概率为 $\mu\Delta t$;

（6）检测到的故障到达排错系统后进入队列等待,满足一定的时间延迟后才能参与排错人员分配,排错人员的服务规程为随机分配。

我们用类 C 语言开发出基于率的仿真算法 Procedure S 实现对软件测试中软件可靠性增长过程的仿真,如图 2 所示。该过程接收如下参数作为输入:时间步 *dt*(为了保证仿真精度,*dt* 必须足够小,并且在任何时候都满足 $rate(t) * dt < 1$,$rate(t)$ 表示事件发生速率;软件测试时间 *t*;软件的失效速率(* *lamda*);排错人员总数 *level*;排错速率(* *mu*);完美排错概率 *p*;分配排错人员的时间延迟 *offset*。除了输入参数,ProcedureS还定义如下变量来控制仿真过程的执行:变量 *global _ clock* 记录测试已经执行的时间;*working _ server* 累计已经分配的排错人员数;*total _ arrival* 累计检测到的故障数,即到达排队系统的总故障数;*waiting _ head* 指示等待排错资源的第一个故障;*correcting _ head* 指示占有资源正在进行排除的第一个故障;数组 *queue* 描述由检测到的故障组成的排错队列,如图 3 所示。故障根据检测到的先后次序进入队列 *queue* 等待分配排错人员,当分配到排错人员后就进入滑动窗口进行修复,修复完成后退出窗口。队列 *queue* 又由滑动窗口分为两条队列,一条队列靠近队列头,由已经完成修复的故障组成,另一条队列靠近队列尾,由等待排错资源的故障组成。新分配到排错资源的故障进入滑动窗口,修复完成的故障退出窗口,窗口随之由队头向队尾不断滑动,两条队列此消彼长,直至测试完成。滑动窗口是变长的,某时刻滑动窗口的长度由等待的故障数目和排错人员的数目决定的。

```
1   void simulation ( double dt, double t, double (*lamda)(double), int
2   level, double (*mu)(double),double p,double offset)
3   {   double global_clock; struct fault_info queue[max_size];
4       int working_server, total_arrival, waiting_head, correcting_head;
5       while (global_clock < t)
6       {    ALLOCATING:
7            while(working_server<level&&waiting_head<total_arrival)
8       {if ((global_clock-queue[waiting_head].detection_time) < offset)
9                    break;
10               queue[waiting_head].state= CORRECTING;
11               queue[waiting_head].allocation_time=global_clock;
12               working_server++; waiting_head ++;   }
13           DETECTING:
14           if ( occur ( dt, lamda (global_clock) ) )
15           {   queue[total_arrival].detection_time=global_clock;
16               queue[total_arrival].state=WAITING;
17               total_arrival ++;   }
18          CORRECTING:     int temp=correcting_head;
19          for (int i = temp; i < waiting_head; i++)
20          {    if ((occur(dt,mu(global_clock)))&&(random(0,1) < p))
21               {   queue[i].state = CORRECTED;
22                   queue[i].correction_time = global_clock;
23                   if(i !=correcting_head)
24                   {    struct fault_info fault=queue[correcting_head];
25                        queue[correcting_head]=queue[i];
26                        queue[i]=fault;   }
27                   working_server --; correcting_head++;   }    }
28          global_clock += dt;   }   }
```

图 2　Procedure S：考虑不完美排错的仿真过程

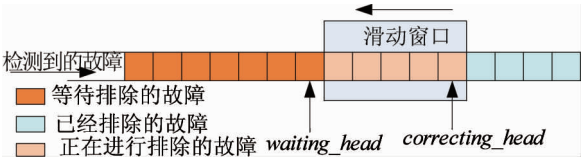

图 3　带有滑动窗口的队列

我们定义结构体 *fault _ info* 来封装故障检测过程中检测到的故障信息,跟踪故障状态的变化,如图 4所示。*fault _ info* 中记录了故障被检测到的时间 *detection _ time*,分配到排错资源进行排除的时间 *allocation _ time*,完成排除的时间 *correction _ time*,以及故障状态 *state*。三个时间满足 *detection _ time* < *allocation _ time* < *correction _ time*。故障从被检测出到被排除,一共经历三种状态:WAITING,表示等待排错资源;CORRECTING,表示占有资源正在被修复;CORRECTED,表示修复完成,释放资源。

```
struct fault_info
{    double detection_time;
     double allocation_time;
     double correction_time;
     fault_state state;
}
```

图 4　结构体定义

接下来详细介绍 Procedure S 的具体实现。仿真过程中,每次执行采取的行动包括如下三步:

资源分配 ALLOCATING:这一步为等待队列中的故障分配排错资源。如果有空闲的排错资源且等待队列中有故障在等待,那么检查等待队列对首的故障是否达到可以分配资源的时间,如果时间不满足,则退出资源分配,因为队列头的故障是最早检测到的,如果它的时间不满足,那么队列中其他故障肯定也不满足。如果条件满足,给等待队列队首的故障分配资源,更改其状态为 CORRECTING,记录分配时间,故障进入滑动窗口,窗口变长。以上过程重复进行,直到条件不满足为止。

故障检测过程 DETECTING:根据排队模型的顾客到达规律,对故障检测过程执行仿真。函数 *lamda*( )返回软件失效速率。由函数 *occur*( )判断,如果软件此次执行过程中发生失效,记录故障发生时间,并将其放入等待队列对尾。

故障排除过程 CORRECTING:在这一步对滑动窗口中(*waiting _ head-correcting _ head*)个占有排错资源的故障依次进行修复。利用函数 *occur*( )判断修复是否完成。如果修复完成,由于考虑不完美排错,还需要判断修复是否成功。采用生成随机数的方法,如果生成 0 到 1 之间的随机数小于完美排错概率 p,则故障修复成功,更改故障的状态为 CORRECTED,记录排除时间,释放排错资源,累加相应的计数器,并将其移动出滑动窗口。如此重复,直到完成对滑动窗口中所有故障的遍历。

函数 *occur*( )实现对失效发生事件和故障排除事件的仿真。对事件的仿真可以有多种实现方法。我们采用最普遍的随机数生成器来实现。根据事件的数学概率分布,编程实现随机数生成器来模拟各种事件。根据假设(2)和(5)可知,在时间间隔 dt 内事件发生的概率近似为 $rate(t) * dt$。因此该函数首先生成一个 0 到 1 之间均匀分布的随机数 x,然后比较 $rate(t) * dt$ 和 x。如果 $x < rate(t) * dt$,则事件发生。

以上三个步骤重复进行,直到到达总测试时间 t。

4 实例分析

本节将用公开发表的数据集 DS1 来验证我们提出的仿真方法的有效性。DS1 来自于 RADC (Rome Air Development Center)工程系统 T1。失效数据是在严格的管理下仔细收集到的。T1 系统包含 217000 条发布的目标指令,可应用于实时命令和控制。9 个程序员经过 21 周的测试,共检测和排除了 136 个故障。DS1 表示为规定时间区间内发生故障数的测定数据。

根据 DS1,我们用考虑不完美排错的变形 S-形模型来建模故障检测过程,如式(11)和(12),因为该模型与数据拟合效果较好。采用最小二乘法进行参数估计,得到的参数为 $a = 136.0548$, $b = 0.3417$, $p = 0.9349$, $\alpha = 0.0651$, $\beta = 136.3469$。

设仿真过程中总的测试时间为 $t = 21$,时间步 $dt = 0.001$,每个排错人员的排错速率 $\mu = 3$。根据 Procedure S 对软件测试过程进行仿真。执行仿真过程 5000 次,计算得出平均运行剖面。实验结果发现,当排错资源分配时间延迟满足 *offset* = 0.8 时,得到最好的拟合结果,仿真结果如图 5 所示。

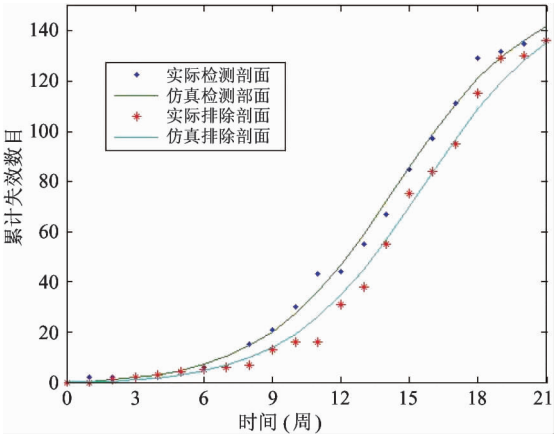

图 5 故障检测剖面和故障排除剖面

从图中可以看出,软件测试中的故障检测剖面和故障排除剖面是不同的,因此在对软件可靠性进行分析时,不能忽略故障排除过程。Procedure S 的仿真结果与实际的测试数据具有较好的拟合效果。

为了进一步验证我们提出的仿真方法的有效性,我们采用量化的方法来比较 Procedure S 和 Lin 提出仿真方法[6]。用于度量曲线拟合效果的标准主要有误差平方和(sum of squared errors,简称 SSE)和回归曲线方程相关指数(R-Square)。分别定义如下:

$$SSE = \sum_{i=1}^{n} (y_i - \hat{m}(t_i))^2 \tag{15}$$

$$R - Square = \frac{\sum_{i=1}^{n} (\hat{m}(t_i) - \bar{y})^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{16}$$

其中 n 表示失效数据集中失效样本的数量; $\hat{m}(t_i)$

表示到 t_i 时刻为止故障累计数的估算值; y_i 表示到 t_i 时刻为止故障累计数的实测值。

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i \tag{17}$$

SSE 的值越小,曲线拟合得越好;*R-Square* 的值越接近 1,曲线拟合越好。两种仿真方法拟合结果如表 1 所示。分析结果表明,虽然 Lin 仿真算法的 R-Square 略优于 Procedure S,但 SSE 指标远远落后与 Procedure S,综合衡量,我们提出的仿真方法的拟合结果较理想,优于 Lin 的仿真方法[6]。

表 1　仿真算法的数据拟合情况对比

比较标准	Procedure S	Lin 仿真算法
SSE	355.629	1288.947
R-square	0.928	1.035

接下来,进一步仿真在不同的排错人员数目约束条件下的故障排除过程,分析排错人员数目对排错系统吞吐量的影响。设排错人员数目分别为 1 到 10,以及无穷大。在每种资源配置条件下分别执行仿真算法各 5000 次,计算出每种情况下排错系统的平均故障排除剖面,如图 6 所示。

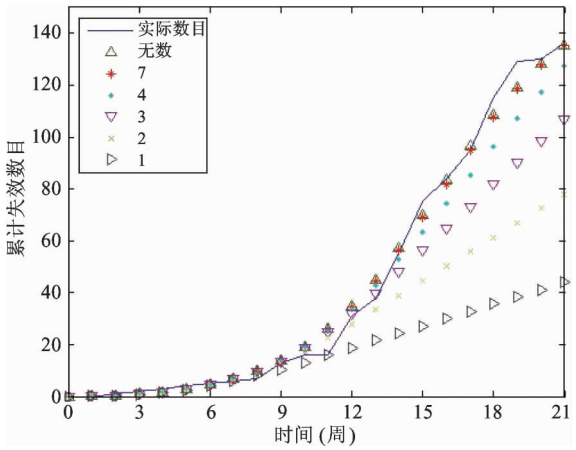

图 6　不同排错人员数目约束条件下的故障排除剖面

为了区别不同情况下的故障排除剖面,图 6 只显示出排错人员数为 1 到 7,以及无穷大时的故障排除剖面。从图中可以清楚看出,当排错人员数目从 1 增加到 7 时,排错系统的吞吐量逐渐增加。排错系统的排错人员数为 7 的故障排除剖面与排错人员数目无穷大时的剖面非常接近,它们都接近于实际的故障排除剖面。当排错人员数目达到 7 个之后,再增加排错人员的数目不会显著提高排队系统的吞吐量。而排错人员数目到达 7 之前,不同排错人员数目配置下的故障排除剖面是不同的。

表 2 进一步统计分析了不同排错人员数目配置情况下的排错系统性能。可以看出,随着排错人员数目的增多,排错人员的利用率、平均队列长度、平均等待时间、平均响应时间逐渐降低。当排错人员的数目超过 7 个时,排错系统的吞吐量与排错人员的数目就不存在明显的依赖关系,所有检测到的故障都需要 22 周完成排除。此时再增加排错人员的数目无法提高排错系统性能。当排错人员的数目满足系统需求时,项目管理者如果想提高排错系统的性能,提前完成软件测试,就需要提高排错人员的技能,提高他们的排错速度 μ。

表 2　不同排错人员数目配置下排错系统的性能比较

人数	1	2	3	5	7	9
周平均排除故障数	2.09	3.72	5.09	6.4	6.43	6.44
21 周未排除故障数	98	65	36	8	7	7
排除故障时间(周)	58	46	35	25	22	22
平均响应时间 (周)	4.24	3.07	2.28	1.32	1.17	1.15
平均等待时间(周)	4	2.81	1.99	0.97	0.82	0.82
平均队长	32.71	21.82	13.72	5.08	2.93	1.83
排错人员利用率(%)	92.33	88.75	87.78	85.95	84.12	78.7

5　结　论

本文针对传统 NHPP 类软件可靠性分析方法对软件测试中故障排除过程的忽略或简化,提出了一个新的基于率的仿真方法来描述软件测试中实际的可靠性增长过程。该方法采用排队论建模软件测试过程,并考虑了软件测试中可能存在的不完美排错情况,以及排错资源的局限性问题。该仿真方法放宽了传统方法中一些过于严格的假设,全面考虑了软件测试中可能存在的排错行为,因此可以更准确地描述软件的可靠性过程,预测软件可靠性随测试过程的增长。

参考文献

[1] Goel A L, Okumoto K. Time-dependent error-detection rate model for software and other performance measures.

IEEE Transactions on Reliability, 1979, 28(3): 206-211

[2] 谢景燕, 安金霞, 朱纪洪. 考虑不完美排错情况的 NHPP 类软件可靠性增长模型. 软件学报, 2010, 21(5): 942-949

[3] Kapur P K, Pham H, Anand S, et al. A unified approach for developing software reliability growth models in the presence of imperfect debugging and error generation. *IEEE Trans On reliability*, 2011, 60(1):331-339

[4] Dohi T, Osaki S, Trivedi K S. An infinite server queueing approach for describing software reliability growth: unified modeling and estimation framework. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference(APSEC'04), Busan, Korea, 2004. 110-119

[5] Huang CY, Huang WC. Software reliability analysis and measurement using finite and infinite server queueing models. *IEEE Trans on Reliability*, 2008, 57(1): 192-203

[6] Lin C T, Huang C Y. Staffing level and cost analyses for software debugging activities through rate-based simulation approaches. *IEEE Trans on Reliability*, 2009, 58(4): 711-724

[7] Huang C Y, Hung T Y, Hsu C J. Software reliability prediction and analysis using queueing models with multiple change-points. In: Proceedings of the 3rd IEEE International Conference on Secure Software Integration and Reliability Improvement, Shanghai, China, 2009. 212-221

[8] Tausworthe R C, Lyu M R. A generalized technique for simulating software reliability. *IEEE Software*, 1996, 13(2): 77-88

[9] Gokhale S S, Lyu M R, Trivedi K S. Incorporating fault debugging activities into software reliability models: a simulation approach. *IEEE Trans on Reliability*, 2006, 55(2): 281-292

[10] Gokhale S S, Lyu M R. A simulation approach to structure-based software reliability analysis. *IEEE Trans on Software Engineering*, 2005, 31(8): 643-656

[11] 侯春燕,崔刚,刘宏伟. 基于率的构件软件可靠性过程仿真. 软件学报, 2011, 22(11):2749-2759

Software reliability process simulation considering imperfect debugging

Hou Chunyan, Chen Chen, Wang Jinsong, Lin Sheng

(School of Computer and Communication Engineering, Tianjin University of Technology, Tianjin 300384)

Abstract

The study aimed at the problem that traditional methods for analysis of non-homogeneous Poisson process (NHPP) software reliability do not take account of the imperfect debugging in software test, and proposed a new simulation approach to describe the real software reliability growth process in software testing. It applies the queueing theory to modeling of possible debugging behaviors during software testing. Both imperfect debugging and limitations of debugging resources are considered in the queueing model. By taking the queueing model as a simulation model, a simulation procedure was developed to describe the software reliability growth process. The goodness-of-fit of the new simulation approach was examined by using a public software failure data set. The results show that compared with other existing simulation approaches, the proposed approach fits the failure data better due to the consideration of more details of software testing.

Key words: software fault correction, non-homogeneous Poisson process (NHPP), imperfect debugging, software reliability growth model (SRGM), queueing theory