软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(11):2749−2759 [doi: 10.3724/SP.J.1001.2011.03930]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于率的构件软件可靠性过程仿真*

侯春燕+, 崔 刚, 刘宏伟

(哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

Rate-Based Component Software Reliability Process Simulation

HOU Chun-Yan+, CUI Gang, LIU Hong-Wei

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

+ Corresponding author: E-mail: houchunyan@ftcl.hit.edu.cn

Hou CY, Cui G, Liu HW. Rate-Based component software reliability process simulation. *Journal of Software*, 2011,22(11):2749−2759. http://www.jos.org.cn/1000-9825/3930.htm

Abstract: In contrast to traditional model-based component software reliability analysis approaches, rate-based simulation approaches can flexibly trace the dynamic failure procedure of software. Consequently, they have been tentatively applied to analyze the reliability procedure of component software in recent years. However, existing simulation approaches are based on excessively simple assumptions in terms of the fault correction procedure in component software testing and are not able to describe the practical reliability procedure of the software system. Therefore, a simulation approach is proposed. It models the fault correction procedure with a hybrid queueing model, in which it takes the fault repair policy and the limitations of debugging resources into account. On this basis, a simulation procedure is developed to realize the simulation of component software reliability procedure. Finally, the evaluation experiment shows the potential of the simulation approach.

Key words: software reliability; component; component software; rate-based simulation; queueing theory

摘 要: 与传统的基于模型的构件软件可靠性分析方法相比,基于率的仿真方法由于可以灵活地跟踪软件动态失效过程,近年来开始用于分析构件软件的可靠性过程.但是,目前已经提出的仿真方法对构件软件测试中的故障排除过程做了过分简化的假设,而未能描述软件系统实际的可靠性过程.针对这个问题,提出了一种仿真方法.该方法采用一个混合排队模型建模故障排除过程,其中考虑到了排错策略和排错资源的局限性问题.在此基础上开发出仿真过程,实现对构件软件可靠性过程的仿真.实验结果表明了该仿真方法的有效性.

关键词: 软件可靠性;构件;构件软件;基于率的仿真;排队论

中图法分类号: TP311 **文献标识码**: A

从 20 世纪 90 年代开始,软件产业快速发展,软件迅速脱离“一切从零开始”的开发模式,转向高级复用技术.目前,基于构件的软件开发逐渐增多.与传统的软件开发模式相比,构件软件开发以集成已有的软件构件构造新的软件系统为目的.所集成的构件可能是由第三方开发,因此,构件的内部信息对构件使用者来说是透明的.随着构件软件的应用日益广泛,人们对构件软件质量的要求也越来越高.可靠性作为衡量构件软件质量的重要特

* 基金项目: 国家自然科学基金(60503015); 国家高技术研究发展计划(863)(2008AA01A201)
收稿时间: 2010-05-07; 修改时间: 2010-07-05; 定稿时间: 2010-08-13

性,其定量评估和预测已成为人们关注和研究的焦点.

对构件软件可靠性分析技术的早期研究主要集中在对基于体系结构的模型[1−4]的研究上.该方法根据软构件第三方开发和基于构件的软件开发特点,虽然具体的软构件仍是黑盒,但要求系统内部的体系结构信息可见.基于模型的方法建模构件软件系统的体系结构以及系统中每个构件的失效行为,然后采用解析的方法对模型求解得到软件应用的可靠性估计.这种模型适用于软件生命周期中各个阶段.除了进行可靠性分析以外,它们还能分析系统可靠性对构件可靠性的灵敏度,识别系统可靠性瓶颈,为各个阶段中任务实施以及资源配置提供指导.但是,基于体系结构的模型是一种静态模型,它要求软件系统的体系结构以及构件失效行为都采用时间无关的静态方式建模,最后得到的可靠性估计也是一个时间无关的单一的估计值.这种静态特性使基于体系结构的模型无法描述构件软件测试过程中应用可靠性在时间域上的动态变化性.在对测试阶段进行可靠性分析时,为了使模型易于求解,基于体系结构的模型往往会对测试过程做出偏离实际的简化假设[3].一个公用的假设就是,假设检测到的故障能够立即完全排除.实际上,这样的假设并不合理,它使构件软件可靠性分析只考虑了测试中的故障检测过程,忽略了故障排除过程.

通过改进基于体系结构模型的方法来解决以上问题,将会导致模型难以求解.离散事件仿真为这些问题提供了一种解决方法.仿真方法由于它所具有的灵活性和动态性,使它能够放宽基于模型的方法中一些过于严格的假设,跟踪软件的动态失效过程.近年来,基于率的仿真方法开始用于分析软件的随机失效过程.目前所提出的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.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.Tausworthed 等人用纯生 NHCTMC(non-homogeneous continuous time Markov chain)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[5].后来,Gokhale 等人针对软件测试阶段在仿真过程中考虑了测试过程中可能出现的不完全排错的情况[6].Lin 等人进一步发现,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[7].以上仿真方法分析的是普通软件的可靠性,虽然可用于对单个构件进行可靠性分析,但不适用于构件软件.Gokhale 和 Lyu 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,并考虑了排错策略[8].但是,该方法采用同种排错机制建模应用中所有构件故障的排除行为,共享排错机制或独立排错机制.构件软件应用是以异构方式开发的,一种排错机制不能满足应用中所有的构件,更一般的排错机制应是由这两种机制组成的混合排错机制.而且,该仿真方法没有考虑排错资源的约束性问题.实际的软件公司不可能拥有无限的排错资源,由于成本以及预算等原因,排错人员的数目往往是有限的且严格受控制的.当所有排错人员都被占用时,新检测到的故障必须排队等待.

本文针对传统的构件软件可靠性分析方法中所存在的问题,提出一种仿真方法来描述构件软件的可靠性过程.

1 可靠性过程建模

构件软件的可靠性过程,是指将所有构件组装在一起进行集成测试的过程.在集成测试阶段,根据构件软件应用的运行剖面对整个应用执行测试,应用中所有构件协同工作.随着应用的执行,构件中的故障不断被发现、排除,软件应用经历可靠性增长过程.该过程包括故障检测过程和故障排除过程,集成测试排错策略将这两个随机过程联系起来.

在软件测试中,根据排错策略组织排错活动的进行.排错策略根据项目的实际开发特征以及预算和进度要求,合理地配置排错资源,将检测到的故障以一定的方式分配给合适的排错人员进行故障排除.集成测试排错策略包含应用中每个构件的排错机制.构件软件应用是以异构方式开发的,对于检测到的不同构件的故障,需要采

用如下两种不同的机制进行排除:

(1) 独立排错机制.对于检测到的某个构件的故障,使用专门的独立排错资源进行修复.

(2) 共享排错机制.对于检测到的某些构件故障,根据检测到的先后顺序,使用共享的排错资源进行修复.

本文考虑由 n 个构件组成的顺序执行的终止构件软件应用.不失一般性,假设应用从构件 1 开始执行,到构件 n 执行结束.对于该应用,集成测试中需要用 $k(0<k\leqslant n)$个独立的排错系统对构件故障进行排除,k 由排错策略决定.我们用一个包含 k 个独立的 FSQ(finite server queueing)系统的混合的有限服务排队模型 HFSQM 来建模构件软件集成测试中的故障排除过程,如图 1 所示.

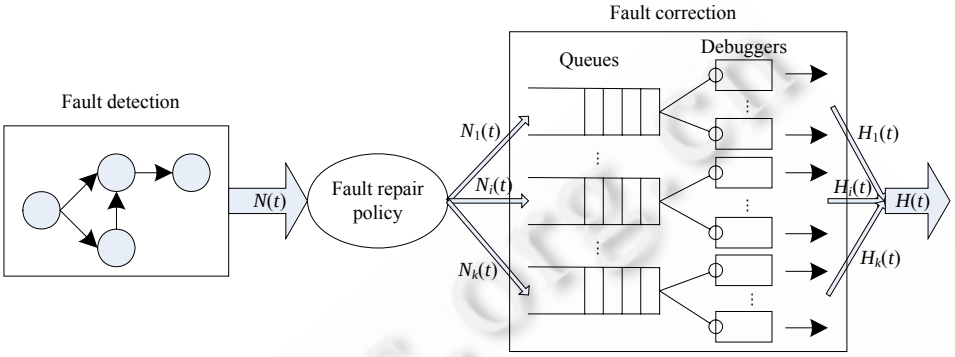


Fig.1 A hybrid finite server queueing model: HFSQM

图 1 混合的有限服务排队模型:HFSQM

过去,研究人员已经开始讨论如何使用排队理论来解释软件测试中的排错行为.Dohi 等人把有限故障和无限故障两类 NHPP(non-homogeneous poisson process)模型放到一个统一的建模框架中,通过引进一个 ISQ (infinite server queueing)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[9].Huang 等人分别使用 ISQ 和 FSQ 两类模型描述软件的排错行为,推导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,最终未能求解[10].Lin 等人用基于率的仿真方法解决了这个问题,基于 FSQ 模型实现了对软件可靠性过程的仿真[7].后来,Huang 等人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[11].

以上提到的排队模型都是基于普通软件的黑盒测试过程.集成测试是基于构件软件应用的运行剖面执行的灰盒测试过程,虽然对每个构件来说执行的仍是黑盒测试,但应用的体系结构信息可见.目前,还没有研究工作明确地将排队论用于构件软件可靠性分析中.HFSQM 排队模型建模构件软件集成测试过程,模型中每个 FSQ 排队系统描述根据排错策略分配到该系统中的构件故障的排除行为.我们将基于该模型实现对构件软件可靠性过程的仿真.

2 排队系统分析

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 1 个或多个服务人员.设 FSQ_i 表示 HFSQM 中第 i 个排队系统,$0<i\leqslant k$.根据经典排队模型 $A/B/n$ 建模 FSQ_i.其中,A 表示顾客到达的规律;B 表示服务时间分布;n 表示服务员的数目,即排错人员的数目,满足 $1\leqslant n<\infty$.

2.1 顾客到达规律

设随机过程$\{N_i(t),t\geqslant 0\}$表示FSQ_i的顾客到达过程.$N_i(t)$表示在长度为 t 的执行时间间隔内到达故障数.其密度函数$\Lambda_i(t)$表示故障到达速率.$\Lambda_i(t)$依赖于分配到该系统的每个构件的随机失效过程.

设随机过程$\{X_j(\tau),\tau\geqslant 0\}$表示分配到$FSQ_i$的构件$j$的随机失效过程,$X_j(\tau)$表示构件$j$在长度为$\tau$的执行时间间隔内发生的失效数.假设构件$j$的失效过程满足NHPP,失效速率为$\lambda_j(\tau)$,则无论在$\tau$时刻构件$j$处于什么状态,在$\tau$与$\tau+\Delta\tau$之间的充分小的时间区间内下式成立:

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

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1次失效发生的概率值是可忽略不计的.

因为构件的失效过程满足NHPP,所以FSQ_i的顾客到达过程也为NHPP,到达速率可表示为

$$\Lambda_i(t)=\sum\nolimits_j \eta_j(t)\lambda_j(t_j) \tag{3}$$

其中,$\eta_j(t)$表示t时刻构件j执行的概率,t_j表示到时间t构件j的累计执行时间.它们由集成测试剖面决定.在集成测试过程中,每当发生失效后,软件系统重启,从第1个构件重新开始执行.因此,发生在执行序列前面的构件中的故障被检测到并排除后,发生在执行序列后面的构件中所存在的故障才有可能被发现.也就是说,前面构件中的故障"掩盖"了后面构件中的故障.随着测试的进行,前面构件发生失效的概率越来越小,这样,后面构件中的故障就有机会被检测到.可以看出,集成测试剖面基于软件应用的运行剖面在集成测试过程中随着失效的发生和故障的排除不断发生变化,FSQ_i的到达速率随之动态变化.该动态变化过程具有非常复杂的非线性特点,利用解析模型难以描述,而仿真方法正是一种处理这类复杂问题的较好方法.

2.2 顾客服务规律

设随机过程$\{H_i(t),t\geqslant 0\}$表示FSQ_i的顾客离开过程,$H_i(t)$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$H_i(t)\leqslant N_i(t)$.$H_i(t)$的密度函数$U_i(t)$表示故障排除速率.$U_i(t)$依赖于故障到达速率$\Lambda_i(t)$、排错人员的数目以及每个排错人员的排错速率.

指数分布是常用的服务时间分布假设.设FSQ_i中每个排错人员的排错速率为μ_i,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

$$P(t\leqslant T\leqslant t+\Delta t\,|\,T>t)=\frac{g(t)\times\Delta t}{P(T>t)}=\frac{g(t)\times\Delta t}{1-G(t)}=u_i\times\Delta t \tag{4}$$

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

基于以上分析,FSQ_i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

$$\begin{cases}\Lambda_{il}(t)=\sum\nolimits_j \eta_j(t)\lambda_j(t_l),\ l=0,1,2,...\\ U_{il}=\begin{cases}l\mu_i, & 0\leqslant l<N\\ N\mu_i, & l\geqslant N\end{cases}\end{cases} \tag{5}$$

其中,$\Lambda_{il}(t)$和U_{il}分别表示FSQ_i处于状态l时的生速率和灭速率,即顾客到达速率和离开速率;t_l表示在系统处于状态l时构件j的累计执行时间.

FSQ_i队列系统的状态图如图2所示.

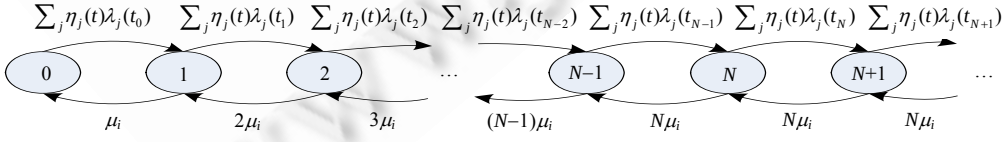


Fig.2 State diagraph of FSQ_i queueing system

图2 FSQ_i队列系统的状态图

3 仿真过程

构件软件的可靠性过程是一个复杂的随机过程,用解析的方法难以对HFSQM模型求解.基于HFSQM,本节用类C语言开发出一个仿真过程实现对构件软件可靠性过程的仿真.仿真过程的实现基于如下假设:

(1) 软件系统为由 n 个构件组成的顺序执行的终止应用,对软件应用执行基于其运行剖面的集成测试,所有构件失效都会引发系统失效.

(2) 构件失效过程满足 NHPP,所有故障之间相互独立,构件 i 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$\lambda_i(t)\Delta t$,多于1次失效发生的概率可以忽略.

(3) 排错是完全的,不会引入新故障,故障排除活动不会影响故障检测过程的继续进行;

(4) 故障排除过程用 HFSQM 排队模型描述,其中,每个 FSQ 建模为 G/M/N 排队系统,服务时间服从指数分布,第 i 个 FSQ 系统中每个排错人员的排错速率为μ_i,在充分小的时间区间$(t,t+\Delta t)$内,排错人员完成故障排除的概率为$\mu_i\Delta t$.

(5) 故障到达 HFSQM 后,如果有可用的排错人员,则立即分配;否则,进入队列排队等待.

基于以上假设,开发出基于率的仿真过程如图3所示.

```
1   int simulation_procedure(double dt,double t,double P[n][n],double phi[n][n],double (*lamda)[n](double),
2                            int policy[n],int level[k],double mu[k]){
3       int curr_comp, next_comp, total_faults_detect, total_faults_correct, faults_detect[n], faults_correct[n];
4       double time_this_visit, time_so_far, local_clock[n], global_clock; struct queue_info queue[k];
5       while (global_clock<t){
6           ALLOCATING:
7           for (int i=0; i<k; i++){
8               while (queue[i].working_server<level[i] && queue[i].head1!=null){
9                   head1→state=CORRECTING; head1→arrival_time=global_clock;
10                  tail0=head1; head1=head1→next; queue[i].working_server++;
11          }  }
12          DETECTING:
13          next_comp=determine_next_comp(P,curr_comp); time_this_visit=get_time_this_visit(phi[curr_comp][next_comp]);
14          while (time_so_far<time_this_visit){
15              time_so_far+=dt; global_clock+=dt; local_clock[curr_comp]+=dt;
16              if (occur(dt,lamda[curr_comp](local_clock[curr_comp]))){
17                  struct fault_info*ft=null; ft→arrival_time=global_clock; ft→comp=curr_comp; ft→state=WAITING;
18                  ft→next=null; int curr_queue=policy[curr_comp]; queue[curr_queue].total_arrival++;
19                  queue[curr_queue].tail1→next=ft; queue[curr_queue].tail1=ft;
20                  total_faults_detect++; faults_detect[curr_comp]++; curr_comp=n; break;
21          }  }
22          if (curr_comp==n) curr_comp=1; else curr_comp=next_comp; time_so_far=0;
23          CORRECTING:
24          for (int i=0; i<k; i++){
25              struct fault_info*f=queue[i].head0;
26              while (f!=queue[i].tail0→next){
27                  if ((f→state==CORRECTING) && (occur(dt,u[i]))){
28                      f→state=CORRECTED; f→departure_time=global_clock;
29                      queue[i].working_server--; total_faults_correct++; faults_correct[f→comp]++;
30                  }
31                  f=f→next;
32      } } }
33      return total_faults_correct;
34  }
```

Fig.3 Simulation procedure

图3 仿真过程

仿真过程返回到测试完成时总共修复的故障数.它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 $rate(t)\times dt<1$,$rate(t)$表示事件发生速率;集成测试时间 t;应用的运行剖面 $P[n][n]$.其中,$P[i][j](0<i,j\leqslant n)$表示构件 i 执行完成之后转移到构件 j 执行的概率.该参数可以通过咨询对该系统非常熟悉的专家得到,也可以从该应用以前发布的版本或者相似应用的历史运行数据中估计得到;每次访问构件时构件执行时间的期望值 $phi[n][n]$.其中,$phi[i][j](0<i,j\leqslant n)$表示构件 i 正常执行完成后控制转移到构件 j 时构件 i 的执行时间.由于集成测试时构件已经开发完成,执行时间可以通过实际测试得到;构件的失效速率 $(*lamda)[n]$,该参数根据构件单元测试数据建模得到;排错策略 $policy[n]$,每个 FSQ 系统中排错人员数 $level[k]$,排错速率 $mu[k]$.以上 3 个参数由实际的集成测试情况确定.除了输入参数以外,仿真过程还定义如下变量来控制仿真过程的执行:$curr\_comp$ 表示当前正在执行的构件;$next\_comp$ 表示下一个要执行的构件;$total\_faults\_detect$ 记录检测到构件软件应用发生的总故障数; $faults_detect[n]$累计检测到的每个构件的故障数;$time\_this\_visit$ 表示本次访问当前构件在不发生失效的情况下所需要执行的总时间;$time\_so\_far$ 表示当前构件在本次访问中已经执行的时间,它的值小于 $time\_this\_visit$;数组 $local_clock[n]$保存每个构件执行的总时间;变量 $global\_clock$ 记录测试已经执行的时间;$total\_faults\_correct$ 记录排除的总故障数;$faults_correct[n]$累计排除的每个构件的故障数;数组 $queue[k]$建模 HFSQM.

为了描述 HFSQM,我们定义两个结构体:$fault\_info$ 和 $queue\_info$,如图 4 所示.$fault\_info$ 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.从被检测出到被排除,故障一共经历 3 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.$queue\_info$ 建模一个 FSQ 排队系统.排队队列根据应用中故障被检测到的先后顺序组织而成,我们用链表数据结构表示.整条队列可分为头尾相接的两条队列,分别为排错队列和等待队列.$head0$ 指向排错队列的表头,表示占有排错资源的第 1 个故障;$tail0$ 指向排错队列的表尾,表示占有资源的最后一个故障.$head1$ 指向等待队列的表头,表示等待排错资源的第 1 个故障;$tail1$ 指向等待队列的表尾,表示等待资源的最后一个故障.两条队列满足 $tail0\rightarrow next=head1$.

```
struct fault_info{
    int comp;                   //the corresponding component
    double arrival_time;        //the time to occupy a resource
    double departure_time;      //the time to release a resource
    state;                      //the current status of the fault
    struct fault_info*next;     //the next detected fault
}
```

```
struct queue_info{
    int total_arrival;          //the cumulative number of faults
    int working_server;         //the number of occupied resource
    struct fault_info*head0;    //the first fault occupied the resource
    struct fault_info*tail0;    //the last fault occupied the resource
    struct fault_info*head1;    //the first fault waiting for the resource
    struct fault_info * tail1;  //the last fault waiting for the resource
}
```

Fig.4 Structure definition

图 4 结构体定义

接下来详细介绍仿真过程的具体实现.仿真过程中,每次执行采取的行动包括如下 3 步:

资源分配 ALLOCATING:这一步为 HFSQM 中等待排错资源的故障分配合适的资源.首先检查每个 FSQ 系统中有没有空闲的排错资源,以及等待队列中有没有故障在等待.如果条件满足,给等待队列队首的故障分配资源,并将其移到排错队列的队尾.以上过程重复进行,直到条件不满足为止.

故障检测过程 DETECTING:在测试过程中,基于软件应用的运行剖面对整个应用执行集成测试.在执行当前构件 $curr\_comp$ 之前,首先根据运行剖面 P 确定下一个要执行的构件 $next\_comp$,然后由 $curr\_comp$ 和 $next\_comp$ 确定 $curr\_comp$ 本次执行所需要的时间 $time\_this\_visit$.如果 $curr\_comp$ 执行过程中发生失效,则执行如下操作:将检测到的故障用结构体 $fault\_info$ 进行封装,然后根据排错策略放入相应的等待队列中,在下一个时间步到来之时参与排错资源分配;更新相应的计数器;设置 $curr\_comp$ 等于最后一个构件 n,以便在下一个时间步到来之时重启系统;中断对 $curr\_comp$ 的执行.可以看出,仿真过程考虑到了测试过程中随着失效的发生,测试剖面不断发生变化的问题.$curr\_comp$ 执行完成之后,检查 $curr\_comp$ 是否等于 n.有两种情况会使 $curr\_comp$ 等于 n:1) 终止应用一次执行正常完成;2) 发生失效.在这两种情况下都需要重新启动系统,可以通过设置 $curr\_comp$ 等于构件 1 来实现.

故障排除过程 CORRECTING:在这一步对 HFSQM 中占有排错资源的故障进行修复.定义变量 f 指向排错队列中的每个故障.如果 f 的状态是 CORRECTING,表示还未被排除,则执行修复.利用函数 *occur*()判断修复是否完成.如果修复完成,更改 f 的状态为 CORRECTED,释放排错资源,累加相应的计数器.如此重复,直到完成对 HFSQM 中所有排错队列的遍历.

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)、假设(4)可知,在时间间隔 dt 内,事件发生的概率近似为 $rate(t)\times dt$.因此,该函数首先生成一个 0~1 之间均匀分布的随机数 x,然后比较 $rate(t)\times dt$ 和 x.如果 $x<rate(t)\times dt$,则事件发生.

以上 3 个步骤重复进行,直到到达时间 t.最后,返回在这段时间内修复的总故障数.

4 实例分析

本节采用文献[12]中的应用作为示例,对上一节提出的仿真过程进行实验分析.该应用已经被广泛用来阐明基于体系结构的软件可靠性分析技术[8,13,14],应用的体系结构如图 5 所示.

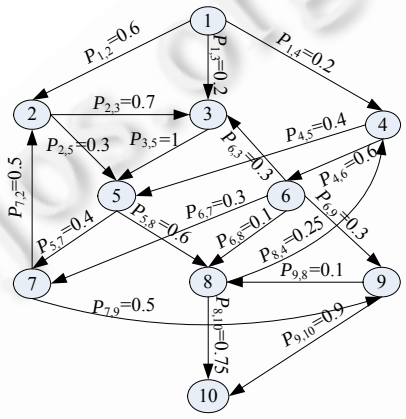


Fig.5 Architecture of an application

图 5 一个示例应用的体系结构

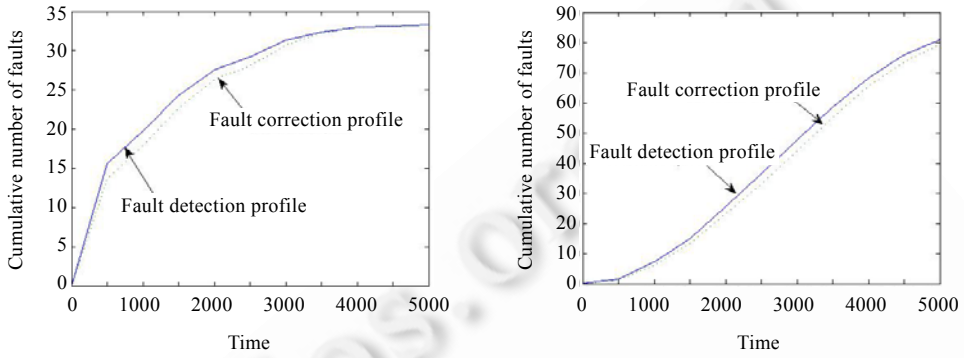
每个构件的失效行为用 Goel-Okumoto 软件可靠性增长模型[10]的失效速率来描述,则构件 i 的失效速率为 $\lambda_i(t_i)=a_ib_ie^{-b_it_i}$,其中,$a_i$ 表示最终可能从构件 i 中检测出的故障总数的期望值,b_i 表示每个故障的查出率,t_i 表示构件 i 的累计执行时间.不失一般性,设应用中所有构件满足 a_i=20.05,b_i=0.0057,每次访问构件时构件执行时间长度为 1 个时间单元.

设每个排错程序员的排错速率为 μ=0.025.该构件软件应用一共包括 10 个构件,根据集成测试中采用的排错策略不同, HFSQM 排队模型可能由 $k(0<k\leqslant 10)$个 FSQ 系统组成,存在 10 种建模方式.这里只选取 k=3 时的 HFSQM 模型进行分析,由这种一般性情况可以推广到其他各种情况.k=3 时,HFSQM 模型由 3 个 FSQ 系统组成:FSQ_1,FSQ_2,FSQ_3.设排错策略为构件 1 和构件 5 采用独立排错机制,检测到的构件 1 的故障进入 FSQ_1,构件 5 的故障进入 FSQ_2;其他构件采用共享排错机制,检测到的故障进入 FSQ_3.

设仿真过程中测试总时间为 5 000 个时间单元.首先不考虑排错资源的限制,分析 HFSQM 中各个 FSQ 系统的故障检测剖面和故障排除剖面.设仿真过程输入参数 $level=\infty$,执行仿真过程 2 000 次,计算得出平均运行剖面,如图 6 所示.

图 6(a)显示了 FSQ_1 中的两个剖面,也就是构件 1 的故障检测和排除剖面;图 6(b)显示了 FSQ_3 中的两个剖面,即除构件 1、构件 5 之外的其他 8 个构件的故障检测和排除剖面.可以看出,即使在排错人员数目无穷大的情况

下,两个剖面也是不同的.故障排除过程滞后于故障检测过程,因此在构件软件可靠性分析中不可以忽略故障排除过程.在测试区间[0,5000]上,FSQ_1 和 FSQ_3 的故障检测剖面曲线都呈现出先凸后凹的趋势,这表明故障到达系统的速率先增后减.FSQ_1 的凸区间比较短,在该区间内,到达 FSQ_1 的故障数目直线上升,而到达 FSQ_3 的故障数几乎为 0.这就是前面所提到的构件软件集成测试中的"掩盖"现象.构件 1 由于排在执行序列的第一位,在集成测试初期得到充分测试,它的故障"掩盖"了后面构件中的故障发生.随着测试的进行,构件 1 发生失效的概率越来越小,排队系统 1 的故障到达速率逐渐降低,这样,后面构件中的故障就陆续有机会被检测到,FSQ_3 中的故障到达速率逐渐增大.因为 FSQ_3 为 8 个构件的共享队列,构件遍布执行序列的前后,被"掩盖"的层次不一样,因此,该系统故障到达速率增长相对缓慢,凸区间比较长.

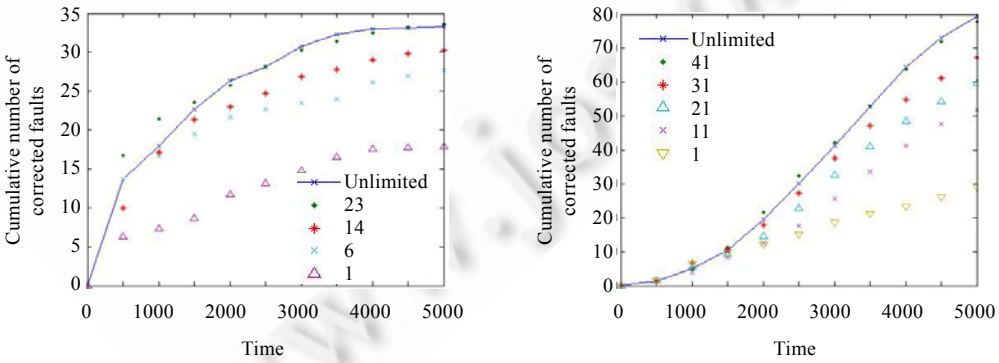


(a) FSQ_1 (b) FSQ_3

Fig.6 Fault detection profile and fault correction profile

图 6 故障检测剖面和故障排除剖面

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每个 FSQ 系统吞吐量的影响.设每个 FSQ 系统中排错人员数目分别为 1~100 以及无穷大.在这种资源配置条件下,分别执行仿真过程各 2 000 次,计算出每种情况下每个 FSQ 系统的平均故障排除剖面.对仿真结果进行分析发现,FSQ_1 和 FSQ_3 在排错人员数目分别为 23 和 41 时达到临界值.达到临界值后,再增加排错人员的数目不会提高排队系统的吞吐量.图 7 显示了 FSQ_1 和 FSQ_3 在不同排错人员数目约束条件下的故障排除剖面.从图中可以观察到,排队系统在临界值处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.而到达临界值之前,不同排错人员数目配置下的故障排除剖面是不同的.

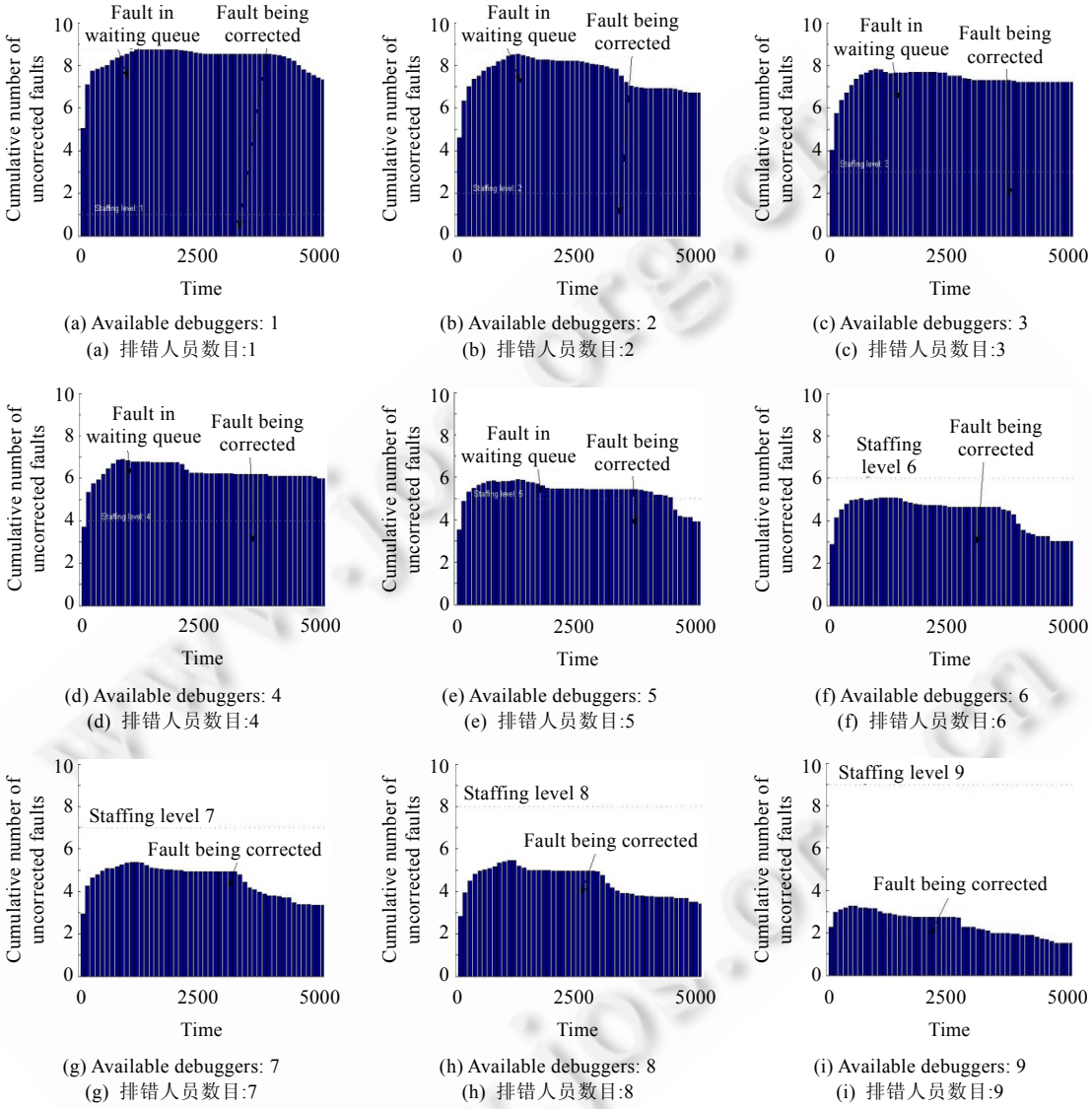


(a) FSQ_1 (b) FSQ_3

Fig.7 Fault correction profile

图 7 故障排除剖面

为了更形象地说明排错人员数目对 FSQ 系统服务水平的影响,图 8 列出了排错人员数目为 1~9 时 FSQ_1 中未排除故障数随时间的变化情况.未排除故障包括等待队列中未分配排错资源的故障和排错队列中占有资源还未被排除的故障.从图中可以清楚地观察到在不同的约束条件下,排错人员利用率剖面以及队列长度剖面.随着排错人员数目的增加,队列长度逐渐缩短,排错系统效率提高,但排错人员的利用率逐渐下降.表 1 列出了[0,5000]测试区间内,FSQ_1 和 FSQ_3 中排错人员的利用率统计值.



(a) Available debuggers: 1 (a) 排错人员数目:1
(b) Available debuggers: 2 (b) 排错人员数目:2
(c) Available debuggers: 3 (c) 排错人员数目:3
(d) Available debuggers: 4 (d) 排错人员数目:4
(e) Available debuggers: 5 (e) 排错人员数目:5
(f) Available debuggers: 6 (f) 排错人员数目:6
(g) Available debuggers: 7 (g) 排错人员数目:7
(h) Available debuggers: 8 (h) 排错人员数目:8
(i) Available debuggers: 9 (i) 排错人员数目:9

Fig.8 Number of unrepaired faults in FSQ_1 versus time

图 8 FSQ_1 中未排除故障数随时间的变化

Table 1 Debugger utilization in FSQ_1 and FSQ_3 (%)

表 1 FSQ_1 和 FSQ_3 中排错人员利用率 (%)

Staffing level	1	2	3	4	5	6	7	8	9
FSQ_1	100	100	100	99.85	97.22	72.62	64.55	55.77	26.98
FSQ_3	97.90	96.33	95.98	91.86	86.93	83.56	79.59	78.07	72.18

从图 8 可以明显发现,排错系统 1 在排错人员数目不小于 6 时,平均运行剖面显示等待队列中已经没有故障.因此,FSQ_1 在排错人员数目等于 6 时虽然没有达到临界值,但是已经满足系统平均故障检测剖面,此时再分配更多的人员不会显著提高系统的吞吐量.以图 8(e)为例可以观察到,在测试开始时到达 FSQ_1 的故障数比较少,故障到达后能够立即分配排错人员进行排除,排错系统效率比较高,但排错人员的利用率低;随着测试进行,故障到达速率逐渐增大,排错人员利用率随之增加,但等待队列中的故障也迅速增多,排错系统效率降低.此时,项目管理者需要决定在什么时候如何调整排错人员的数目来实现排错系统效率和排错资源利用率之间的有效平衡.在选定的时间点,以新配置的排错人员数目为参数,重新执行仿真过程,根据仿真结果,项目管理者可以预测到重新进行资源配置对测试的影响,进一步决定是否要进行资源重新配置.

仿真过程的仿真结果准确地描述了构件软件应用的可靠性过程,预测构件软件可靠性随测试过程的增长.这能够为软件发布和测试资源的合理配置提供依据,使软件既能按计划准时发布,又能满足用户的可靠性需求,降低软件的总成本;反之,则会给软件企业和用户造成经济和信誉上的损失.因此,仿真过程的仿真结果对于构件软件测试,进一步对于软件市场决策具有重要的指导意义.

5 结束语

本文从描述构件软件实际的可靠性过程出发,提出了一个仿真框架.针对目前已经提出的仿真方法对构件软件集成测试中的故障排除过程的忽略和简化,该方法用一个混合的有限服务排队模型来建模故障排除过程,并考虑了排错策略和排错资源的约束性问题.由于该仿真方法放宽了传统方法中一些过于严格的假设,全面考虑了构件软件集成测试所包括的两个随机过程:故障检测过程和故障排除过程,因此它更符合构件软件的测试特征,可以更准确地描述构件软件的可靠性过程,预测构件软件可靠性随测试过程的增长.

References:

[1] Wang WJ, Hemminger TL, Tang MH. A moving average non-homogeneous Poisson process reliability growth model to account for software with repair and system structures. IEEE Trans. on Reliability, 2007,56(3):411−421. [doi: 10.1109/TR.2007.903119]

[2] Mao XG, Deng YJ. A general model for component-based software reliability. Journal of Software, 2004,15(1):27−32 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/27.htm

[3] Popstojanova KG, Trivedi KS. Architecture-Based approach to reliability assessment of software systems. Performance Evaluation, 2001,45(2−3):179−204. [doi: 10.1016/S0166-5316(01)00034-7]

[4] Cai KY, Bai CG, Zhong XJ. Introduction to reliability models of component-based software system. Journal of Xi'An Jiaotong University, 2003,37(6):551−564 (in Chinese with English abstract).

[5] Tausworthe RC, Lyu MR. A generalized technique for simulating software reliability. IEEE Software, 1996,13(2):77−88. [doi: 10.1109/52.506464]

[6] Gokhale SS, Lyu MR, Trivedi KS. Incorporating fault debugging activities into software reliability models: A simulation approach. IEEE Trans. on Reliability, 2006,55(2):281−292. [doi: 10.1109/TR.2006.874911]

[7] Lin CT, Huang CY. Staffing level and cost analyses for software debugging activities through rate-based simulation approaches. IEEE Trans. on Reliability, 2009,58(4):711−724. [doi: 10.1109/TR.2009.2019669]

[8] Gokhale SS, Lyu MR. A simulation approach to structure-based software reliability analysis. IEEE Trans. on Software Engineering, 2005,31(8):643−656. [doi: 10.1109/TSE.2005.86]

[9] Dohi T, Osaki S, Trivedi KS. An infinite server queueing approach for describing software reliability growth: Unified modeling and estimation framework. In: Proc. of the 11th Asia-Pacific Software Engineering Conf. (APSEC 2004). Busan: IEEE Computer Society, 2004. 110−119. [doi: 10.1109/APSEC.2004.29]

[10] 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. [doi: 10.1109/TR.2007.909777]

[11] Huang CY, Hung TY, Hsu CJ. Software reliability prediction and analysis using queueing models with multiple change-points. In: Proc. of the 3rd IEEE Int'l Conf. on Secure Software Integration and Reliability Improvement. Shanghai: IEEE Computer Society, 2009. 212−221. [doi: 10.1109/SSIRI.2009.11]

[12] Cheung RC. A user-oriented software reliability model. IEEE Trans. on Software Engineering, 1980,6(2):118−125. [doi: 10.1109/TSE.1980.234477]

[13] Popstojanova KG, Kamavaram S. Assessing uncertainty in reliability of component-based software systems. In: Proc. of the 14th Int'l Symp. on Software Reliability Engineering (ISSRE 2003). Denver: IEEE Computer Society, 2003. 307−320. [doi: 10.1109/ISSRE.2003.1251052]

[14] Lo JH, Kuo SY, Lyu MR, Huang CY. Optimal resource allocation and reliability analysis for component-based software applications. In: Proc. of the 26th Annual Int'l Computer Software and Applications Conf. (COMPSAC 2002). Oxford: IEEE Computer Society, 2002. 7−12. [doi: 10.1109/CMPSAC.2002.1044526]

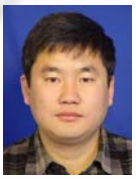
附中文参考文献:

[2] 毛晓光,邓勇进.基于构件软件的可靠性通用模型.软件学报,2004,15(1):27−32. http://www.jos.org.cn/1000-9825/15/27.htm

[4] 蔡开元,白成刚,钟小军.构件软件系统的可靠性评估模型简介.西安交通大学学报,2003,37(6):551−564.

侯春燕(1980－),女,山东威海人,博士生,主要研究领域为软件测试,软件可靠性评估,容错计算.

崔刚(1947－),男,教授,博士生导师,CCF高级会员,主要研究领域为容错计算技术,嵌入式系统的研究及设计.



刘宏伟(1971－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为容错计算,移动计算.