A Simulation Approach to Structure-Based Software Reliability Analysis

Swapna S. Gokhale, *Member*, *IEEE*, and Michael Rung-Tsong Lyu, *Fellow*, *IEEE*

Abstract—Structure-based techniques enable an analysis of the influence of individual components on the application reliability. In an effort to ensure analytical tractability, prevalent structure-based analysis techniques are based on assumptions which preclude the use of these techniques for reliability analysis during the testing and operational phases. In this paper, we develop simulation procedures to assess the impact of individual components on the reliability of an application in the presence of fault detection and repair strategies that may be employed during testing. We also develop simulation procedures to analyze the application reliability for various operational configurations. We illustrate the potential of simulation procedures using several examples. Based on the results of these examples, we provide novel insights into how testing and repair strategies can be tailored depending on the application structure to achieve the desired reliability in a cost-effective manner. We also discuss how the results could be used to explore alternative operational configurations of a software application taking into consideration the application structure so as to cause minimal interruption in the field.

Index Terms—Application structure, reliability analysis, discrete-event simulation.

1 Introduction

Structure-based software reliability analysis techniques are gaining increasing attention with the advent of component-based software development paradigm [16], [17], [34]. These techniques are more suited to assess the reliability of modern software systems[1] than the traditional software reliability growth models [6] due to a variety of reasons. These techniques enable us to:

1. relate system reliability to its structure and the individual component[2] reliabilities,
2. analyze the sensitivity of system reliability to the reliabilities of its components,
3. explore alternatives to optimize various system parameters such as performance, reliability, and cost,
4. identify reliability bottlenecks,
5. assess system reliability earlier in the life cycle where maximum latitude exists to take corrective action if the system reliability does not meet the desired expectations, and
6. assess the reliability of operational systems to identify components which provide maximum potential for reliability improvement.

1. The terms system, software system, application, and software application are used interchangeably in this paper.

2. The terms component and module are used interchangeably in this paper.

- *S.S. Gokhale is with the Department of Computer Science and Engineering, 371 Fairfield Road Unit 1155, University of Connecticut, Storrs, CT 06269. E-mail: ssg@engr.uconn.edu.*
- *M.R.-T. Lyu is with the Computer Science and Engineering Department, The Chinese University of Hong Kong, Shatin NT, Hong Kong. E-mail: lyu@cse.cuhk.edu.hk.*

Manuscript received 10 Mar. 2004; revised 6 Sept. 2004; accepted 27 Dec. 2004; published online 12 Aug. 2005.
Recommended for acceptance by P. Jalote.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0040-0304.

The structure of a software application may be defined as a collection of components comprising the application and the interactions among the components. A component could be a single function, a class, an object, or a collection of these. The interactions among the components may be procedure calls, client-server protocols, links between distributed databases, or synchronous and asynchronous communication using middleware [41]. From the point of view of reliability analysis, it is the dynamic structure of the software application which is important. The dynamic structure of a software application consists of its static structure augmented by runtime information, where static structure consists of the interactions among the components of an application that can be derived by analyzing the design and code of the application, but without executing the application (or simulating its execution). The runtime information may include the frequency of occurrence of the interactions, the time spent in the interactions, and any other data that may be relevant. Dynamic structure may be obtained by executing the application or in the early phases by simulating its execution. It depends on how the application is used, which may be given by its operational profile [27]. Reliability analysis based on dynamic structure thus considers the usage characteristics of the application.

Existing structure-based reliability analysis techniques consider only sequential applications, that is, applications where exactly one component is executing at any given time. The application structure is represented by its control flow graph augmented by transition probabilities which govern the actual flow of control among the components at runtime. The transition probabilities depend on the usage characteristics of the application. Depending on how the application structure represented by its probabilistic control flow graph is analyzed to obtain application reliability, structure-based techniques can be classified into two

0098-5589/05/$20.00 © 2005 IEEE Published by the IEEE Computer Society
Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

categories, namely, path-based [20], [35], [40] and state-based [2], [22], [24], [32], [33]. In the path-based techniques, paths through the control flow graph are enumerated. The enumeration may be performed either algorithmically [40], by simulation, or by experimentation [20]. The reliability of each path is obtained as a product of the component reliabilities along the path, and the application reliability is obtained by averaging over the path reliabilities. Simulation has also been used in conjunction with the path-based approach to obtain the distribution of application reliability instead of just a point estimate [34]. An inherent drawback of the path-based approach is that it cannot consider infinite paths that may exist in the control flow graph due to the presence of loops. State-based techniques use a state space model such as a discrete time Markov chain (DTMC) [2], [32], [33], a continuous time Markov chain (CTMC) [22], or a semi-Markov process (SMP) [24] to represent the probabilistic control flow graph and obtain an estimate of the application reliability analytically. A state-based approach can consider the impact of infinite paths through the analytical solution of the state space model representing application structure.

Most of the existing state-based and path-based approaches represent the failure behavior of the components of the application either using their reliabilities [2], [32], [33] or constant failure rates [22]. This may be adequate in the early phases of the software life cycle when structure-based analysis is used to obtain an initial judgment regarding the sensitivity and the criticality of the components. These two failure models, however, are inappropriate in the testing phase where testing results in reliability growth of the components, and this component-level reliability growth needs to be propagated to the application-level reliability growth based on the application structure. Reliability growth of a component depends on its fault detection rate and the fault repair rate. A fault detection rate may be given by one of the software reliability growth models, whereas the fault repair rate may be determined by the repair policy employed [13]. In order to determine the impact of fault detection rate, it is necessary to represent the failure behavior of a component using a time-dependent failure rate as in the research reported by Laprie et al. [23] and Kanoun et al. [19]. The research reported in [23] and [19], however, assumes instantaneous and perfect repair, an assumption which has been found to be unrealistic in practice [3], [39]. A number of research efforts have incorporated explicit repair into software reliability models [13], [31]. Although these models with explicit repair can be used for a single component, techniques which use the component models to determine the application reliability within the context of its structure do not exist. An analysis of the impact of component level fault detection and repair on application reliability can be used to guide the allocation of resources to the components so that the application reliability target can be achieved in a cost-effective manner. A yet another limitation of the existing structure-based techniques is that they cannot consider configurations where a subset of the application components are employed with fault tolerance capability in order to improve the application reliability. Fault tolerance is considered to be an effective way to improve system reliability. Employing fault tolerance, however, is expensive and, hence, to achieve a balance between the costs incurred in providing fault tolerance and the level of reliability achieved, it may be employed for some rather than all of the application components. The subset of application components for which fault tolerance is employed may be chosen taking into consideration a number of factors such as the criticality of the functions provided by the component, the usage frequency of the component, and the reliability of the component. Fault tolerant software systems have been extensively analyzed for their reliability and availability, some of these efforts are reported in [37], [28]. However, typically, these models treat each version of the application as a black box. When fault tolerance is employed for only a subset of the application components, the impact of component level fault tolerance must be considered in the context of the application structure.

Enhancing the state space models to consider various aspects described above that occur during testing and operation will lead to mathematically intractable models. Discrete-event simulation offers an attractive alternative to analytical models as it can represent the impact of several strategies that may be employed during testing and different deployment configurations during operation. In this paper, we develop detailed simulation procedures which can be used to assess the impact of fault detection and repair at the component-level on the application reliability during testing. We also develop simulation procedures to assess the application reliability when fault tolerant configurations are employed for a subset of the application components. We illustrate the simulation procedures developed in the paper using several examples. Based on the results obtained from the examples, we provide novel insights into how the testing and the repair strategies could be tailored specific to the application structure to achieve the desired reliability in a cost-effective manner. We also discuss how the results could be used to explore alternative operational configurations of a software application for a specific application structure so as to cause minimal interruption in the field. Our prior research [11], [12] has focused on incorporating explicit repair and fault tolerance in black-box models. In this paper, we incorporate these factors within the context of application structure. Similar to the existing research in structure-based reliability analysis, the simulation procedures developed in this paper consider sequential applications. Developing simulation procedures for the reliability analysis of concurrent applications is the topic of future research.

The rest of the paper is organized as follows: Section 2 describes the stochastic failure process of a single component as well as an application comprised of a collection of interacting components. While describing the stochastic failure process, we further motivate the need for discrete-event simulation as an alternative to analytical modeling. Section 3 presents the simulation procedures to analyze the application reliability in the testing and operational phases. Section 4 illustrates the simulation procedures presented in Section 3 using examples. Section 5 presents conclusions and directions for future research.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

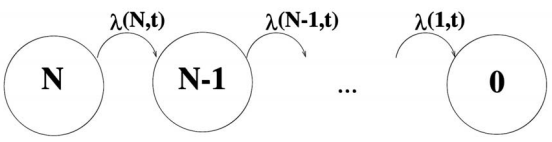

Fig. 1. State space view of pure death NHCTMC.

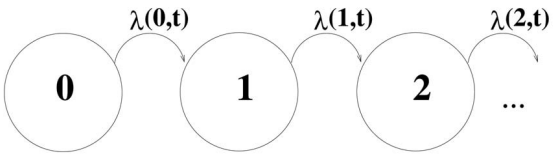

Fig. 2. State space view of pure birth NHCTMC.

2 Stochastic Failure Process

In this section, we initially describe the stochastic failure process of a single component. We then describe the stochastic failure process of a software application composed using software components. Through the discussion of the application-level stochastic failure process, we further motivate the need for discrete-event simulation.

2.1 Component-Level Stochastic Failure Process

The stochastic failure process of each component denoted $\{X(t)\}$ is the number of failures observed in an execution interval of length t. We assume that a failure occurs when the service delivered by the component deviates from its specified service [26]. We assume one-to-one mapping between faults and failures [6]. Also, each failure is unique and there are no failure dependencies.

The failure behavior of a component can be characterized using:

- Probability of failure or reliability: If the failure behavior of the component is modeled by its probability of failure or reliability, then $\{X(t)\}$ counts the number of executions of the component that resulted in a failure. The notion of time in this case is replaced by the number of times the component is executed.
- Failure rate: If the failure behavior of the component is modeled by a failure rate, then the stochastic process $\{X(t)\}$ can be modeled by a class of nonhomogeneous continuous time Markov chain (NHCTMC) which depends only on the failure rate of the component. The failure rate of the component may be denoted $\lambda(n,t)$, where n is the state of the component. The state of the component depends on the number of failures observed from the component. If the maximum number of failures that can be observed from a component is fixed, say N, and i failures have occurred by time t, then the state of the component is given by $N-i$. The stochastic process $\{X(t)\}$ can be viewed as a pure death NHCTMC in this case. Fig. 1 depicts pictorially the state space view of a pure death NHCTMC. On the other hand, if the maximum number of failures that can be observed from a component is a random variable, then i, the number of failures observed up to time t, defines the state of the component. The stochastic process $\{X(t)\}$ can be viewed as a pure birth process in this case. Fig. 2 depicts pictorially the state space view of a pure birth NHCTMC. Modeling the stochastic failure process of a component as a nonhomogeneous continuous time Markov chain (NHCTMC) provides a generic framework and easily accommodates the scenario when the failure behavior of a component is described either by a constant failure rate or the failure rate of one of the popular software reliability growth models. If the failure behavior is described by a constant failure rate, then $\lambda(n,t)=\lambda$. In this case, the failure rate does not depend on the state of the component as well as the time. If the failure behavior is described by the failure rate of one of the software reliability growth models, then the failure rate may depend on the state of the component as well as the time spent in the component. The Jelinski-Moranda model [18] describes the stochastic failure process of a component by a pure death NHCTMC with the failure rate dependent on the state of the component. On the other hand, the Goel-Okumoto model [7], Generalized Goel-Okumoto model [8], Yamada S-shaped model [42], Duane model [4], and the log-logistic model [15] describe the stochastic failure process by a pure birth NHCTMC. The failure rate in the case of these models is dependent only on the execution time in the component. Without loss of generality, in the subsequent discussion, we assume that the stochastic failure process can be described by a pure birth NHCTMC process, with both time and state-dependent failure rate.

One of the parameters of interest is the expected number of failures that are observed from a component in an interval of length t (or in a given number of executions k when the failure behavior is given by probability of failure) and in the steady state ($t=\infty$ or $k=\infty$). In addition to the expected number of failures, a mean or average realization of the stochastic failure process of the component over a given time interval is also of interest. Analytical closed form expressions can be derived to obtain both transient and steady state expected number of failures [38]. These processes are also called as conditional event rate processes [36].

2.2 Application-Level Stochastic Failure Process

In this section, we describe the application-level stochastic failure process. We consider an application with k components. We assume that the failure of each component results in the failure of the application. If the failure behavior of each component is specified by the probability of failure or reliability, then the component-level stochastic failure process depends on the reliability of each component and the number of times each component is executed. If the failure behavior of each component is described by a failure rate, then let $\lambda_i(n_i,t_i)$ denote the failure rate of component i. n_i denotes the number of failures observed from component i upto time t_i, where t_i is the total time spent in the execution of component i. Let $\Lambda(\mathbf{n},\mathbf{t})$ denote the failure rate of the application. Here, $\mathbf{n}=(n_1,n_2,\ldots,n_k)$ and

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

$\mathbf{t} = (t_1, t_2, \ldots, t_n)$. The length of the interval t can be obtained by adding the t_is. The application-level stochastic failure process depends on the failure rate $\Lambda(\mathbf{n}, \mathbf{t})$, which in turn depends on the number of failures observed from each component and the total time spent in the execution of each component. The total execution time in a component depends on the execution time per visit and the number of visits to the component. The number of visits to each component is a function of the application structure. The application-level stochastic failure process thus depends on the stochastic failure process of each component and the dynamic structure of the application. The application-level stochastic failure process in this case can no longer be described by a NHCTMC and is generally analytically intractable.

```
int component_level(double t, double dt, double (* lambda) (int, double))
{
    int total_faults_detect;
    double current_time;

   n=0;
   current_time = 0.0;
   while (current_time <= t)
   {
       if (occurs(lambda(total_faults_detect,current_time)))
           total_faults_detect++;
       current_time += dt;
   }
  return total_faults_detect;
}
```

Fig. 3. Component-level simulation procedure.

3 Simulation Procedures

In Section 2, we discussed the intractable nature of the stochastic failure process of the application although the failure process of each component may be analytically tractable. Rate-based simulation can be used to analyze such processes which may be analytically intractable [36]. In this section, we first present a generalized simulation procedure for the stochastic failure process of a component. Subsequently, we present simulation procedures for the application for different fault detection and repair strategies that might occur during testing and alternative configurations that might be deployed during operation. These procedures build upon the simulation procedure for a single component and incorporate the structure of the application. The simulation procedures consider the relationship and ordering of the several events that occur during the execution of the application during testing and operation. These events include transfer of control among the components, failures of the components, and repair of the detected faults. The procedures in this section are described in a C-like form. We investigated the use of several commercially available simulation tools for the implementation of these procedures [30]. However, the primary drawback of these tools is their inability use a time-dependent rate for event occurrences. As a result, for illustrative purposes, the procedures outlined in this section were implemented in the C programming language. They could be implemented in any other general purpose language as well.

3.1 Component-Level Simulation Procedure

The simulation procedure shown in Fig. 3 simulates a single realization of the failure process of a component and returns the total number of failures observed in a given interval. It accepts as input the length of the time duration denoted t, the time step used in the simulation dt, and the failure rate of the component $\lambda(n, t)$. For each time step dt, the function *occurs* determines if the component fails in that step dt by comparing the product $\lambda(n, t) * dt$ with a random number x between 0.0 and 1.0. The component is said to have failed in the time step dt if $\lambda(n, t) * dt < x$. This is repeated for the entire interval t. The procedure assumes that the component begins execution at time 0.0 and no failures have occurred prior to that time. Upon completion, the procedure returns the total number of failures observed from the component during the time interval $(0, t)$. The procedure can be easily modified to return a single realization of the stochastic failure process of the component over the entire time interval.

Although the simulation procedure shown in Fig. 3 generates a realization of the stochastic failure process when the failure behavior of a component is described by a failure rate, the same procedure can accommodate the case when the failure behavior of the component is described by its reliability through an appropriate interpretation of the input parameters. The parameter t can be set to represent the total number of executions of the component, dt can be set to 1 indicating a single execution, and $\lambda(n, t)$ can be set to the reliability of the component. The function *occurs* in this case would compare the reliability of the component with a random number x between 0.0 and 1.0, and the component is considered to have failed if its reliability is less than x.

3.2 Application-Level Simulation Procedures

Building upon the component-level simulation procedure presented in Section 3.1, in this section, we present application-level simulation procedures which could be used to analyze the impact of different fault detection and repair strategies during testing and alternative configurations during operation.

We consider a terminating application, that is, an application that operates on demand. The application consists of k components and, without loss of generality, we assume that the application begins execution with component 1 and terminates upon the execution of component k. The structure of the application is specified by intercomponent transition probabilities denoted $p_{i,j}$. $p_{i,j}$ represents the probability that component j is executed upon the completion of component i. The time spent in each component per visit can be fixed or a random variable which may follow any general distribution with known parameters. The transition probabilities and the time spent in each component[3] per visit could be estimated based on the profile data collected during simulation of the specification of the application [10], [41]. We note that, unlike most

3. Exactly what is designated as a component is a matter of trade-off between the overhead incurred in data collection and the granularity of the analysis desired. The simulation procedures described in this paper are independent of what is designated as a component.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

analytical models which require component execution times to be exponentially distributed, simulation can accommodate any standard distribution such as uniform and normal or any non standard general distribution. We let vector Φ hold the distribution of the time spent in component i along with the parameters of the distribution.

3.2.1 Testing Phase

During integration testing, the entire application is tested with all the components working together using test cases sampled from the operational profile of the application. If a given test case results in a failure, then the fault that caused the failure is identified and isolated. The process of testing with the next test case continues after the fault is identified. If the nature of the fault is such that it prevents testing from proceeding any further, usually a workaround is procured so that testing can continue. Initially, we assume that the faults are repaired instantaneously upon detection. Thus, as the application is executed with test cases and faults are identified and repaired (instantaneously), the components experience reliability growth. In this type of testing, it is important to note that, unless the faults residing in the components that occur earlier in the execution sequence are detected and isolated, the faults that exist in the components that occur later in the sequence cannot surface. In other words, the faults in the earlier components "mask" faults in the subsequent components. As testing continues, fewer and fewer faults are detected from the earlier components, thereby providing an opportunity for the faults from the later components to be detected. The rate of reliability growth of each component as a function of testing time will thus depend on the structural context of the component. Thus, for a given level of testing, the level of reliability for a given component will depend on its failure rate, the time spent in the component during each visit, and the number of visits to the component. The latter two factors determine the extent to which the component is exercised, which will depend on the application structure. The initial failure rate of each component used in integration testing may be the failure rate estimated using the failure data at the end of unit testing of the component.

The simulation procedure presented in Fig. 4 simulates a single realization of the stochastic failure process for the above scenario. It returns the total number of failures observed in a time interval t. It can be easily modified to return a realization of the stochastic failure process of the application. The application starts executing at time 0.0 and no failures are observed prior to that time. The procedure accepts the following parameters as input: length of time interval t, time step dt, failure rate of the components in the array Λ, information regarding the execution time spent in each component per visit stored in the array Φ, and intercomponent transition probabilities in the two dimensional array $\mathbf{P}$. The procedure described in Fig. 4 assumes that the components fail independently. Almost without exception, existing research in the area of structure-based reliability analysis relies on this assumption. This assumption was necessary to enable tractability in the present state space models. Simulation offers the flexibility to relax this assumption, and developing procedures to consider dependent component failures is the topic of our future research.

```
int application_level(double t, double dt, double (* lambda)[k] (int, double),
                      double *phi[k], double P[k][k])
{
    int total_faults_detected, failed, faults_detect[k];
    double global_clock, local_clock[k],  total_time_this_visit, time_so_far;
    initialize();
    while (global_clock < t) {
        total_time_this_visit = generate_time_this_visit(phi[curr_comp]);
        while (time_so_far < total_time_this_visit && failed ! = 1)  {
            time_so_far += dt;
            global_clock += dt; local_clock[curr_comp] += dt;
            if (occurs(generate_failure(lambda[k](faults_detect[curr_comp],
                       local_clock[curr_comp]))  {
                total_faults_detected++; faults_detect[curr_comp]++;
                failed = 1;
                curr_comp = k;
                break;
            }
        }
        failed = 0; time_so_far = 0;
        if (curr_comp == k) curr_comp = 1;
        else  curr_comp = determine_next_comp(P,curr_comp);
    }
    return total_faults_detected;
}
```

Fig. 4. Simulation procedure with instantaneous repair.

In addition to the input parameters, the procedure uses several variables to control the simulation process. *curr_comp* denotes the component that is currently executing. *total_faults_detected* keeps track of the number of application failures. *total_time_this_visit* denotes the time that will be spent in executing the current component provided that the component does not fail. The variable *time_so_far* represents the time spent in the component during the present visit thus far and will be less than the *total_time_this_visit*. The array *local_clock* keeps track of the time that has already been spent in each component since the beginning of testing. The variable *global_clock* keeps track of the total time spent in testing the application. The array *faults_detect* keeps track of the number of failures that have been observed from each component. The function *generate_time_this_visit* determines the time that will be spent executing the current component during the present visit. The inner while loop determines if the component fails during the time interval that is generated by the function *generate_time_this_visit*. The function *generate_failure* is used for this purpose. If the component fails, then *total_faults_detect* is incremented, and *curr_comp* is set to the last component k before breaking from the while loop. The *if* statement following the while loop checks if *curr_comp* is set to k. The value of the *curr_comp* can be equal to k in two cases: 1) any component (including component k) resulted in a failure, and 2) component k finished executing successfully and, hence, the application finished executing successfully. In both these cases, the execution of the application needs to be started all over, which is accomplished by setting the variable *curr_comp* to 1. If the value of *curr_comp* is not k, then the component to be executed next is determined by the function *determine_next_component*. This function accepts intercomponent transition probabilities in the matrix $\mathbf{P}$ and the current component as input.

The simulation procedure offers the flexibility to characterize the failure behavior of each component using

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

a different software reliability growth model. In addition, the simulation procedure can also permit the failure behavior of the components to be represented in a heterogeneous manner, that is, the failure behavior of some of the components could be represented by time-dependent failure rates, some of the components by constant failure rates, and some of the components by reliabilities. Representing the failure behavior of the components in a heterogeneous manner may be necessary for an application assembled from components, where information at different levels of detail may be available for different components [9]. The simulation procedure presented in Fig. 4 can be used to perform several types of predictive or "what-if" analysis. For example, it can be used to determine the failure rate that needs to be achieved for a component at the end of unit testing for its execution time per visit and structural context for a given level of reliability at the end of integration testing.

The simulation procedure presented in Fig. 4 assumes that the faults are repaired instantaneously upon detection and without the introduction of any new faults. In order to analyze a realistic testing situation, we develop simulation procedures which incorporate repair explicitly. We assume that the repair is perfect and introduces no new faults. Incorporating imperfect repair into the simulation procedures is the topic of future research. When repair is considered explicitly, at any given time, the expected number of faults repaired will in general be less than the expected number of faults detected. A detected but as yet unrepaired fault can cause additional failures. However, from the point of view of repair, multiple failures of the same fault are equivalent to a single failure since fixing one fault would simultaneously eliminate all the failures that occurred due to the fault. During testing, the failure rate of each component thus depends on the number of undetected faults but not unrepaired faults, as in the case of instantaneous repair. The failure rate during operation, however, will depend on both undetected faults and detected but unrepaired faults. Hence, the detected faults must be repaired before the application is released into the field.

We consider the following two cases of explicit repair:

- In the first case, we assume shared repair facility for all the components, and the faults are repaired in the order of their detection. The repair rate of the shared repair facility is assumed to be constant, denoted μ. This scenario is also known as sequence dependent repair [1]. The simulation procedure with sequence dependent repair is shown in Fig. 5. In addition to the parameters in Fig. 4, the procedure in Fig. 5 has additional parameters to control the repair process. The procedure accepts the repair rate of the shared repair facility μ as input. It keeps track of the total number of faults pending and the total number of faults repaired using the variables *total_faults_pending* and *total_faults_repaired*. It also retains information regarding the sequence in which the faults are detected from each component. At every time step *dt*, the procedure checks for pending faults. If any faults are unrepaired, then the procedure checks if repair occurs during that time step. If only one fault is pending, then the repair is invoked only if the pending fault was not detected in the current run. The variable *this_run* is used to determine if a fault has been detected in the present run. If repair occurs, then the variable *total_faults_repaired* is incremented, whereas the variable *total_faults_pending* is decremented.
- In the second case, we assume that each of the k components has a dedicated or independent repair facility. The simulation procedure for this case is shown in Fig. 6. The procedure accepts the repair rate of each component as input in the array $\mu[k]$. The variables in this procedure are very similar to the variables in the simulation procedure with sequence dependent repair. At each time step, the procedure checks if there any faults pending for each component and determines whether the fault is repaired during that time step. If only one fault is pending, then the repair is invoked only if the pending fault was detected prior to the current run. The variable *this_run[i]* is used to determine if a fault has been detected from component i in the present run. If repair occurs, then the corresponding variables are adjusted.

```
int app_level_seq_dep(double t, double dt, double (* lambda)[k] (int, double),
                      double *phi[k],  double P[k][k], double mu)
{
    int total_faults_detected, total_faults_pending, total_faults_repaired;
    int failed, faults_detect[k], faults_pending[k], faults_repaired[k], this_run;
    double global_clock, local_clock[k],  total_time_this_visit, time_so_far;
    initialize();

    while (global_clock < t)  {
        total_time_this_visit = generate_time_this_visit(phi[curr_comp]);
        while (time_so_far < total_time_this_visit && failed ! = 1) {
              time_so_far += dt;
              global_clock += dt; local_clock[curr_comp] += dt;
              if (occurs(generate_failure(lambda[k](faults_detect[curr_comp],
                    local_clock[curr_comp]))) {
                   total_faults_detected++; faults_detect[curr_comp]++;
                   total_faults_pending++;
                   faults_pending[total_faults_detected] = curr_comp;
                   failed = 1; curr_comp = k; this_run = 1;
                   break;
              }
        }
        if (total_faults_pending > 0) {
            if (total_faults_pending == 1 && this_run !=1) {
              if (occurs(mu*dt)) {
                  total_faults_pending--;
                  total_faults_repaired++;
                  index = faults_pending[total_faults_repaired];
                  faults_repaired[index]++;
              }
            }
        }
        failed = 0; time_so_far = 0; this_run = 0;
        if (curr_comp == k) curr_comp = 1;
        else  curr_comp = determine_next_comp(P,curr_comp);
    } return total_faults_repaired;
}
```

Fig. 5. Simulation procedure with sequence dependent repair.

The procedures shown in Figs. 5 and 6 return the total number of faults repaired upon completion. Although the simulation procedures described in Figs. 5 and 6 consider

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

```
int app_level_indep(double t, double dt, double (* lambda)[k] (int, double),
                    double *phi[k],  double P[k][k], double mu[k])
{
    int total_faults_detected, total_faults_pending, total_faults_repaired;
    int failed, faults_detect[k], faults_pending[k], faults_repaired[k], this_run[k]
    double global_clock, local_clock[k],  total_time_this_visit, time_so_far;
    initialize();
    while (global_clock < t)  {
        total_time_this_visit = generate_time_this_visit(phi[curr_comp]);
        while (time_so_far < total_time_this_visit && failed ! = 1)  {
              time_so_far += dt;
             global_clock += dt; local_clock[curr_comp] += dt;
             if (occurs(generate_failure(lambda[k](faults_detect[curr_comp],
                       local_clock[curr_comp])))  {
                  total_faults_detected++; faults_detect[curr_comp]++;
                  total_faults_pending++;  faults_pending[curr_comp]++;
                 failed = 1; this_run[curr_comp] = 1; curr_comp = k;
                 break;
              }
        }
        for (i=1;i<=k;i++)  {
             if (faults_pending[i] > 0)  {
               if (faults_pending[i] == 1 && this_run[i] !=1 ) {
                  if (occurs(mu[i]*dt))  {
                     total_faults_pending--;  total_faults_repaired++;
                     faults_repaired[i]++; faults_pending[i]--;
                  }
               }
             }
         this_run[i]  = 0;
        }
        failed = 0; time_so_far = 0;
        if (curr_comp == k) curr_comp = 1;
        else  curr_comp = determine_next_comp(P,curr_comp);
    }
    return total_faults_repaired;
}
```

Fig. 6. Simulation procedure with independent/dedicated repair.

```
int app_level_operation(int max_runs, double rel[k],
                        double P[k][k])
{
     int total_num_of_failures, failed, curr_comp, faults_detect[k];
     int curr_num_runs, complete;
     initialize();
     while (curr_num_runs < max_runs) {
          while (complete != 1) {
               if (occurs(rel[curr_comp])) {
                    failed = 1;
                    complete = 1;
               }
               curr_comp = determine_next_component(P,curr_comp);
               if (curr_comp == 10 && failed !=1 ) {
                    complete = 1;
                    if (occurs(rel[curr_comp]) {
                         failed = 1;
                    else
                         failed = 0;
                    }
               }
     }
     if (failed == 1)
          total_num_of_failures++;
     curr_num_runs++;
     curr_comp = 1;
     failed = 0; complete = 0;
     }
     return total_num_of_failures;
}
```

Fig. 7. Simulation procedure in the operational phase.

constant repair rates, procedures which accommodate different types of repair rates, such as time-dependent, fault-dependent, etc. [14], can be developed easily.

3.2.2 Operational Phase

When the application is released into the field or transitions to the operational phase, the emphasis shifts from finding and repairing faults to minimizing the interruptions in the service. In general, the faults detected in the operational phase are not repaired, and the application does not experience reliability growth. Therefore, the failure behavior of each component in the operational phase can be modeled either using a probability of failure (or reliability) of the component or by a constant failure rate. Based on the estimates of the number of faults detected and repaired at the end of testing, the reliability of each component can be determined using the method described in Section 4. Using the component reliabilities at the end of testing, discrete-event simulation can be used to determine the reliability of the application in the operational phase. Fig. 7 shows the simulation procedure used to determine the number of application failures in a given number of runs which is provided as input. The procedure shown in Fig. 7 can also be used to determine the percentage of application failures that can be attributed to each component, or the *criticality* of each component.

One of the ways of minimizing the interruptions in the service provided by the application is to employ fault-tolerant configurations for some or all of its components. Since employing fault tolerance for all the application components may be prohibitively expensive, a cost-effective alternative may be to employ fault tolerant configurations for a subset of components. This subset can be determined based on component criticalities which can be obtained based on the simulation procedure described in Fig. 7. Fig. 8 shows the simulation procedure which returns the number of failures in a given number of runs when fault tolerant configurations are employed for some critical components. The procedure accepts the components for which fault tolerant configuration is employed as input. For every component for which a fault tolerant configuration is employed, the procedure invokes *sim_ft()* to determine if the component fails. The procedure *sim_ft()* can simulate various types of fault tolerant configurations, such as Distributed Recovery Block [29] and NVP/1/1 [21], using the procedures reported in [11]. The procedure returns the total number of failures observed in a given number of runs.

The effectiveness of a fault tolerance scheme to provide continued service despite failures is captured by a parameter termed coverage [5]. The simulation procedure shown in Fig. 9 seeks to determine the impact of coverage parameters of the components on the application reliability. The procedure accepts the coverage parameters of the components as input and returns the number of failures in a given number of runs.

Procedures shown in Figs. 7, 8, and 9 return the number of failures in a given number of runs. An estimate of application reliability can be obtained as the ratio of the number of failures to the total number of test runs. Also, in

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

```
int app_level_operation_ft(int max_runs, double rel[k],
                    double P[k][k], int ft[k])
{
    int total_num_of_failures, failed, curr_comp;
    int curr_num_runs, complete;
    initialize();
    while (curr_num_runs < max_runs) {
        while (complete != 1) {
            if (ft[curr_comp] == 1)
                failed = sim_ft(curr_comp);
            else {
                if (occurs(rel[curr_comp]))
                    failed = 1;
            }
            if (failed == 1)
                complete = 1;
            curr_comp = determine_next_component(P,curr_comp);
            if (curr_comp == k && failed !=1 ) {
                complete = 1;
                if (ft[curr_comp] == 1)
                    failed = sim_ft(curr_comp);
                else {
                    if (occurs(rel[curr_comp]))
                        failed = 1;
                }
            }
        }
        if (failed == 1)
            total_num_of_failures++;
        curr_num_runs++;
        curr_comp = 1;
        failed = 0; complete = 0;
    }
    return total_num_of_failures;
}
```

Fig. 8. Simulation procedure with fault tolerant configurations for critical components.

all the simulation procedures, a call to *initialize()* at the beginning of the procedure initializes the necessary parameters to appropriate values. It also initializes the variable *curr_comp* to 1.

4 Illustrations

In this section, we demonstrate the potential of the simulation procedures developed in Section 3 through several examples. We use the application reported by Cheung [2] as an example for illustration. This application has been used extensively to illustrate structure-based reliability assessment techniques in the recent past [16], [17], [25]. The structure of the application is shown in Fig. 10. The intercomponent transition probabilities among the components are summarized in Table 1. Only the nonzero entries in $\mathbf{P}$ are listed in Table 1. Section 4.1 illustrates the simulation procedures for the testing phase. Section 4.2 illustrates the simulation procedures for the operational phase.

4.1 Testing Phase

In the first experiment, we use the simulation procedure shown in Fig. 4 to simulate the fault detection profile of the entire application for a given testing interval. For the sake of

```
int app_level_operation_cov(int max_runs, double rel[k],
                    double P[k][k], double cov[k])
{
    int total_num_of_failures, failed, curr_comp;
    int curr_num_runs, complete;
    initialize();
    while (curr_num_runs < max_runs) {
        while (complete != 1) {
            if (occurs(rel[curr_comp])) {
                if (cov[curr_comp] >random())
                    failed = 1;
            }
            curr_comp = determine_next_component(P,curr_comp);
            if (curr_comp == k && failed !=1 ) {
                complete = 1;
                if (occurs(rel[curr_comp]))
                    if (cov[curr_comp > random())
                        failed = 1;
                }
            }
        }
        if (failed == 1)
            total_num_of_failures++;
        curr_num_runs++;
        curr_comp = 1;
        failed = 0; complete = 0;
    }
    return total_num_of_failures;
}
```

Fig. 9. Simulation procedure with coverage parameter of components.

illustration, we assume that the failure behavior of each component is characterized by the failure rate of the Goel-Okumoto software reliability growth model [7]. The failure rate of component i is given by $\lambda_i(t_i) = a_i b_i e^{-b_i t_i}$, where a_i is the expected number of faults detected from component i in infinite testing time and b_i is the detection rate per fault. t_i is the cumulative time spent in component i. Without loss of generality, we set $a_i = 20.05$ and $b_i = 0.0057$ for all the components. We assume that the application spends 1.00 time unit in each component per visit. We simulate the fault detection profile of the application for a testing interval of 1,000 units using the procedure shown in Fig. 4.

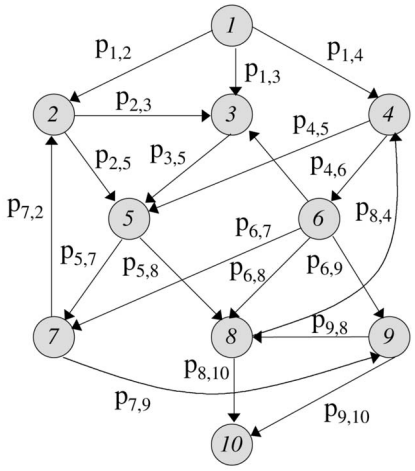

Fig. 10. Structure of an example application.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Intercomponent Transition Probabilities

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

The fault detection profile was simulated 1,000 times and an average of the profile obtained during each run was computed. The average fault detection profile along with the confidence intervals is shown in Fig. 11 which indicates that the upper and lower confidence bounds are within 5 percent around the mean.

In order to illustrate the effect of "masking," where faults from the earlier components preclude the faults from subsequent components from being detected, we compare the total execution time of the components when the components fail according to the above failure rate and when the components do not fail in Table 2. The execution times in Table 2 indicate that the components are exercised in a different manner as a result of failures than they would be exercised if the components did not fail or failed very rarely. In particular, the first few components in the execution sequence are exercised more extensively than what they would be (components 1 through 3), while the latter few components in the sequence are exercised less extensively

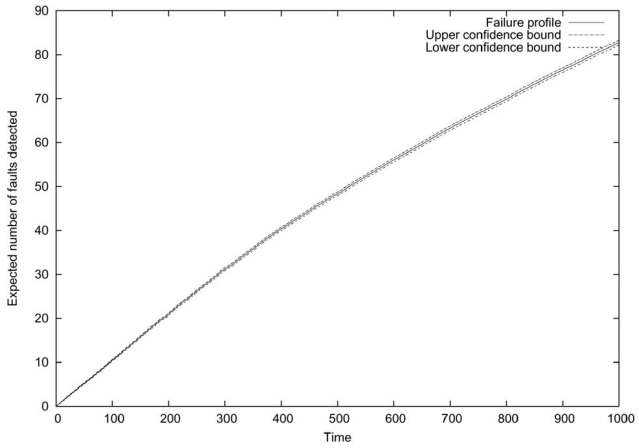

Fig. 11. Fault detection profile during testing.

than what they normally would be (components 5 through 10). Component 4 is approximately exercised to the same level both with and without failures. Fig. 12 depicts the time profile which indicates the rate at which each component is exercised during testing. It can be seen that the rate at which the first component in the execution sequence (component #1) is exercised is the highest among all the components.

Next, we illustrate the simulation procedures shown in Figs. 5 and 6 which incorporate explicit repair. For the case of shared repair, we assume the repair rate of the shared facility to be $\mu = 0.075$. For the case of independent repair, we assume that the repair rate of each repair facility is one-tenth of the repair rate in the case of sequence dependent repair. Thus, the faults detected from component i are repaired at the rate of $\mu_i = 0.0075$, for $i = 1, \ldots, 10$. The expected total number of faults detected in the case of shared as well as independent repairs is 82.74, while the expected total number of faults repaired in the case of shared repair is 72.69, and the total number of faults repaired in the case of independent repair is 49.43. In the case of shared repair, if the faults are detected uniformly over the entire interval, then the average number of faults that would be repaired is 75.00. In the case of independent repair, if the fault detection from each component is

TABLE 2
Comparison of Component Execution Times with and without Failures

Component #	Execution time		Component #	Execution time	
	With failures	Without failures		With failures	Without failures
1	180.39	130.47	2	130.69	117.60
3	124.87	117.17	4	55.07	55.63
5	170.11	175.32	6	29.75	32.78
7	69.55	78.83	8	100.14	114.39
9	39.58	49.34	10	100.41	129.55

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

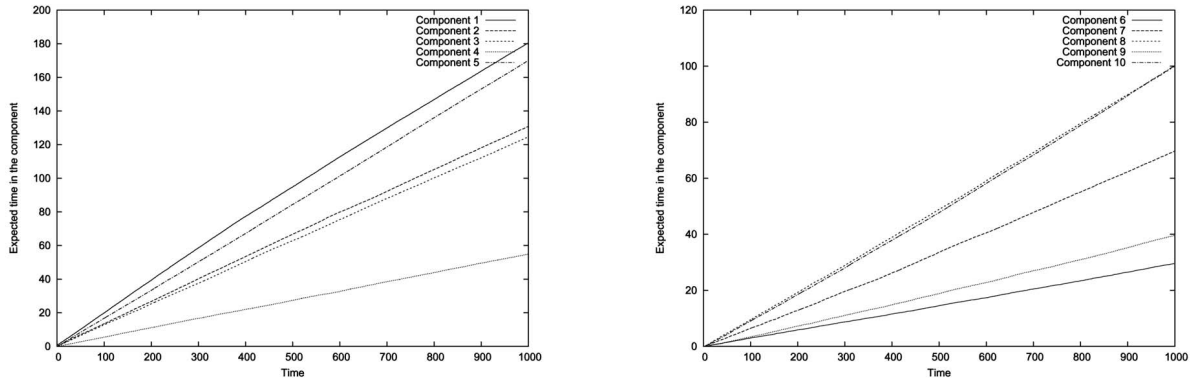

Fig. 12. Execution time profile of each component.

TABLE 3
Expected Number of Faults Detected and Repaired

Component #	Detected	Repaired		Component #	Detected	Repaired	
		Shared	Indep.			Shared	Indep.
1	12.8620	11.5980	6.3330	2	10.6790	9.3980	5.7970
3	10.2230	8.9765	6.0460	4	5.5980	4.6485	3.7810
5	12.4430	11.1980	6.3950	6	3.0840	2.6555	2.4830
7	6.5590	5.7095	4.5440	8	8.6260	7.5430	5.4130
9	3.7860	3.3610	3.0920	10	8.8770	7.6005	5.5410

uniform over the entire interval, then the maximum average number of faults repaired from each component would be 7.5. The maximum number of repaired faults will reach 7.5 only for those components from which the detected faults is higher than 7.5.

The expected number of faults repaired for shared and independent repair from every component by $t = 1,000$ time units is summarized in Table 3. Table 3 indicates that in the case of shared repair the expected number of faults repaired approaches the expected number of faults detected for the earlier components (especially, component 1), whereas, the difference between the expected number of faults detected and repaired increases and is the highest for the last component that occurs in the execution sequence, namely, component 10. In the case of independent repair, for components from which the expected number of faults detected is more than 7.5, the difference between the detected and repaired faults is higher. For components where the expected number of detected faults is less than 7.5, independent repair performs nearly as well as shared repair. Ideally, for these components, independent repair should perform better (higher expected number of repaired faults) than shared repair. However, since the fault detection profile is nonuniform as shown in Fig. 13, where faults from the later components are detected later, shared repair performs slightly better than independent repair even in the case of these components. Based on the above results, a practical testing strategy such as the one described below can be devised. The components could be divided into two groups, where the first group consists of components which occur earlier in the execution sequence, and the second group consists of components which occur later in the execution sequence. A repair facility can be assigned to each one of the two groups. In the earlier part of testing, when a higher number of faults are being detected from the components belonging to the first group as compared to the components belonging to the second group, the first repair facility can be allocated higher resources so that all the detected faults from these components are repaired. In the latter part of testing, when a higher number of faults are being detected from components belonging to the second group as compared to the components belonging to the first group, the second repair facility could be allocated higher level of resources.

In the next example, we determine the impact of the repair rate of the shared repair facility on the total number of repaired faults. Towards this end, we vary the repair rate μ from 0.01 to 0.15 in steps of 0.01 and determine the expected number of faults repaired for each repair rate. Fig. 14 shows the expected number of faults repaired as a function of the repair rate. As intuitively expected, the figure indicates that the higher the repair rate the higher the number of faults repaired. However, the higher the repair rate, the higher are the resources that need to be expended.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

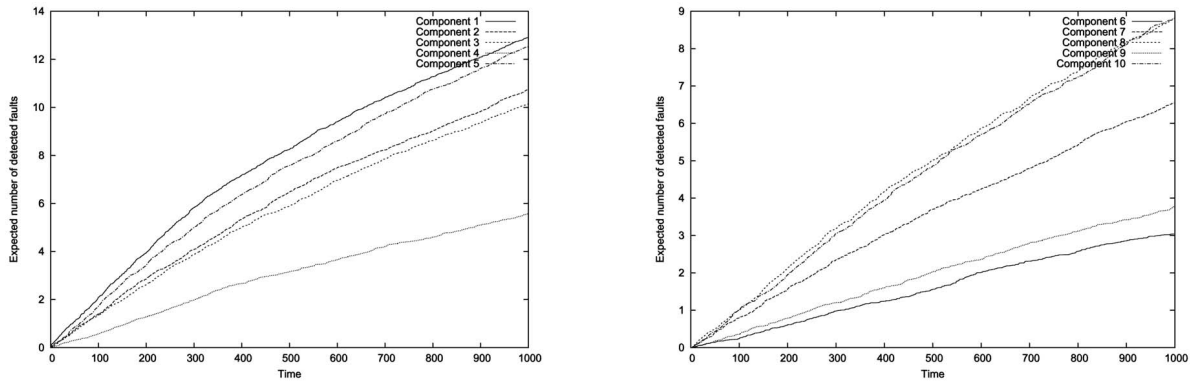

Fig. 13. Fault detection profile of each component.

Also, if the repair rate is beyond a certain threshold, then the repair facility will be idle most of the time waiting for faults to be detected. The percentage of the time the repair facility is busy can be measured by the utilization of the repair facility. Fig. 15 shows the utilization of the repair facility as a function of the repair rate. The figure indicates that, when the repair rate is high, the utilization of the repair facility is low, which results in a waste of resources. This may eventually lead to a budget overrun for the project. On the other hand, if the repair rate is too low, the repair facility is utilized to the fullest extent. However, in this case, the detected faults cannot be repaired with the efficiency that may be necessary to achieve the desired level of reliability in a timely manner. The above analysis indicates that it is imperative to strike a right balance between the utilization of the repair facility and the number of faults repaired. For a given level of resource allocation, if the repair rate can be estimated based on prior experience or historical data, then the simulation procedures can be used to guide resource allocation decisions to achieve a maximum level of reliability in a cost-effective manner.

4.2 Operational Phase

The probability of failure or the reliability of each component in the operational phase is a function of how well the component has been tested, which is indicated by the total time spent in the component. If $t_{i,t}$ is the total time spent in component i at the end of the testing phase, the failure rate of the component prior to release, denoted λ_i assuming that the failure rate during testing was given by the failure rate of the Goel-Okumoto model [7] is given by:

$$\lambda_i = a_i b_i e^{-b_i t_{i,t}}. \tag{1}$$

If the application spends τ_i time units in component i per visit, the reliability of the component at release for every execution can be given by:

$$R_i = e^{-\lambda_i \tau_i}. \tag{2}$$

Equation (2) is based on the assumption of instantaneous repair. For the application in Fig. 10, the total expected time spent in each component at the end of testing and the reliability of each component for $\tau_i = 1$ computed using (2) is summarized in Table 4.

In order to determine the reliability of a component in the presence of explicit repair, we use the technique reported in [12], which is summarized here. In general, for every component i, at time $t_{i,t}$, with explicit repair, the expected number of faults detected will be greater than the expected number of faults repaired. The technique consists of computing a time $t_{i,R}$ such that, at time $t_{i,R}$, the expected number of faults detected and repaired under the assumption of instantaneous repair is equal to the expected number of faults repaired at time $t_{i,t}$ considering explicit repair. In

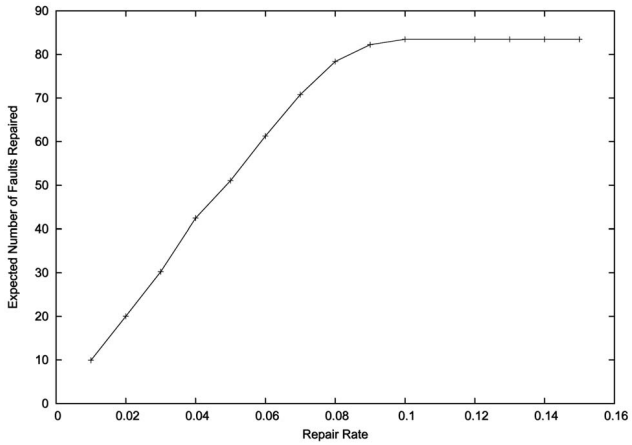

Fig. 14. Expected number of faults repaired versus repair rate.

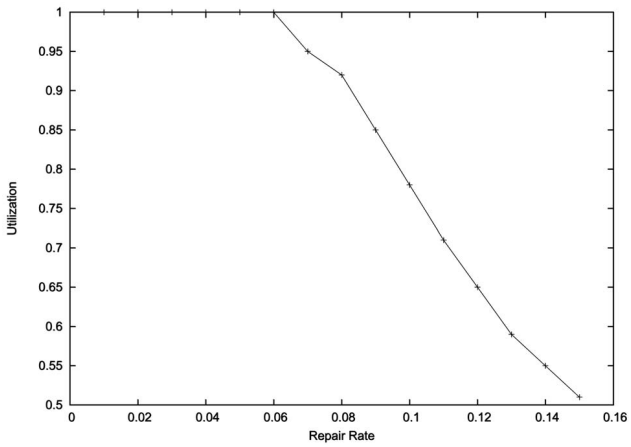

Fig. 15. Utilization versus repair rate.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Reliability of Each Component with Instantaneous Repair

Component #	Detected	Repaired		Component #	Detected	Repaired	
		Shared	Indep.			Shared	Indep.
1	12.8620	11.5980	6.3330	2	10.6790	9.3980	5.7970
3	10.2230	8.9765	6.0460	4	5.5980	4.6485	3.7810
5	12.4430	11.1980	6.3950	6	3.0840	2.6555	2.4830
7	6.5590	5.7095	4.5440	8	8.6260	7.5430	5.4130
9	3.7860	3.3610	3.0920	10	8.8770	7.6005	5.5410

general, $t_{i,R} \leq t_{i,t}$. This may be considered as a "rollback" in time and is like saying that accounting for fault detection and repair separately upto time $t_{i,t}$ is equivalent to instantaneous and perfect repair upto time $t_{i,R}$. $t_{i,R}$ can be used in the failure rate of the appropriate software reliability model to obtain the failure rate of the component at the end of testing phase taking into consideration explicit repair. The failure rate so obtained can be used in (2) to obtain the reliability of the component in the presence of explicit repair. $t_{i,R}$ and the reliability for each component with shared and independent repair is reported in Table 5. From Tables 4 and 5, we can see that the reliability of the components with explicit repair is lower than the reliability of the components with instantaneous repair, which is expected. The important contribution of our technique is that it enables us to quantify the effect of different repair strategies such as shared and independent repair on the reliability of the component.

TABLE 5
Reliability of Each Component with Explicit Repair

Component #	Shared repair		Independent repair	
	$t_{i,R}$	Reliability	$t_{i,R}$	Reliability
1	151.5614	0.9530	66.5953	0.9248
2	110.9649	0.9411	59.8705	0.9220
3	104.1534	0.9388	62.9625	0.9233
4	46.2744	0.9160	36.6610	0.9114
5	143.4361	0.9508	67.3901	0.9251
6	24.9255	0.9056	23.1942	0.9047
7	58.7967	0.9215	45.0881	0.9154
8	82.7966	0.9312	55.2064	0.9200
9	32.1894	0.9093	29.3841	0.9079
10	83.6050	0.9315	56.7474	0.9206

Next, we illustrate the simulation procedure shown in Fig. 7 to obtain the criticality of each component using the component reliabilities obtained at the end of testing. We use the component reliabilities reported in Table 4 for the sake of illustration. Table 6 lists the components in the order of their criticality. From the table, it can be seen that approximately 15 percent of the application failures can be attributed to the failures of each one of components 1 and 5. This percentage is the highest that can be attributed to a single component. As a result, components 1 and 5 can be considered to be highly critical to the uninterrupted operation of the application.

The next scenario demonstrates the ability of discrete-event simulation to simulate the failure profile of an application when fault tolerant configurations are employed for some of its components using the procedure shown in Fig. 8. Through this scenario, we also demonstrate how discrete-event simulation can facilitate the evaluation of competing alternative configurations. Table 6 identifies components 1 and 5 as the most critical components for the success of the application. As a result, for the sake of

TABLE 6
Criticality of Each Component

Component #	Criticality	Component #	Criticality
1	0.1548	2	0.1292
3	0.1235	4	0.0675
5	0.1504	6	0.0375
7	0.0797	8	0.1041
9	0.0459	10	0.1073

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

TABLE 7
Application Reliability for Coverage Parameter of Component #5

Coverage	Reliability	Coverage	Reliability	Coverage	Reliability
0.1	0.6920	0.2	0.7042	0.3	0.7058
0.4	0.7080	0.5	0.7086	0.6	0.7130
0.7	0.7196	0.8	0.7252	0.9	0.7285

illustration we choose to employ a fault tolerant configuration for component 5. We wish to evaluate among two competing alternatives, namely, NVP/1/1 [21] with three parallel versions of the component running on three hardware hosts, or a RB/1/1 [21], i.e., Distributed Recovery Block (DRB) configuration, with two hardware hosts running two recovery blocks [29]. For both the configurations, we assume that the hardware hosts running the software versions/alternates do not fail. We also assume that the voter in the NVP system is perfect. The failure probability of the acceptance test (AT) in case of the DRB was set to one hundredth of the failure probabilities of the alternates. The reliability of the application with no fault tolerant configuration employed for component 5 is 0.6882. The reliability of the application with DRB configuration employed for component 5 is 0.7237 and the reliability of the application with NVP configuration employed for component 5 is 0.7244.

In the next experiment, we seek to illustrate the impact of the coverage parameter of component 5 on the application reliability. Table 7 shows application reliability for different values of coverage of component 5 obtained using the simulation procedure in Fig. 9. The coverage parameter of a fault tolerant scheme typically depends on the correlation among the versions which constitute the fault tolerant configuration. Thus, intuitively, a low correlation among the versions should result in high coverage. As seen from the reliability values in Table 7, the reliability of the application when the coverage is 0.8 is approximately equivalent to the reliability of the application using NVP configuration for component 5, with a correlation of 0.0 among the versions. Thus, we can roughly say that a correlation of 0.0 for the NVP configuration corresponds to a coverage of 0.8. Although no mathematical relationship exists between correlation and coverage, simulation can be thus used to determine an approximate empirical relationship between two related variables for a given system. Studies like these, which help assess the sensitivity of the application reliability to different parameters, can be of great significance in conducting "what-if" analysis to enable the exploration of a set of alternatives.

5 Conclusions and Future Research

In this paper, we have presented detailed simulation procedures which could be used to analyze the impact of different fault detection and repair strategies on the application reliability during testing. Simulation procedures were also developed to explore alternative application configurations during operation. We illustrated the potential of the simulation procedures using several examples. Based on the results obtained from the examples, we provided novel insights into how fault detection and repair strategies can be customized specific to the structure of an application in order to achieve the desired reliability in a cost-effective manner. We also describe how deployment configurations could be chosen to minimize the interruptions in the service during operation.

Developing simulation procedures to analyze the impact of dependent failures among the components is the topic of future research. Also, incorporating imperfect repair in the simulation procedures is the concern of future research.

Acknowledgments

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. CUHK4205/04E).

References

[1] S.J. Bavuso, J.B. Dugan, K.S. Trivedi, E.M. Rothmann, and W.E. Smith, "Analysis of Typical Fault-Tolerant Architectures Using HARP," *IEEE Trans. Reliability,* vol. 36, no. 2, pp. 176-185, June 1987.
[2] R.C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Software Eng.,* vol. 6, no. 2, pp. 118-125, Mar. 1980.
[3] M. Defamie, P. Jacobs, and J. Thollembeck, "Software Reliability: Assumptions, Realities and Data," *Proc. Int'l Conf. Software Maintenance,* Sept. 1999.
[4] J.T. Duane, "Learning Curve Approach to Reliability Monitoring," *IEEE Trans. Aerospace,* vol. 2, pp. 563-566, 1964.
[5] J. Dugan and K.S. Trivedi, "Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems," *IEEE Trans. Computers,* vol. 38, no. 6, pp. 775-787, June 1989.
[6] W. Farr, *Handbook of Software Reliability Engineering,* chapter software reliability modeling survey, M.R. Lyu, ed., pp. 71-117, McGraw-Hill, 1996.
[7] A.L. Goel and K. Okumoto, "A Non-Homogeneous Poisson Process Model for Software Reliability and Other Performance Measures," Technical Report 79-1, Dept. of IE and OR, Syracuse Univ., 1979.
[8] A.L. Goel and K. Okumoto, "Time-Dependent Error-Detection Rate Models for Software Reliability and Other Performance Measures," *IEEE Trans. Reliability,* vol. 28, no. 3, pp. 206-211, Aug. 1979.
[9] S. Gokhale, "Architecture-Based Heterogeneous Software Reliability Framework," *Proc. ISSAT Conf.,* Aug. 2004.
[10] S. Gokhale, J.R. Horgan, and K.S. Trivedi, *Book on Architecting Dependable Systems,* R. de Lemos, et al., eds., vol. 2677, chapter specification level integration of performance and dependability analysis, pp. 245-266, Springer-Verlag, July 2003.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.

[11] S. Gokhale, M.R. Lyu, and K.S. Trivedi, "Reliability Simulation of Fault-Tolerant Software and Systems," *Proc. Pacific Rim Int'l Symp. Fault-Tolerant Systems (PRFTS 97),* pp. 167-173, Dec. 1997.
[12] S. Gokhale, M.R. Lyu, and K.S. Trivedi, "Software Reliability Analysis Incorporating Debugging Activities," *Proc. Ninth Int'l Symp. Software Reliability Eng. (ISSRE 98),* pp. 202-211, Nov. 1998.
[13] S. Gokhale, M.R. Lyu, and K.S. Trivedi, "Analysis of Software Fault Removal Policies Using a Non Homogeneous Continuous Time Markov Chain," *Software Quality J.,* 2004.
[14] S. Gokhale, P.N. Marinos, K.S. Trivedi, and M.R. Lyu, "Effect of Repair Policies on Software Reliability," *Proc. Conf. Computer Assurance (COMPASS 97),* pp. 105-116, June 1997.
[15] S. Gokhale and K.S. Trivedi, "A Time/Structure Based Software Reliability Model," *Annals of Software Eng.,* vol. 8, pp. 85-121, 1999.
[16] S. Gokhale and K.S. Trivedi, "Reliability Prediction and Sensitivity Analysis Based on Software Architecture," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE 02),* Nov. 2002.
[17] K. Goseva-Popstojanova and S. Kamavaram, "Assessing Uncertainty in Reliability of Component-Based Software Systems," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE),* pp. 307-320, Nov. 2003.
[18] Z. Jelinski and P.B. Moranda, *Statistical Computer Performance Evaluation,* W. Freiberger, ed., chapter software reliability research, pp. 465-484, Academic Press, 1972.
[19] K. Kanoun, M. Kaaniche, C. Beounes, J.C. Laprie, and J. Arlat, "Reliability Growth of Fault-Tolerant Software," *IEEE Trans. Reliability,* vol. 42, no. 2, pp. 205-219, June 1993.
[20] S. Krishnamurthy and A.P. Mathur, "On the Estimation of Reliability of a Software System Using Reliabilities of Its Components," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 146-155, Nov. 1997.
[21] J.C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and Analysis of Hardware- and Software Fault-Tolerant Architectures," *Computer,* vol. 23, no. 7, pp. 39-51, July 1990.
[22] J.C. Laprie and K. Kanoun, "X-Ware Reliability and Availability Modeling," *IEEE Trans. Software Eng.,* vol. 15, pp. 130-147, 1992.
[23] J.C. Laprie, K. Kanoun, C. Beounes, and M. Kaaniche, "The KAT (Knowledge-Action-Transformation) Approach to the Modeling and Evaluation of Reliability and Availability Growth," *IEEE Trans. Software Eng.,* vol. 17, no. 4, pp. 370-382, 1991.
[24] B. Littlewood, "A Semi-Markov Model for Software Reliability with Failure Costs," *Proc. Symp. Computer Software Eng.,* pp. 281-300, Apr. 1976.
[25] J. Lo, S. Kuo, M.R. Lyu, and C. Huang, "Optimal Resource Allocation and Reliability Analysis for Component-Based Software Applications," *Proc. 26th Ann. Int'l Computer Software and Applications Conf. (COMPSAC),* pp. 7-12, Aug. 2002.
[26] M.R. Lyu, *Handbook of Software Reliability Engineering.* McGraw-Hill, 1996.
[27] J.D. Musa, "Operational Profiles in Software-Reliability Engineering," *IEEE Software,* vol. 10, no. 2, pp. 14-32, Mar. 1993.
[28] V.F. Nicola and A. Goyal, "Modeling of Correlated Failures and Community Error Recovery in Multiversion Software," *IEEE Trans. Software Eng.,* vol. 16, no. 3, pp. 350-359, Mar. 1990.
[29] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, no. 2, pp. 220-232, June 1975.
[30] A.E. Rizzoli, "A Collection of Modeling and Simulation Resources," http://www.idsia.ch/andrea/simtools.html, 2004.
[31] N.F. Scheidewind, "Fault Correction Profiles," *Proc. Int'l Symp. Software Reliability Eng.,* pp. 257-267, Nov. 2003.
[32] K. Seigrist, "Reliability of Systems with Markov Transfer of Control," *IEEE Trans. Software Eng.,* vol. 14, no. 7, pp. 1049-1053, July 1988.
[33] K. Seigrist, "Reliability of Systems with Markov Transfer of Control, II," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1478-1480, Oct. 1988.
[34] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A Bayesian Approach to Reliability Prediction and Assessment of Component-Based Systems," *Proc. Int'l Symp. Software Reliability Eng. (ISSRE),* Nov. 2001.
[35] N.D. Singpurwalla and S.P. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk.* Springer Verlag, 1999.
[36] R.C. Tausworthe and M.R. Lyu, "A Generalized Technique for Simulating Software Reliability," *IEEE Software,* vol. 13, no. 2, pp. 77-88, Mar. 1996.
[37] L.A. Tomek, J.K. Muppala, and K.S. Trivedi, "Modeling Correlation in Software Recovery Blocks," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1071-1086, Nov. 1993.
[38] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications.* John Wiley, 2001.
[39] A. Wood, "Software Reliability Growth Models: Assumptions vs. Reality," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 136-141, Nov. 1997.
[40] S. Yacoub, B. Cukic, and H. Ammar, "Scenario-Based Analysis of Component-Based Software," *Proc. 10th Int'l Symp. Software Reliability Eng.,* Nov. 1999.
[41] S.M. Yacoub and H.H. Ammar, "A Methodology for Architecture-Level Reliability Risk Analysis," *IEEE Trans. Software Eng.,* vol. 28, no. 6, pp. 529-547, June 2002.
[42] S. Yamada, M. Ohba, and S. Osaki, "S-Shaped Reliability Growth Modeling for Software Error Detection," *IEEE Trans. Reliability,* vol. 32, no. 5, pp. 475-485, Dec. 1983.

Swapna S. Gokhale received the BE degree (honors) in electrical and electronics engineering and computer science from the Birla Institute of Technology and Science, Pilani, India, in June 1994, and the MS and PhD degrees in electrical and computer engineering from Duke University in September 1996 and September 1998, resectively. Currently, she is an assistant professor in the Department of Computer Science and Engineering at the University of Connecticut. Prior to joining UConn, she was a research scientist at Telcordia Technologies in Morristown, New Jersey. Her research interests include software reliability and performance, software testing, software maintenance, program comprehension and understanding, and wireless and multimedia networking. She is a member of the IEEE.

Michael Rung-Tsong Lyu (S'84-M'88-SM'97-F'04) received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1981, the MS degree in computer engineering from University of California, Santa Barbara, in 1985, and the PhD degree in computer science from the University of California, Los Angeles, in 1988. He is currently a professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. He was with the Jet Propulsion Laboratory as a technical staff member from 1988 to 1990. From 1990 to 1992, he was with the Department of Electrical and Computer Engineering, The University of Iowa, Iowa City, as an assistant professor. From 1992 to 1995, he was a member of the technical staff in the applied research area of Bell Communications Research (Bellcore), Morristown, New Jersey. From 1995 to 1997, he was a research member of the technical staff at Bell Laboratories, Murray Hill, New Jersey. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, mobile networks, Web technologies, multimedia information processing, and E-commerce systems. He has published more than 200 refereed journal and conference papers in these areas. He received Best Paper Awards in ISSRE'98 and ISSRE'2003. He has participated in more than 30 industrial projects and helped to develop many commercial systems and software tools. He was the editor of two book volumes: *Software Fault Tolerance* (New York: Wiley, 1995) and *The Handbook of Software Reliability Engineering* (Piscataway, NJ: IEEE and New York: McGraw-Hill, 1996). Dr. Lyu initiated the First International Symposium on Software Reliability Engineering (ISSRE) in 1990. He was the program chair for ISSRE'96 and General Chair for ISSRE'2001. He was also PRDC'99 program cochair, WWW10 program cochair, SRDS'2005 program cochair, and PRDC'2005 general cochair, and served in program committees for many other conferences including HASE, ICECCS, ISIT, FTCS, DSN, ICDSN, EUROMICRO, APSEC, PRDC, PSAM, ICCCN, ISESE, and WI. He has been frequently invited as a keynote or tutorial speaker to conferences and workshops in the US, Europe, and Asia. He served on the editorial board of *IEEE Transactions on Knowledge and Data Engineering* and has been an associate editor of the *IEEE Transactions on Reliability* and *Journal of Information Science and Engineering*. He is a fellow of the IEEE for his contribution to software reliability engineering and software fault tolerance.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.

Authorized licensed use limited to: Chinese University of Hong Kong. Downloaded on February 11,2021 at 08:55:26 UTC from IEEE Xplore. Restrictions apply.