HIGH TECHNOLOGY LETTERS | Vol. 20 No. 1 | Mar. 2014 | pp. 9 ~ 15

doi:10.3772/j.issn.1006-6748.2014.01.002

Component-based software reliability process simulation considering imperfect debugging①

Zhang Ce (张 策)②*,**, Cui Gang*, Bian Yali**, Liu Hongwei*

(*School of Computer Science and Technology, Harbin Institute of technology, Harbin 150001, P. R. China)

(**School of Computer Science and Technology, Harbin Institute of Technology at Weihai, Weihai 264209, P. R. China)

Abstract

In view of the flaws of component-based software (CBS) reliability modeling and analysis, the low recognition degree of debugging process, too many assumptions and difficulties in obtaining the solution, a CBS reliability simulation process is presented incorporating the imperfect debugging and the limitation of debugging resources. Considering the effect of imperfect debugging on fault detection and correction process, a CBS integration testing model is sketched by multi-queue multichannel and finite server queuing model (MMFSQM). Compared with the analytical method based on parameters and other nonparametric approaches, the simulation approach can relax more of the usual reliability modeling assumptions and effectively expound integration testing process of CBS. Then, CBS reliability process simulation procedure is developed accordingly. The proposed simulation approach is validated to be sound and effective by simulation experiment studies and analysis.

Key words: software reliability growth model (SRGM), component-based software (CBS), imperfect debugging, reliability simulation, queuing theory

0 Introduction

At present, component-based software (CBS) has become a kind of mainstream software form, been widely used in all kinds of mission-critical systems, and its reliability problems get greater attention. Software reliability can be effectively measured and predicted by software reliability growth models (SRGMs)[1,2].

From the viewpoint of system structure, CBS reliability model can be classified into three broad categories[2,3]: state-based, path-based, and additive model. Reliability analysis approaches based on architecture will usually involve the following problems:

(1) Ignore much information of the actual software development process, e. g., imperfect debugging[4-7], infinite debugging resources, and so on.

(2) Can't describe the reliability growth with increasing time as conventional existing non-homogeneous Poisson process (NHPP) software reliability growth models.

(3) With the tendency to establish a mathematical model, often for too many hypothesis, the proposed model can't be solved well by the analytical method when the research problem becomes complex, resulting in bigger deviation.

In recent years, using simulation approaches to research SRGM has increased gradually[7-9]. Simulation approaches to software reliability modeling relaxing hypothesis conditions of modeling based on analytical parameter modeling can effectively simulate the stochastic process of software testing, and could be used to measure and predict application reliability in each phase of its life cycle. Rate-based simulation (RBS) research thinks that the biggest difference between the SRGMs is the diversities of failure rate function $\lambda(t)$[9-12], so it can be used to describe varied SRGMs and perform imperfect debugging process analysis. For instance, the author in Ref. [11] proposed a simulation approach to give perfect debugging of CBS, considering the limitation of debugging resources, but not the situation of imperfect debugging. Gokhale also tried to rely on the simulation to evaluate the CBS reliability, a positive practice to simulation analysis of CBS reliability[13], covering the component-level and the application-level stochastic testing process simulation, considering sequence dependent repair and fault-tolerant configuration of critical components. However, due to the lack of consideration in imperfect debugging and

① Supported by the National High Technology Research and Development Program of China (No. 2008AA01A201) and the National Nature Science Foundation of China (No. 60503015, 90818016).

② To whom correspondence should be addressed. E-mail: zhangce@hitwh.edu.cn

Received on Sep. 18, 2012

limited debugging resources, the model proposed in Ref. [13] has a significant gap with the actual situation.

In this paper, we propose an approach of imperfect debugging of CBS reliability process simulation, which can describe fault detection process (FDP) and fault correction process (FCP) of CBS integration testing and consider the problems of imperfect debugging and debugging resources limitation in the process of debugging.

1 Imperfect debugging MMFSQM

Research into fault debugging activities using the queuing has increased gradually[9,13], and there is a corresponding relationship in concepts between the queuing and SRGMs: Customers⇔Faults and Servers ⇔Debuggers. Considering imperfect debugging, the faults detected but not allocated debugging resources will enter the waiting queues, while faults detected will be repaired and leave the queuing system when debugging resources are idle. Huang[7] utilized finite/infinite service queuing-perfect/imperfect debugging modeling approaches to conduct reliability process simulation, achieving an accurate prediction performance. However, the drawback is lack of consideration of the debugging resources limitation. Later, Lin[8,9] proposed a single queue multi-service channels model based on the simulation to illustrate perfect and imperfect debugging, including fault detecting and correcting process. The approach elaborated the relation of fault detection and correction profile considering the limitation of debugging resources, but does not apply to the CBS reliability analysis in integration testing.

Actually, infinite service queue is not realistic, as debugging resources, e.g., debuggers and testing efforts are limited in real software testing. Hereon, we present a multi-queue multichannel and finite server queuing model (MMFSQM) to sketch the CBS debugging system, and consider imperfect debugging and finite debugging resources as well, as shown in Fig. 1 below.

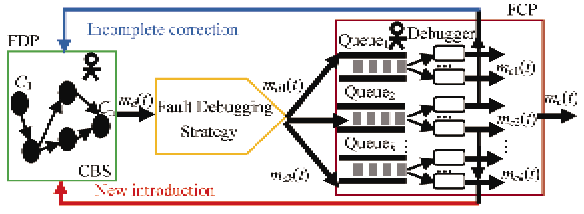

Fig. 1 Multi-queue multichannel and finite server queuing model considering imperfect debugging

Fig. 1 shows the CBS integration testing based on its operational profile, which is a grey box testing model. Faults belonging to component C_i enter fault repair queue FRQ_j according to the fault debugging strategy (FDS). FCP allocates appropriate debugging resources to faults to be repaired based on FDS. The debugging resources herein are mainly the debuggers of each queue. Furthermore, due to the existence of imperfect debugging, there are feedback relations between FDP and FCP.

For the sake of explicit illustration, herein, we characterize reliability modeling of component C_i based on the G-O model[14]. Considering the simplicity of modeling, we make the following assumption as that of many literatures[4, 8,15]:

(1) Let $\{N_i(t), t \geqslant 0\}$ be a counting process representing the cumulative failures number of component C_i with the mean value function $m_i(t)$ in the time interval $(t, t + \mathrm{d}t)$.

$$\begin{cases} P\{N_i(t+\mathrm{d}t) - N_i(t) = 0\} = 1 - \lambda_i(t)\mathrm{d}t + o(\mathrm{d}t) \\ P\{N_i(t+\mathrm{d}t) - N_i(t) = 1\} = \lambda_i(t)\mathrm{d}t + o(\mathrm{d}t) \\ P\{N_i(t+\mathrm{d}t) - N_i(t) \geqslant 2\} = o(\mathrm{d}t) \\ \lim\limits_{\mathrm{d}t \to 0^+} o(\mathrm{d}t)/\mathrm{d}t = 0 \end{cases} \tag{1}$$

where $\lambda_i(t)$ is the failure intensity of C_i at t, i.e., failure rate in a unit time. Eq. (1) means the failure probability of C_i is approximately $\lambda_i(t)\mathrm{d}t$ in a small $\mathrm{d}t$. In addition, failure occurring at the same time two or more times is small probability of pieces, which could be ignored.

(2) Fault detection rate is proportional to the mean number of remaining faults in system, proportion functions is $b_i(t)$.

(3) Fault correction is not complete, i.e., fault is repaired successfully with $p_i(t)$.

(4) New fault may be introduced, and probability of introduction is proportional to the number of faults detected at t, proportion functions is $r_i(t)$.

According to the assumptions above, the following differential equations can be derived as

$$\begin{cases} \dfrac{\mathrm{d}m_i(t_i)}{\mathrm{d}t} = b_i(t_i)[a_i(t_i) - c_i(t_i)] \\ \dfrac{\mathrm{d}c_i(t_i)}{\mathrm{d}t} = p_i(t_i)\dfrac{\mathrm{d}m_i(t_i)}{\mathrm{d}t} \\ \dfrac{\mathrm{d}a_i(t_i)}{\mathrm{d}t} = r_i(t_i)\dfrac{\mathrm{d}c_i(t_i)}{\mathrm{d}t} \end{cases} \tag{2}$$

where t_i is the execution time of component C_i during integration testing. Eq. (2) boundary conditions for: $m_i(0) = 0$, $a_i(0) = a_{i0}$, $c_i(0) = 0$. Solving the above differential equations with the assumption (1)-(4) yields

$$m_i(t_i) = a_{i0}\int_0^t b_i(x)\left[1 - \frac{\int_0^x (1 - r_i(u))p_i(u)b_i(u)}{e^{-\int_0^u (1-r_i(\tau))p_i(\tau)b_i(\tau)\mathrm{d}\tau}}\mathrm{d}u\right]\mathrm{d}x \tag{3}$$

$$\lambda_i(t_i) = \frac{\mathrm{d}m_i(t_i)}{\mathrm{d}t} = a_{i0}b_i(t_i)\left[1 + \int_0^{t_i}\left[\frac{(r_i(u)-1)p_i(u)b_i(u)}{e^{-\int_0^{t_i}(1-r_i(\tau))p_i(\tau)b_i(\tau)\mathrm{d}\tau}}\right]\mathrm{d}u\right] \quad (4)$$

So, fault arrival rate γ_j of queue FRQ_j is

$$\gamma_j(t_i) = \sum_{\forall C_i \in FRQ_j}\xi_i(t)\lambda_i(t_i) = \sum_{\forall C_i \in FRQ_j}\xi_i(t)a_{i0}b_i(t_i)\left[1 + \int_0^{t_i}\left[\frac{(r_i(u)-1)p_i(u)b_i(u)}{e^{-\int_0^{t_i}(1-r_i(\tau))p_i(\tau)b_i(\tau)\mathrm{d}\tau}}\right]\mathrm{d}u\right] \quad (5)$$

where $\xi_i(t)$ is the execution probability of C_i in queue FRQ_j. As failure process of each component follows the NHPP model, the accumulation of more components failure process follows the same.

Generally speaking, fault correction time follows exponential distribution[7], so let T_c be the correction time of a debugger in queue FRQ_j, then $C(t_c)$ can be given as the distribution function of T_c:

$$C(t_c) = P(T_c \leqslant t_c) = 1 - e^{-\mu_j \times t_c},\ t_c > 0 \quad (6)$$

The probability of fault correction in the time interval $(t_c, t_c + \mathrm{d}t)$ after correction time t_c is

$$P(t_c \leqslant T_C \leqslant t_c + \mathrm{d}t \mid T_C > t_c) = \frac{P(t_c \leqslant T_C \leqslant t_c + \mathrm{d}t)}{P(T_C > t_c)} = \frac{\left(\frac{\mathrm{d}C(t_c)}{\mathrm{d}t_c}\right)\times \mathrm{d}t}{P(T_C > t_c)} = \mu_j \times \mathrm{d}t \quad (7)$$

So, the repair rate (service rate) of MMFSQM can be derived as

$$L_j^k(t) = \begin{cases} k\mu_j, & 0 \leqslant k < N \\ N\mu_j, & N \leqslant k \end{cases} \quad (8)$$

The queue system proposed can be expressed as $M/M/N$ model with arrival rate and repair rate given by Eqs(5) and (8). Obviously, its birth rate is non-stationary. Considering fault correction probability $p_i(t)$ and fault introduction probability $r_i(t)$, the model proposed can reflect the nature of actual FDP and FCP more accurately, so arrival rate in Eq. (5) is more complicated than that in Ref. [11].

2 CBS simulation procedure considering imperfect debugging

According to the meaning of imperfect debugging and MMFSQM in Section 1, due to non-stationary, using analytical method based on parameter to conduct CBS reliability process analysis will become extremely complicated. Comparatively, simulation approach may provide more effective measures.

On the assumptions in Section 1, assume that CBS integration test profile and operational profile are identical, CBS S consists of n components, each time, S runs from C_1 and ends with C_n; the failure probability of $C_i(1 \leqslant i \leqslant n)$ is the same and S termination can be triggered by the failures in $C_i(1 \leqslant i \leqslant n)$ and then be restarted immediately. Pseudo codes of simulation process CBS _ SIMPRO _ CONIMPDEBUG developed are shown in Fig. 2 below.

```
1:  CBS_SIMPRO_CONIMPDEBUG(double time_limited, double dt, int debugger[k], double P[n][n], double phi[n][n], double (*lamda)[n](double), double mu[n], double exposing_rate, double introduction_rate ) {
2:  while (global_clock<time_limited) {
3:  ALLOCATING:
4:
5:  for (i = 0; i < k; i++) {
6:  while((length(C[i])+length(R[i]))<debugger[i]&&W[i]!=NULL){
    newfault=get_from_queue(W[i]); put_into_queue(C[i],fault);}}
7:  DETECTING:
8:  next_comp=determine_next_comp(cur_comp, P);
9:  whole_time_this_access=get_whole_time_this_access(cur_comp,
10: phi[cur_comp][next_comp]);
11: while (time_so_far<whole_time_this_access&&failed!=1){
12: time_so_far+=dt;local_clock[cur_comp]+=dt;global_clock+=dt;
13: if (occur(dt, lamda[cur_comp](local_clock[cur_comp]))) {
14:     new fault_detected=encapsulate(DETECTED);
15:     int cur_queue=correcting_strategy(cur_comp);
16:     if ((length(C[cur_queue])+length(R[cur_queue]))==debug-
    ger[cur_queue]){fault_detected→state=WAITING; put_into_queue
    (W[cur_queue],
17: fault_detected);}
18:     else {fault_detected→state=CORRECTING;
19:     put_into_queue(C[cur_queue], fault_detected)}
20: failed=1; whole_fault_detected++; fault_detected[cur_comp]++;
21:     time_so_far=0; this_run[cur_queue]; break;}}
    if(cur_comp==n) cur_comp=1; else cur_comp=next_comp;
22: EXPOSURING:
23: for (i=0; i<k; i++){
24: while(fault_introduced=get_from_queue(I[i])){
25:
26:     if(expose(exposing_rate, dt)){I[i]=I[i]-fault_introduced;
        if((length(C[i])+length(R[i]))==debugger[i])put_into
        _queue(W[i],
27: fault_introduced);
        else put_into_queue(C[i], fault_introduced);}}}
28: CORRECTING:
29: for(i=0; i<k; i++){
30: if (this_run[i]!=1){
31:   while(fault=get_from_queue(R[i])){
32:     if (fault→state==CORRECTING&&correct(dt, mu[i])){
33:     R[i]=R[i]-fault;F[i]=F[i]+fault;fault→state=CORRECTED;
        fault→leaving
34: _time=global_time;
35:     whole_fault_corrected++;fault_corrected[fault→comp]++;}}
        this_run[i]=0;}}
36: INTRODUCING:
37: for (i=0; i<k; i++){ if (correct(dt, mu[i]) )
38: { if introduce(introduction_rate) {
39:   new fault_introduced=encapsulate(INTRODUCED);put_into_queue
    (I,fault_introduced); }}}
40: for(i=0; i<k; i++){R[i]=R[i]+C[i]; C[i]=NULL;}}
```

Fig. 2 Simulation process—CBS _ SIMPRO _ CONIMPDEBUG

Considering imperfect debugging and limitation of debugging resources, simulation procedure accepts the following input parameters: *time _ limited*, the upper limit of total integration test time; dt, the duration of each run time step denoted, and dt must be set small enough. So , at most, one failure occurs at any given dt, that is, in the time interval $(t, t+\mathrm{d}t)$, $\lambda(t)$ keeps stable; $P[i][j]$ and $phi[i][j]$ $(1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant n)$ reflect the CBS S operational profiles, which represent respectively inter-component transition probabilities among the components and mean execution time proportion of C_i that the transition (C_i, C_j) is taken after the execution of C_i; k denotes the number of queues which is determined by the debugging strategy; array *debugger*$[k]$ means the number of debugger in queue k; $(*lamda)[i]$ and $(*mu)[i]$ denote failure rate and repair rate of C_i respectively. *Introduction _ rate* and *exposing _ rate* are fault introduction rate and detection rate.

CBS _ SIMPRO _ CONIMPDEBUG is composed of 5 parts:

(1) ALLOCATING: $W[i]$ records the number of faults detected but not repaired. Fault is taken out orderly from $W[i]$ and put into $C[i]$ which keeps the faults allocated idle and available debuggers in the current time step, if the present debuggers of queue i are idle and $W[i]$ is not null.

(2) DETECTING: This part corresponding to the faults detection based on the operational profile of CBS S is to determine whether a new fault occurs during the current component execution and to decide how to deal with the detected fault according to the present debugging resources. Four terms are of particular interest here: ① Firstly, get the execution time *whole _ time _ this _ access* of current component (*cur _ comp*) and next component *next _ comp* that will execute; ② For the fault detected, determine the queue *cur _ queue* to which fault belongs, and decide the debuggers of *cur _ queue* whether are idle or not, if there are available debugger the detected fault is put into $W[i]$ else $C[i]$. ③ Update the counting variables; ④ Hereon, utilize *occur*( ) function to determine whether a fault is detected by comparing a random number x in (0,1) with $\lambda_i(t)\mathrm{d}t$ of current component[9,13]: if $x < \lambda_i(t)\mathrm{d}t$, then a fault is detected.

(3) CORRECTING: This part accomplishes correction of the detected fault. ① Considering the premise of imperfect debugging, the fault detected in current time step dt can't be repaired instantly, the faults detected in previous time step can only be repaired. So, in DETECTING part, *this _ run*$[i]$ variable is marked for this intention, and judged in this step; ② Array $R[i]$ records the faults to which debugging resources are allocated in the current step before, $F[i]$ stores the corrected faults; ③ Implementation method of fault correction and detection process is identical, i. e. , using *correct*( ) function.

(4) INTRODUCING: This part mainly handles this situation of introducing new faults in the process of imperfect debugging. ① As the precondition of introducing new faults is successful fault correction in CORRECTING part, *correct*( ) is used to determine whether the fault is corrected or not; ② Using *introduce*( ) to decide the new introduced fault. *introduce*( ) and *correct*( ) are realized in the same method, that is, judge random number $x <$ *introduction _ rate*; ③ The introduced faults are put into $I[i]$.

(5) EXPOSURING: This part detects and processes the faults introduced in the previous dt, so we need to determine whether $I[i]$ is null or not. ① Detection rate of inherent faults in system and that of new introduced faults are different, so *expose*( ) is used to realize this function other than *occur*( ) in DETECTING part; ② When *expose*( ) is ture and debuggers are available then *fault _ introduced* is put into $W[i]$ else $C[i]$.

By the analysis above, *fault _ introduced* will move complying with the following processes: *fault _ introduced* $\rightarrow I[i] \rightarrow W[i] \rightarrow C[i] \rightarrow R[i] \rightarrow F[i]$, which is the most complete process of dealing with new introduced faults.

3 Simulation and validation

In this section, experimental studies are conducted and the potential of the simulation procedures developed in Section 1 through the CBS reported in Ref. [16] are demonstrated, which has been widely used to illustrate structure-based reliability process assessment techniques in recent years[11, 13]. Fig. 3 presents the structure of the CBS application and Table 1 shows transition probabilities among the components.

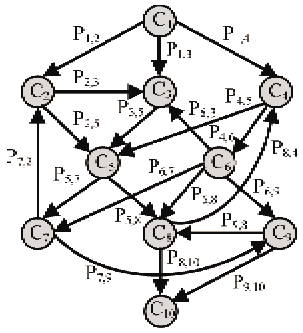

Fig. 3 Structure of the component-based software application

Table 1 Transition probabilities among the components

$P_{1,2}=0.60$	$P_{1,3}=0.20$	$P_{1,4}=0.20$	$P_{2,3}=0.70$	$P_{2,5}=0.30$
$P_{3,5}=1.00$	$P_{4,5}=0.40$	$P_{4,6}=0.60$	$P_{5,7}=0.40$	$P_{5,8}=0.60$
$P_{6,3}=0.30$	$P_{6,7}=0.30$	$P_{6,8}=0.10$	$P_{6,9}=0.30$	$P_{7,2}=0.50$
$P_{7,9}=0.50$	$P_{8,4}=0.25$	$P_{8,10}=0.75$	$P_{9,8}=0.10$	$P_{9,10}=0.90$

In consideration of imperfect debugging, components in Fig. 3 are represented by Eq. (2). Since there are no unit testing failure data of $C_i(1 \leqslant i \leqslant n)$, parameters of $\lambda_i(t_i)$ can't be estimated. The main feature of interest here is integration testing process, so component $C_i(1 \leqslant i \leqslant n)$ can be set to meet research requirements: $a_i = 35.14$, $b_i = 0.0086$. Repair rate is a result of multiple factors including debugger proficiency, testing environment and other factors, and is unable to be accurately estimated, so is set $\mu[j] = 0.035$. Degree of imperfect debugging is determined by fault introduction rate, without loss of generality, here set *introduction _ rate* = 0.15 and *exposing _ rate* = 0.035. Ref. [13] has concluded that C_1 and C_5 are critical components, and that as a result the following strategy is employed: set three correction queues, FRQ_1、FRQ_2 and FRQ_3, faults detected in C_1 and C_5 are put into FRQ_1 and FRQ_2 respectively, the other faults of components into FRQ_3. It is assumed that the simulation procedure executes 5,000 time units and spends 1.00 time unit in each component per visit. The simulation procedure CBS _ SIMPRO _ CONIMP-DEBUG was executed 1,000 times and an average as statistical data of the profile obtained during each run was computed.

The fault profile of each queue is analyzed at first. Considering imperfect debugging, Fig. 4 shows the fault profiles of FRQ_1 、FRQ_2 and FRQ_3 including

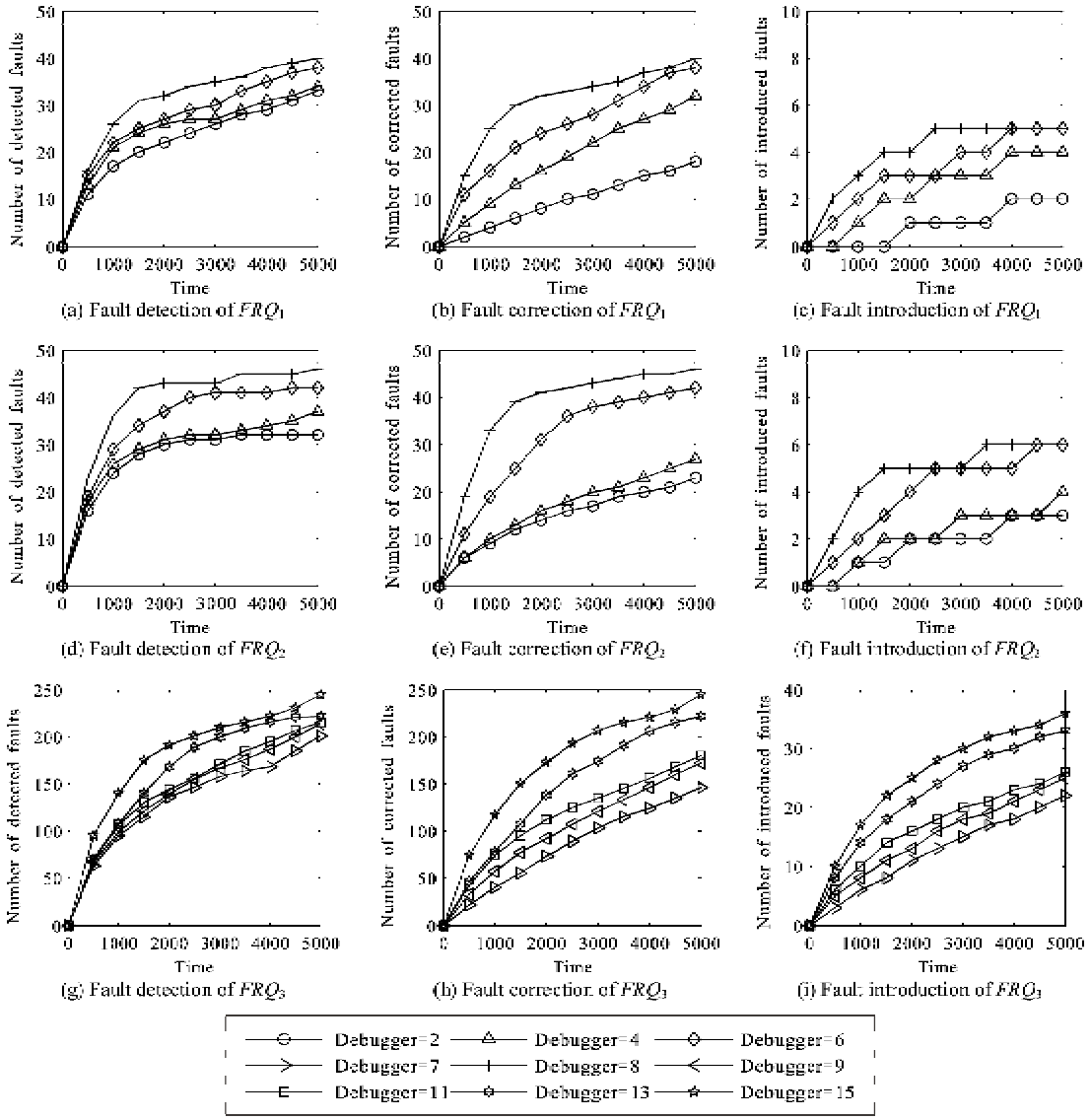

Fig. 4 Fault profiles of FRQ_1, FRQ_2 and FRQ_3

fault detection profile, correction profile and introduction profile, particularly, the detected or corrected faults encompass inherent faults in application and newly introduced faults. As can be seen from (b), (e) and (h), the number of faults corrected and the number of debuggers is the positive relation, i. e., the detected fault can be more corrected as the number of debugger increases in a continuous time slice. Meanwhile, looking at (a)、(d) and (g), there has also been a same positive relation in the number of detected faults and debuggers, mainly due to the existence of fault introduction rate. Considering imperfect debugging, the increasing debuggers cause the growth of the number of corrected and introduced fault, and ultimately causes the growth of number of detected faults.

As we can see from Fig. 4, fault correction profiles of FRQ_1、FRQ_2 and FRQ_3 lag behind detection profiles with the tendency of remaining a wide gap at the initial and narrowed or overlapped at last. In CBS _ SIMPRO _ CONIMPDEBUG, in order to satisfy the assumption of considering imperfect debugging, variable of *this _ run*[*i*] needs to be judged in the CORRECTING part, so the faults only detected in a previous time step *dt* can be corrected, leading to hysteresis phenomenon. It is tally with the actual situation: due to existence of correction time, faults newly detected can't be corrected instantaneously, debuggers can only correct faults detected formerly, and the number of detected faults is more than that of corrected faults. In particular, if debugging personnel allocation is reasonable, then all faults detected will be completely corrected.

Besides that, due to imperfect debugging, new faults can be introduced as the testing. In Fig. 4(c), (f) and (i), note that the number of new introduced faults is positively correlated to the number of debuggers. It is easy to understand that the more debuggers the more corrected faults, causing more introduced faults. Table 2 summarizes the correspondence between the number of introduced faults in FRQ_1, FRQ_2 and FRQ_3 and the number of debuggers. The numbers of the new introduced faults are 5, 6 and 36 respectively which are the result of current *introduction _ rate* = 0.15. For other settings, simulation results also reflect the same positively relation and limitations of space, so that won't be covered again here.

Table 2 Number of fault introduced in FRQ_1, FRQ_2 and FRQ_3

FRQ_i	Number of debuggers								
	2	4	6	7	8	9	11	13	15
FRQ_1	2	4	5	—	5	—	—	—	—
FRQ_2	3	4	6	—	6	—	—	—	—
FRQ_3	—	—	—	22	—	25	26	33	36

Next, to illustrate the effect of debuggers' number allocation on simulation result, Fig. 5 depicts the relationship between faults in waiting queue of FRQ_1、FRQ_2 and FRQ_3 and debuggers. It's obvious, the number of remaining faults in waiting queue can reflect whether debugging staffing is reasonable or not. As can be seen from the Fig. 5, with the increasing of debuggers, the number of faults in waiting queues has since been falling steadily. For FRQ_1, when the debugging staffing expands to 6 persons, there are no faults in waiting queue. This means that 6 debuggers can meet debugging requirements, i. e., the correction profile can be appropriate to the detection profile and additional number of debuggers will not have significant effect on debugging throughput rate and efficiency. Likewise, the number of debugger of FRQ_2 is also 6. By contrast, simulation results from Fig. 6(c) shows that it is 13 persons needed for FRQ_3, which can be interpreted as,

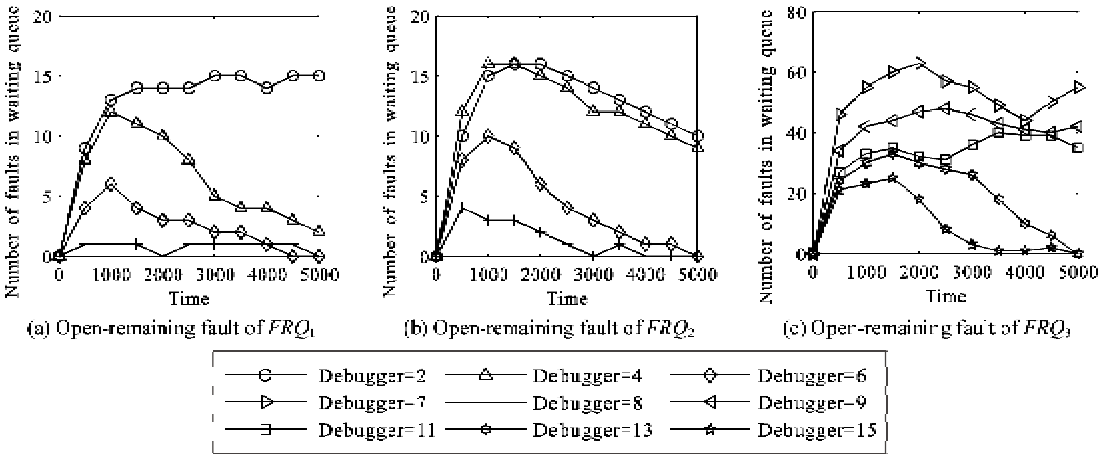

Fig. 5 Open-remaining fault profile

FRQ_3 contains 8 components and has the maximum inherent faults together with newly introduced faults. The number of debugger given here is just the minimum, due to considering imperfect debugging, i. e. , the fault correction probability is not 100% and fault introduction rate is not 0.

In accordance with the results of simulation experiment, CBS _ SIMPRO _ CONIMPDEBUG simulation procedure conducts the CBS integration testing process reliability analysis effectively, and describes imperfect debugging activities accurately covering the whole of fault detection, correction, introduction and explosion. In particular, in a practical application, software engineer can model and simulate CBS testing process with proper parameters, evaluate and predict CBS reliability, and determine an optimal allocation strategy of testing resources based on assistant supports provided by simulation procedures.

4 Conclusions

The major contribution of this paper is that we present an approach of CBS reliability simulation that elaborates the effect of imperfect debugging on FDP and FCP, and illustrates intrinsic relations between imperfect debugging and the number of debuggers. Compared with the approaches available, the proposed approach sketches CBS testing process by MMFSQM and takes account of the limitation of debugging resources. The experimental studies and results indicate that the proposed simulation approach is feasible. The simulation procedures can help software engineers gain an insight into the characteristics of CBS reliability and perform reasonable adjustment of debugging resources & decision support to achieve a maximum level of reliability in a cost-effective manner. In reality, debugging resources cover not only the debuggers, but also CPU hours, imperfect debugging has a big effect on residual faults and the imperative testing cost control. Besides, explicitly incorporate operational profile and changeable structure into CBS reliability modeling and analysis, and further research on these topics would be worthwhile.

References

[1] Almering V, van Genuchten M, Cloudt G, et al. Using software reliability growth models in practice. *IEEE Software*, 2007, 24(6): 82-88

[2] Gokhale S S. Architecture-based software reliability analysis overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 2007, 4(1): 32-40

[3] Gokhale S S. Analytical models for architecture-based software reliability prediction-a unification framework. *IEEE Transactions on Reliability*, 2006, 55(4): 578-590

[4] 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 Transactions on Reliability*, 2011, 60(1): 331-340

[5] Wu Y P, Hu Q P, Xie M, et al. Modeling and analysis of software fault detection and correction process by considering time dependency. *IEEE Transactions on Reliability*, 2007, 56(4): 629-642

[6] Hou C Y, Cui G, Liu H W, et al. A hybrid queueing model with imperfect debugging for component software reliability analysis. *Intelligent Automation and Soft Computing*, 2011, 17(5): 559-570

[7] Huang C Y, Huang W C. Software reliability analysis and measurement using finite and infinite server queueing models. *IEEE Transactions on Reliability*, 2008, 57(1): 192-203

[8] Lin C T. Analyzing the effect of imperfect debugging on software fault detection on software fault detection and correction processes via a simulation work. *Mathematical and Computer Modeling*, 2011, 54: 3046-3064

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

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

[11] Hou C Y, Cui G, Liu H W. Rate-based component software reliability process simulation. *Journal of Software*, 2011, 22(11): 2749-2759

[12] Lin C T, Huang C Y, Sue C C. Measuring and assessing software reliability growth through simulation-based approaches. In: Proceedings of the 31st Annual International Computer Software and Applications Conference, Beijing, China, 2007. 439-448

[13] Gokhale S S, Michael Rung-Tsong Lyu. A simulation approach to structure-based software reliability analysis. *IEEE Transactions on Software Engineering*, 2005, 31(8): 643-656

[14] Goel A L, Okumoto K. Time-dependent error-detection rate model for software reliability and other performance measures. *IEEE Transactions on Reliability*, 1979, R-28(3): 206-211

[15] Xie M, Yang B. A study of the effect of imperfect debugging on software development cost. *IEEE Transactions on Software Engineering*, 2003, 29(5): 471-473

[16] Cheung R C. A user-oriented software reliability model. *IEEE Transactions on Software Engineering*, 1980, SE-6(2): 118-125

Zhang Ce, born in 1978, received his Bachelor and Master degrees of computer science and technology from Harbin Institute of Technology (HIT) and Northeast University (NEU), China in 2002 and 2005, respectively. He has been a Ph. D. candidate of HIT major in computer system structure since 2010. His research interests include software reliability modeling and Trusted Computing (TC).