Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 79 (2006) 132–146
The Journal of Systems and Software
www.elsevier.com/locate/jss

Architecture-based software reliability modeling

Wen-Li Wang [a,*], Dai Pan [b], Mei-Hwa Chen [b]

[a] School of Engineering and Engineering Technology, Penn State Erie, The Behrend College, United States
[b] Computer Science Department, State University of New York at Albany, United States

Received 19 May 2005; received in revised form 2 June 2005; accepted 5 September 2005
Available online 21 October 2005

Abstract

Reliability is one of the essential quality requirements of software systems, especially for life critical ones. Software reliability modeling provides a means for estimating reliability of software, which facilitates effective decision making for quality control. Existing models either use a black-box approach that is based on test data of the whole system or a white-box approach that uses components' reliabilities and inter-component interactions. While the black-box approach is applicable to the late phases of software development, the white-box approach can support decisions on early component reuse and later component upgrades and updates. However, the white-box based models have traditionally considered only homogeneous software behaviors. For this reason, we extended the white-box to an architecture-based approach, utilizing the characteristics of architectural styles to capture design information and to realize non-uniform heterogeneous software behaviors. Adhered to the behaviors, a state machine of a discrete-time Markov model is constructed and employed to compute software reliability. Our approach allows reliability to be modeled at architectural design stage to decide components for reuse as well as later testing and maintenance phases for continuous software evolution. In contrast to the black-box approach, the model only needs to retest the influenced portions for a behavioral or structural change, instead of the complete system. This model yields a compelling result after being applied to an industrial real-time component-based financial system. We believe that this architecture-based modeling technique can have a great potential for use to effectively improve software quality.

Keywords: Software architecture; Architectural style; Markov model; Reliability estimation

1. Introduction

Software plays an important role in our daily life, as software failure can lead to a huge economic loss or even endanger lives. Thus, reliability is a critical quality requirement of software and is an important measurement for use in quality assurance (Laprie, 1995; Lopez-Benitez, 1994). Improving software reliability effectively requires that software design be refined and design defects be identified as early as possible in the software development life cycle so that cost and efforts can be reduced.

Over the past three decades, a number of studies have been conducted for measuring the reliability of software, and many analytical models (Chen et al., 2001, 1995; Farr, 1996; Goel and Okumoto, 1979; Gokhale et al., 1998a,b,c; Musa et al., 1987) have been developed. Most of these analytical models observed the behaviors of software by following an operational profile. They measured software reliability using the collected data for the whole system in the observation period. However, these models using statistical means could be expensive for obtaining sufficient data, and the timing of reliability modeling was subject to the late phase of a software process. Moreover, these models adopt a black-box approach without looking into internal system structures, so that a small modification would require the complete software to be retested. In other words, the statistical means and the black-box approach hinder the broad use of these models.

To address system structure, a white-box approach (Cheung, 1980) that utilized discrete-time Markov chains

* Corresponding author. Tel.: +1 814 8986 386; fax: +1 814 8986 125.
E-mail addresses: wxw18@psu.edu (W.-L. Wang), dpan@cs.albany.edu (D. Pan), mhc@cs.albany.edu (M.-H. Chen).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.09.004

was developed to model simple homogeneous software structures, such as sequential and branching transitions. The sequential behaviors mean that the transitions are from one module to another in sequence, while the branching involves predicates to decide which module to execute next. With increasing demands on software functionalities, modern software often embodies not just homogeneous architecture, but also heterogeneous architecture (Shaw, 1991) to achieve multiple quality and performance requirements. For example, parallel computations are widely used to increase performance, and back-up components are often used to provide fault tolerance capabilities. Consequently, homogeneous Markov-based modeling approaches are no longer suitable for modeling complex software with heterogeneous infrastructures. To address heterogeneity, software reliability modeling techniques must take various software architectures into account to yield a better estimation. In this paper, we show how to incorporate the concepts of software architecture to achieve this goal.

Software architecture, consisting of a set of *components*, *connectors* and *configurations*, describes the structures of software at an abstract level (Bass et al., 1998; Garlan and Shaw, 1993; Perry and Wolf, 1992; Shaw et al., 1995). To design software architecture, architectural styles are commonly used frameworks to characterize the configurations of components and connectors (Allen and Garlan, 1994; Clements, 1996; Garlan, 1995). In the past decade, many architectural styles have been classified (Garlan and Shaw, 1993; Shaw and Garlan, 1994; Taylor et al., 1996; Tracz, 1995) with new styles constantly emerging (Medvidovic et al., 1997; Shaw and Garlan, 1996). A practitioner now faces the challenge of selecting suitable styles to model software architecture and to ensure design quality. In the previous studies (Chen et al., 1998; Wang et al., 1999a,b), we have shown that software systems consisting of different architectural styles could provide different performance and/or availability, and the differences can be observed at the early high-level architectural design stage. In this paper, we show that the reliability of software can also be measured at design stage by analyzing the characteristics of the embedded architectural styles to confine the interactions and interrelationships among software components. The identified interactions are modeled into a state machine, and the interaction frequencies and component reliabilities will determine the transition probabilities between states to compute software reliability. The computation of a transition probability only has to consider the involving components, not the whole system. Our architecture-based model, also a white-box approach, allows testing to be conducted between every two adjacent components, similar to a pair-wise Integration testing. The testing results of those, not of a complete system, are then used to compute the reliability of software. For this nature, component reuse at the architectural design stage only needs to adjust the transition probabilities for changed configurations, while the future component upgrades and updates only require retesting the influenced portions.

The aim of our work is to develop an architecture-based reliability model that takes heterogeneity of software architecture into account to address various types of component interactions. We use a white-box approach built on top of a traditional Markov-based reliability model. A state machine is constructed to address software architecture, in which the realization of component interactions and configurations follows the regulations of the identified architectural styles in the system. Since each architectural style has its own characteristics and constraints that clearly define the configurations of the components and connectors, the construction of the state machine based on styles clarifies the non-uniform behaviors of a system, thus preventing ambiguity. Our state machine addresses the characteristics of heterogeneous architecture, which differentiates our model from the existing work.

The reliability of a system is measured as follows. The first step is to identify the embedded architectural styles, and for each identified style we develop a state model to address the encapsulated behaviors and constraints. Second, a global state machine for the whole system is formulated, combining all the individual state models. Finally, with this global state machine our architecture-based model that extends the traditional Markov-based Models (Cheung, 1980; Musa et al., 1987) can then be used to compute software reliability. In this paper, we demonstrate the model using four architectural styles, including batch-sequential, parallel/pipe-filter, fault tolerance, and call-and-return styles. We first model each individual style and then elaborate a scheme that combines individual state models to address the heterogeneity of different styles. An example of applying the model is given and a case study on an industrial real-time component-based financial system shows the usefulness of this architecture-based modeling approach. Our model utilizes high-level architectural styles to perceive component interactions and interrelationships, and does not require a complete system retest for partial changes. Thus, not only can it be used at an early design stage of the software development life cycle to decide component reuse, but also can be applied effectively in place of component upgrades and updates at the late phases.

The paper is organized as follows: in Section 2 we present the foundations of our architecture-based software reliability model. The details of each style-based model are given in Section 3, and the complete architecture-based model is described in Section 4. Section 5 shows a walk-through modeling example, and Section 6 presents an experiment on an industrial system. Section 7 gives a brief overview of the related work on software reliability measurements. Conclusions and future work are discussed in Section 8.

2. Architecture-based model foundations

Our architecture-based reliability model exploits discrete-time Markov chains as the building blocks to model software and to compute its reliability. The discrete-time

Markov chains are described in the Appendix A. For a discrete-time Markov model, the transitions occur either at discrete-time intervals or at discrete events, and the transition probabilities follow a discrete distribution. If a software system is a set of software components functioning homogeneously, we can perceive the transition from one component to another as the occurrence of a discrete event. Such a mapping from software to state machine is under the assumption that a software process follows a Markov process, which means that a transition to the next state will depend on the current state only. Although one could argue that in reality such assumption may not always be held, the studies by Pinkerton (1968) and Ramamoorthy (1966) concluded that many experiments for large-scale software closely converged to the Markov process results after a long duration. Accordingly, the discrete-time Markov models have been incorporated into a number of reliability modeling tools, such as GRAMS and MARK1 (Johnson and Malek, 1988).

Traditional discrete-time Markov-based reliability models are good for tackling homogeneous software behaviors, including sequential, branching and looping structures. A sequential behavior is an ordered execution of software components, a branching behavior is similar to a conditional statement that has alternative choices, and a looping structure is the repetition of the executions of a set of components. Although these models cover a wide range of module-based software, they are insufficient to model more complex structures as well as heterogeneous architecture. For example, a parallel architecture has multiple components running concurrently, a fault tolerant system has backup components compensating the failure of the others, or a client/server structure has client components calling the server components an indefinite number of times. Such structures may have multiple activities taking place simultaneously, and require some specific actions based on the running situations. Therefore, heterogeneous software, which is a composition of different structures, makes the use of the traditional models for reliability measurement very challenging.

To tackle the shortcomings of homogeneous models, our objective is to construct a state machine capable of addressing both homogeneity and heterogeneity. According to Webster (Agnes, 2000), a state can be a set of circumstances or attributes characterizing a system at a given condition or activity. We apply this definition to software systems and define our state model as follows:

- A *state* is a set of *circumstances* characterizing a system at a given *condition*.
- A *transition* is a passage from one state to another, whose *transition probability* is the probability of undergoing this *transition*.

A *condition* is referred to an instance of a set of components from receiving the control to the release of control, and a *circumstance* is an event that activates one of the components retaining the control. Note that multiple inputs and outputs can be modeled by introducing a *super-initial* state connecting to all the input states and a *super-final* state connecting from all the output states with individual transition probabilities.

Based on the above definition, a single circumstance or multiple circumstances can occur in one state, and the transition from one state to another can involve synchronization. To model heterogeneity, we construct a different non-negative $n \times n$ stochastic matrix T as shown in Eq. (1), which retains the properties of those two lemmas of a discrete-time Markov model in Appendix A.

$$T = \begin{array}{c} \\ S \\ F \\ s_1 \cdots s_k \end{array} \begin{array}{c} \begin{array}{ccc} S & F & s_1 \cdots s_k \end{array} \\ \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ B_1 & B_2 & M \end{bmatrix} \end{array}, \quad B_1 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ R_k^+ \end{bmatrix}, \quad B_2 = \begin{bmatrix} 1 - \sum_{j=1}^{k} M(1,j) \\ \vdots \\ 1 - \sum_{j=1}^{k} M(k-1,j) \\ 1 - R_k^+ \end{bmatrix} \tag{1}$$

Similar to a homogeneous model, states S and F are two absorbing states representing the successful output state and failure state, respectively, which adhere to the discrete-time Markov models described in Appendix A. Thus, the transition probability from S to S is equal to 1, so is from F to F. Therefore, $D_1 = [1]$ and $D_2 = [1]$ and both are irreducible. The inner matrix M is a $k \times k$ matrix with only transient states, in which s_1 is the entry state and s_k is the exit state. The sum of the last row in M is equal to 0, because the exit state s_k can only reach either S or F. In addition, B_1 and B_2 are two vectors. All the entries of B_1 are 0 except the final entry $B_1(k)$ with value R_k^+, which is the probability that component(s) in the final exit state s_k is(are) executed successfully. $B_2(k)$ is equal to $1 - B_1(k)$, and the rest entries of B_2 are equal to 1 minus the sum of the row in M for each row. By Lemma 1 in Appendix A, T of Eq. (1) is a standardized stochastic matrix with each row sum equal to 1. This ensures the spatial radius $\rho(M) < 1$. By Lemma 2 in Appendix A, $\rho(M) < 1$ implies $(I - M)^{-1}$ is non-singular, and $(I - M)^{-1} = \sum_{i=0}^{\infty} M^i$.

In our model, software reliability R is computed based on Eq. (2) not Eq. (A.3), because software behavior in the exit state s_k determines the value of R_k^+. Basically, if only one component is running in s_k, which is a one-to-one mapping, we have $R_k^+ = R_k$; otherwise, the computation of R_k^+ needs to cope with the structure of multiple components in that exit state. In Eq. (2), $|I - M|$ is also not equal to 0, because $(I - M)^{-1}$ is non-singular. In the next two sections, we will demonstrate the usage of architectural styles to comprehend different system structures and transform the comprehension into a state model so that the transition matrix M can be constructed accordingly to compute software reliability.

$$R = (-1)^{k+1} R_k^+ \frac{|E|}{|I - M|} \tag{2}$$

3. Style-based reliability modeling

The theme of our study is to evaluate software reliability taking into account system structures. Software architectural styles depict abstract structures, regulate the topology of configuration, define the constraints, and embed high-level semantics (Garlan, 1995; Garlan et al., 1994). They facilitate the discussion and communications of a software system at a higher level among developers (Garlan and Shaw, 1993; Shaw and Garlan, 1994). In our model, we utilize styles to characterize the organization of a software system, so as to comprehend complex and heterogeneous system structures.

Three attributes are required in our software reliability modeling: system architecture, component reliabilities, and transition probabilities for every two connecting components. System architecture is classified into architectural styles to describe different component interactions and intercommunications. The reliability of a component is either measured by traditional approaches (Laprie and Kanoun, 1996; Littlewood, 1979; Musa et al., 1987) or through the inter-component dependency approach proposed by Hamlet et al. (2001). Transition probabilities are observed from an operational profile if available. Otherwise, a pragmatic approach, which exploits the number of transition occurrences among components during test, will be discussed in different styles and formulated in Section 4.

With these three attributes, we transform system architecture into a state machine, in order to construct a Markov model to compute software reliability. Component reliabilities and transition probabilities among components are used to compute transition probabilities among states. In Section 2, we defined a state as a set of circumstances characterizing a system at a given condition. When mapped to software architecture, a state represents a non-empty set of software components running at a given time duration from the beginning to the end of the execution of the components. A state transition occurs when the time duration ends, and the transition probability is the likelihood of undergoing such a state transition. In this section, we will discuss the reliability modeling for homogeneous software with one single architectural style. Four commonly used styles, including batch-sequential, parallel/pipe-filter, call-and-return, and fault tolerance styles are demonstrated individually. Note that the modeling approach can also be applicable to other styles with certain modifications and extensions.

Throughout this paper, the following notations are used to describe our model. Notation r_x represents the reliability of component c_x, and $t(x,y)$ returns the number of transition occurrences from component c_x to c_y. R_i is the probability of components successfully executed in state s_i, and $P_{i,j}$ is the chance from state s_i to s_j regardless of fault propagation. The product of R_i and P_{ij} or simply $R_iP_{i,j}$ is perceived as the transition probability from state s_i to s_j, under the realization that components in s_i function correctly.

3.1. Batch-sequential style

In a batch-sequential style, components are executed in a sequential manner, i.e., only a single component is executed in any instance of time. For example, a bank performs a batch of transaction updates to a master file in sequence. From this type of architecture, the control is transferred to one (and only one) of its successors upon the completion of a component. The selection of the succeeding component is either probabilistic (if existing more than one successor) or deterministic (if having exactly one successor).

One of the instances of this style is modeled as shown in Fig. 1(a), where $c_1, c_2, \ldots, c_k$ are software components and component c_2 transfers control to one of its branching subsequent components. Since only one component is executed at a time, an incomplete execution of a component will not proceed to the next component. Accordingly, the state machine for batch-sequential style software is modeled as follows. A state represents an execution of a component. A transition from one state to another takes place, when the execution is completed and the control of the system is released to the next component. The transformation from the architecture to a state model can be viewed as a mapping of one component to one state, shown in Fig. 1(b), where $s_1, s_2, \ldots, s_k$ are the mapping states to components $c_1, c_2, \ldots, c_k$.

For those k mapping states, a $k \times k$ matrix M is constructed to compute software reliability. Since the batch-sequential style preserves only the common branching and sequential behaviors, the transition probability from state s_i to state s_j, as stored in entry $M(i,j)$ of Eq. (3), is simply identical to Eq. (A.2) in Appendix A.

$$M(i,j) = \begin{cases} R_iP_{ij}, & \text{state } s_i \text{ reaches state } s_j \text{ and } i \neq k, \\ 0, & \text{otherwise,} \end{cases} \quad \text{for } 1 \leqslant i,j \leqslant k \tag{3}$$

Note that R_i is equal to r_i, the reliability of component c_i that runs in state s_i. P_{ij} in this style is the probability from component c_i to c_j, and can be either observed from an operational profile if available, or computed from $t(i,j)$

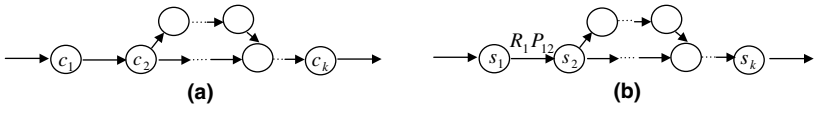

Fig. 1. Batch-sequential style: (a) architecture and (b) state model.

the number of transition occurrences between components as follows:

$$P_{i,j} = t(i,j) \Big/ \sum_{n=1}^{k} t(i,n) \tag{4}$$

3.2. Parallel/pipe-filter styles

Parallel and pipe-filter styles are commonly used in concurrent execution environments, in which a set of components is usually executed simultaneously to improve performance. These two styles are quite similar; the main difference is that parallel computation is generally in a cooperative multi-processors environment, whereas the pipe-filter style is mainly in a single processor, multi-processes environment.

An example of these styles is depicted in Fig. 2(a), where components $c_2, c_3, \ldots, c_{k-1}$, in the dotted oval, are running concurrently. These concurrent components cooperatively work on a partition of outputs produced by component c_1 and synchronously release the control to their subsequent component c_k. We model these styles as follows: a state represents either a single component execution or a set of concurrent components' executions, spanning the time from the beginning to the completion of the executions of all components in the set. A state transition takes place when a transient state completes the required operations. Therefore, the modeling of a set of cooperative concurrent components in one single state conforms to the essence of simultaneous transitions of concurrency in these styles.

Fig. 2(b) shows the transformed state diagram of Fig. 2(a). Components c_1 and c_k are assigned to their individual states s_1 and s_3, and the set of concurrent components from c_2 to c_{k-1} is modeled into one single state s_2. This implies that the control will release from c_1 to the set of $k-2$ concurrent components simultaneously. The concurrent behavior differs from the sequential behavior discussed in the previous section, so that the component-to-state transformation cannot be a one-to-one mapping. Consequently, component c_i may not run in state s_i and c_j may not run in s_j. Therefore, for the transition probability R_iP_{ij} from state s_i to s_j, the computation of R_i needs to consider two situations, i.e., state s_i has either one component running or a set of concurrent components running simultaneously.

The following shows the computation of R_i in the transformed state machine.

$$R_i = \begin{cases} r_\alpha, & \text{only } c_\alpha \text{ in } s_i, \\ \prod r_\alpha, & \forall c_\alpha \text{ in } s_i, \end{cases} \quad 1 \leqslant i \leqslant m \tag{5}$$

where m is the total number of states and the universal quantifier "$\forall$" is denoted to represent "for all" components in a state. From Eq. (5), R_1 is equal to r_1, for c_1 is in s_1. R_2 is equal to $\prod r_\alpha$ with $\alpha = 2, 3, \ldots, k-1$, for $c_2, c_3, \ldots, c_{k-1}$ are in s_2. R_3 is equal to r_k, for c_k is in s_3.

For the computation of P_{ij}, here we discuss the computation of this example and will generalize a computation formula for heterogeneous and complex architecture in Section 4. In Fig. 2(b), there can be two possible transitions, one from s_1 to s_2 and the other from s_2 to s_3. It is thus necessary to compute P_{12} and P_{23}. In this style, we presume a transition to any component in a parallel component set is basically to the whole set. Therefore, the transition probability P_{12} in this case is equal to 1 through the consideration of any one of the components, such as $t(1,2)/t(1,2) = 1$. Similarly, P_{23} is computed as 1.

In summary, the state machine of Fig. 2(b) consists of $m = 3$ states out of k components. The transformation to this state machine not only encapsulates the characteristics of parallel computation, but also maintains the Markov properties, meaning $0 \leqslant R_iP_{ij} \leqslant 1$. For the modeling of this state machine, a 3×3 transition matrix M is constructed as below.

$$\begin{cases} M(1,2) = R_1P_{12} = r_1, \\ M(2,3) = R_2P_{23} = \prod_{\alpha=2}^{k-1} r_\alpha, \quad 1 \leqslant i,j \leqslant m \\ M(i,j) = 0, \text{ otherwise} \end{cases} \tag{6}$$

The reliability for the architecture of Fig. 2(a) can be computed as $(r_1)(r_2 * r_3 * \cdots * r_{k-1})(r_k) = \prod_{\alpha=1}^{k} r_\alpha$. With the transition matrix M of Eq. (6), the reliability $R = (-1)^{k+1} R_k^+ \frac{|E|}{|I-M|}$ based on Eq. (2) derives the same result as $(-1)^4 (r_k)\left(\prod_{\alpha=1}^{k-1} r_\alpha\right)/1 = \prod_{\alpha=1}^{k} r_\alpha$, where

$$|I - M| = \left| \begin{bmatrix} 1 & -r_1 & 0 \\ 0 & 1 & -\prod_{\alpha=2}^{k-1} r_\alpha \\ 0 & 0 & 1 \end{bmatrix} \right| = 1,$$

$$|E| = \left| \begin{bmatrix} -r_1 & 0 \\ 1 & -\prod_{\alpha=2}^{k-1} r_\alpha \end{bmatrix} \right| = \prod_{\alpha=1}^{k-1} r_\alpha$$

and $R_k^+ = r_k$.

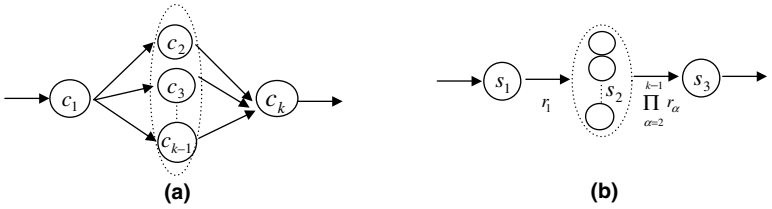

Fig. 2. Parallel/pipe-filter styles: (a) architecture and (b) state model.

3.3. Fault tolerance

A fault-tolerant architectural style is often adopted to improve availability of software. Typically, this style consists of a set of fault forbearing components, including primary components and backup components, in which the backup components compensate for the failures of the primary components. When a primary component fails, one of its backup components instantaneously takes over the responsibility and becomes a new primary component. If the new primary component also fails, another backup component will soon take over. This scenario persists as long as there is a backup component available to keep software up and running. Note that the transition probability to or from a set of fault tolerant components is presumed identical, but the implementations of the fault-tolerant components are not limited to the same algorithms or data structures, so that the reliabilities of these components may vary.

The modeling of a fault tolerant style is similar to the parallel style. If only one single component is running, a state represents an execution of that component. Otherwise, we model the scenario of a set of fault tolerant components by a state, spanning the time from the beginning of the primary components to the completion of the primary and activated backup components. The state transition takes place when the execution in the current state is completed, and the control authority is released to the component(s) in the next state.

An example of the fault tolerant style is presented in Fig. 3(a). Inside the dotted rectangle, there is a set of fault tolerant components, in which c_2 is the primary component with $c_3, c_4, \ldots, c_{k-1}$ as its backup components. In addition, we assume that the system requires only one component in the set running successfully to continue the execution. Fig. 3(b) shows the transformed state model from the architecture in Fig. 3(a). Component c_1, which is not a fault tolerant component, is mapped to state s_1. The set of fault tolerant components $c_2, c_3, \ldots, c_{k-1}$ is modeled into state s_2. Similar to c_1, component c_k is mapped to a state s_3. Consequently, with a set of $k-2$ fault tolerant components out of k components in the architecture, the state diagram consists of a total of only three states. Similar to the parallel style, the number of transformed states is not identical to the number of components.

For the transition probability R_iP_{ij} from state s_i to s_j, the computation of R_i needs to take into account two situations, i.e., state s_i has only one component running or has a set of fault tolerant components cooperating to prevent failures. When only one component is running, the success of s_i depends on the reliability of this running component. Otherwise, if a set of fault tolerant components are cooperating, the success of s_i is the case that at least one component does not fail, i.e., to exclude the failure probability that the set of fault tolerant components all fail. The following computes R_i for software with a total number of m states.

$$R_i = \begin{cases} r_\alpha, & \text{only } c_\alpha \text{ in } s_i, \\ 1 - \left(\prod_\alpha (1 - r_\alpha)\right), & \forall c_\alpha \text{ in } s_i, \end{cases} \qquad 1 \leqslant i \leqslant m \tag{7}$$

For the computation of P_{ij}, we discuss this example here and leave the heterogeneity issues to Section 4. Basically, it is necessary to compute P_{12} and P_{23}, because in the transformed state machine of Fig. 3(b) there can be two possible transitions. One is from s_1 to s_2 and the other is from s_2 to s_3. In this style, we presume a transition to any component in a fault tolerant component set is basically to the whole set. Therefore, the transition probability P_{12} in this case is equal to 1 by taking any one of the components into consideration, e.g., $t(1,2)/t(1,2) = 1$. Similarly, P_{23} is computed as 1.

For the state machine shown in Fig. 3(b), it consists of $m = 3$ states out of a total of k components. The transformation to this state machine depicts the fault tolerance features, and maintains the Markov properties, i.e., $0 \leqslant R_iP_{ij} \leqslant 1$. For the modeling of this state machine, a 3×3 transition matrix M is constructed as below.

$$\begin{cases} M(1,2) = R_1P_{12} = r_1, \\ M(2,3) = R_2P_{23} = 1 - \left(\prod_{\alpha=2}^{k-1}(1 - r_\alpha)\right), \\ M(i,j) = 0, \ \text{otherwise}, \end{cases} \qquad 1 \leqslant i, j \leqslant m \tag{8}$$

The reliability for the architecture of Fig. 3(a) can be computed as $(r_1)(1 - (1 - r_2) * (1 - r_3) * \cdots * (1 - r_{k-1}))(r_k) = r_1\left(1 - \prod_{\alpha=2}^{k-1}(1 - r_\alpha)\right)r_k$. With M in Eq. (8), the reliability based on Eq. (2) derives the same result as $(-1)^4(r_k)\left(r_1\left(1 - \prod_{\alpha=2}^{k-1}(1 - r_\alpha)\right)\right)/1 = r_1\left(1 - \prod_{\alpha=2}^{k-1}(1 - r_\alpha)\right)r_k$, where

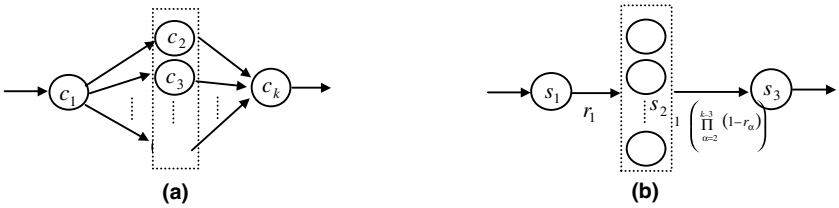

Fig. 3. Fault tolerance style: (a) architecture and (b) state model.

$$|I-M| = \left| \begin{bmatrix} 1 & -r_1 & 0 \\ 0 & 1 & -\left(1-\prod_{\alpha=2}^{k-1}(1-r_\alpha)\right) \\ 0 & 0 & 1 \end{bmatrix} \right| = 1,$$

$$|E| = \left| \begin{bmatrix} -r_1 & 0 \\ 1 & -\left(1-\prod_{\alpha=2}^{k-1}(1-r_\alpha)\right) \end{bmatrix} \right| = r_1\left(1-\prod_{\alpha=2}^{k-1}(1-r_\alpha)\right)$$

and $R_k^+ = r_k$.

3.4. Call-and-return

Call-and-return style is one of the most prevalent designs for both standalone and distributed systems. The evolutions began from procedure calls, client–server architecture, to object-oriented message passing mechanism. In this style, a calling component (caller) can invoke a called component (callee) a number of times, and each time the callee returns the requested services back to the caller. For each invocation, the caller performs a context switch by granting a temporary control to its callee and then waits for its response. Once receiving the response, the caller resumes the execution from where it left. Eventually, the caller transfers its complete control authority to its subsequent component.

An example of the call-and-return style is shown in Fig. 4(a), where component c_1 is a caller to its callee c_2, and c_1 may invoke c_2 from zero to many times. Therefore, component c_1 may transit the control authority to c_3 at once or after no more calls to its callee c_2. The state machine for this example is modeled as follows. A state represents an execution of a component. A transition from one state to another takes place when the execution is completed and the control of the system transfers to the next component, or when the execution encounters a context switch and the control temporarily transfers to the called component. The transformation from the architecture to the state model is simply a one-to-one mapping of a component to a state.

Fig. 4(b) is the state model that is a one-to-one mapping from the architecture of Fig. 4(a). The states s_1, s_2, and s_3 in the state machine represent the executions of components c_1, c_2, and c_3, respectively. In this machine, state s_1 can transit to s_3 after visiting s_2 many times or without even once. Because of the one-to-one mapping, the transformation is similar to the batch-sequential style in Section 3.1. Consequently, R_i is equal to r_i and P_{ij} can be computed by Eq. (4). R_i and P_{ij} are then used to compute the transition probability for the transformed state model of this style. In Section 4, the computation of R_i and P_{ij} will be generalized to facilitate the integration of this style with the other styles.

There is a modeling challenge in this style, because a caller may call a callee from zero to an indefinite number of times. In Fig. 4(a), component c_1 can call c_2 from zero to many times. In this case, state s_1 was visited only once before entering state s_3 in Fig. 4(b), regardless of how many times state s_2 was visited. Therefore, the transition from state s_1 to s_2 should have the reliability of component c_1 considered only once. Since the reliability of the one time execution of component c_1 will be considered in the transition from state s_1 to s_3, the solution for the transition probability from state s_1 to s_2 is thus modeled as P_{12} instead of R_1P_{12} to prevent a redundant modeling of component c_1. For the transition from state s_1 to s_3, it is a sequential transition similar to a batch-sequential style. Therefore, the transition probability is modeled as R_1P_{13}, implying the successful execution of component c_1 that later releases the control to c_3. Note that the transition probability R_1P_{13} resolves the aforementioned one time execution of component c_1, no matter how many times c_1 calls c_2. Likewise, the transition probability from state s_2 to s_1 is modeled as R_2P_{21}, because component c_2 is invoked every time it is called and then returns to c_1.

For the state machine of the call-and-return architecture in Fig. 4(b), a 3×3 matrix M is constructed as follows:

$$\begin{cases} M(1,2) = P_{12} = \frac{t(1,2)}{t(1,2)+t(1,3)}, \\ M(2,1) = R_2P_{21} = r_2, \\ M(1,3) = R_1P_{13} = \frac{r_1\cdot t(1,3)}{t(1,2)+t(1,3)}, \\ M(i,j) = 0, \text{ otherwise}, \end{cases} \quad \text{for } 1 \leqslant i,j \leqslant 3 \qquad (9)$$

In Fig. 4(a), let p_{12} be $\frac{t(1,2)}{t(1,2)+t(1,3)}$, p_{21} be 1, and p_{13} be $\frac{t(1,3)}{t(1,2)+t(1,3)}$, the reliability for the architecture can be computed as $(r_1p_{13})(r_3) + (p_{12})(r_2p_{21})(r_1p_{13})(r_3) + [(p_{12})(r_2p_{21})]^2(r_1p_{13})(r_3) + [(p_{12})(r_2p_{21})]^3(r_1p_{13})(r_3) + \ldots = r_1p_{13}r_3/(1-p_{12}r_2)$. With M in Eq. (9), the reliability computation using Eq. (2) derives the same result as $(-1)^4(r_3)(r_1p_{13})/(1-p_{12}r_2) = r_1p_{13}r_3/(1-p_{12}r_2)$, where

$$|I-M| = \left| \begin{bmatrix} 1 & -p_{12} & -r_1p_{13} \\ -r_2 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right| = 1-p_{12}r_2,$$

Fig. 4. Call-and-return style: (a) architecture and (b) state model.

$$|E| = \left| \begin{bmatrix} -p_{12} & -r_1 p_{13} \\ 1 & 0 \end{bmatrix} \right| = r_1 p_{13}$$

and $R_k^+ = r_3$.

4. Architecture-based reliability modeling

In Section 3, we have demonstrated the reliability modeling for software with homogeneous architectural style. To model reliability of software with heterogeneous architectural styles, a systematic approach is presented in the following.

Step 1: *Identify the architectural styles in a system, based on the design specification of a system.*

From the high-level design specification, the interactions among software components can be examined. The interactions may involve components operating in some sequences, requesting services from others, or functioning as a group to fulfill a requirement. The architectural styles can be characterized from the observations of the interactions among components. In a heterogeneous architecture system, a component may belong to more than one style; for example, a component may run sequentially for some conditions as in the batch-sequential style and call for services under other conditions as in the call-and-return style.

Step 2: *Transform individual architectures of the identified architectural styles into state models.*

Once the architectural styles in a system are identified, the components and their interactions in each style become a sub-graph of the complete architecture view. Each sub-graph is a single-style architecture that can be transformed into a state model, as discussed in Section 3. The transformation can be either a one-to-one mapping or a many-to-one mapping, depending on the characteristics of the components' interactions. The state transition probabilities for the state models can later be computed according to the distinctiveness of the styles.

Step 3: *Integrate these state models into a global state model of the system, based on the overall architecture of a complete system.*

The integration simultaneously addresses different component interactions among associated state models. The process incorporates all of the individual state models to form a global state model. The global state model encompasses the interrelationships of the state models, based on the composed architectural styles.

To describe the process of integration, we first define a set of notations in Table 1 and characterize three situations in which components may be involved in more than one of the aforementioned four styles.

1. Callees are invoked by callers only.
2. Parallel components are independent of fault tolerant components.
3. A set of cooperating parallel or fault tolerant components can also be a caller or a callee.

Table 1
Notations of different state sets

Software G consists of x components whose transformed state model δ is a set of m states
$G = \{c_\alpha \mid$ component $c_\alpha \in B \cup P \cup F \cup S \cup C,\ 1 \leqslant \alpha \leqslant x\}$
B: a set of components in the batch sequential styles
P: a set of components in the parallel styles
F: a set of components in the fault-tolerant styles
C: a set of callers in the call-and-return styles
S: a set of callees in the call-and-return styles
where
$\delta = \delta_B \cup \delta_P \cup \delta_F \cup \delta_C \cup \delta_S$, $\lvert\delta\rvert = m$, $1 \leqslant i \leqslant m$
δ_B: a set of states for components in the batch sequential styles
$\delta_B = \{s_i \mid c_\alpha \in B, c_\alpha$ maps to state $s_i\}$
δ_P: a set of states for u parallel styles p_k, $P = \bigcup_{k=1}^{u} p_k$,
$\delta_P = \{s_i \mid p_k$ maps to state $s_i\}$
δ_F: a set of states for v fault-tolerant styles f_k, $F = \bigcup_{k=1}^{v} f_k$,
$\delta_F = \{s_i \mid f_k$ maps to state $s_i\}$
δ_C: a set of states for y callers θ_k in the call-and-return styles,
$C = \bigcup_{k=1}^{y} \theta_k$, $\delta_C = \{s_i \mid \theta_k$ maps to state $s_i\}$
δ_S: a set of states for z callees ϕ_k in the call-and-return styles,
$S = \bigcup_{k=1}^{z} \phi_k$
$\delta_S = \{s_i \mid \phi_k$ maps to state $s_i\}$

For the first situation, since a caller in δ_C may also be a callee in δ_S and vice versa, or be a batch-sequential component to transit to others, δ_C is drawn to intersect with δ_B and δ_S. However, a callee in δ_S cannot be a batch-sequential component in δ_B to transit to others; thus, $\delta_B \cap \delta_S = \phi$. The second situation implies $\delta_P \cap \delta_F = \phi$ for parallel and fault tolerant components being independent. For the third situation, δ_C and δ_S can also intersect with δ_P and δ_F. Fig. 5 summarizes all the possible relationships of the state sets.

Based on the relationships of the state sets, we can derive possible conditions of state transitions for the next step to construct a transition matrix and to compute software reliability. From Fig. 5, we can classify the state transitions into five types. The first type is for the transition from a caller to a callee. The second type is from a callee returning to a caller. The third one is from a batch-sequential state to a non-callee. The fourth type is from a non-caller/callee parallel state to a non-callee. The last one is from a non-caller/callee fault tolerant state to a non-callee. These five types can be further elaborated into 18 conditions. The following lists the 18 conditions under their own types using the notations in Table 1, in which "$s_i \rightarrow s_j$" represents a transition from state s_i of a state model to s_j of the other state model.

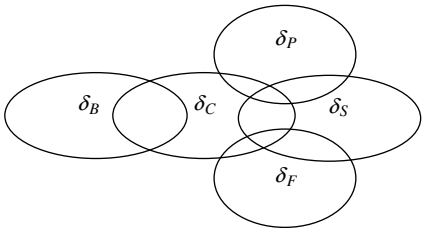

Fig. 5. The relationships of the state sets.

- Type 1: from a caller to a callee, $s_i \in \delta_C$ and $s_j \in \delta_S$, $s_i \rightarrow s_j$.
  - 1.1 $s_i \in \delta_C \cap \delta_P$, $s_j \in \delta_S \cap (\delta_P \cup \delta_F)$, the caller in parallel calls the callee in parallel or in fault tolerance.
  - 1.2 $s_i \in \delta_C \cap \delta_P$, $s_j \in \delta_S - (\delta_P \cup \delta_F)$, the caller in parallel calls the callee not in parallel and fault tolerance.
  - 1.3 $s_i \in \delta_C \cap \delta_F$, $s_j \in \delta_S \cap (\delta_P \cup \delta_F)$, the caller in fault tolerance calls the callee in parallel or in fault tolerance.
  - 1.4 $s_i \in \delta_C \cap \delta_F$, $s_j \in \delta_S - (\delta_P \cup \delta_F)$, the caller in fault tolerance calls the callee not in parallel and fault tolerance.
  - 1.5 $s_i \in \delta_C - (\delta_P \cup \delta_F)$, $s_j \in \delta_S \cap (\delta_P \cup \delta_F)$, the caller not in parallel and fault tolerance calls the callee in parallel or in fault tolerance.
  - 1.6 $s_i \in \delta_C - (\delta_P \cup \delta_F)$, $s_j \in \delta_S - (\delta_P \cup \delta_F)$, the caller not in parallel and fault tolerance calls the callee also not in parallel and fault tolerance.
- Type 2: from a callee returning to a caller, $s_i \in \delta_S$ and $s_j \in \delta_C$, $s_i \rightarrow s_j$.
  - 2.1 $s_i \in \delta_S \cap \delta_P$, $s_j \in \delta_C \cap (\delta_P \cup \delta_F)$, the callee in parallel returns to the caller in parallel or in fault tolerance.
  - 2.2 $s_i \in \delta_S \cap \delta_P$, $s_j \in \delta_C - (\delta_P \cup \delta_F)$, the callee in parallel returns to the caller not in parallel and fault tolerance.
  - 2.3 $s_i \in \delta_S \cap \delta_F$, $s_j \in \delta_C \cap (\delta_P \cup \delta_F)$, the callee in fault tolerance returns to the caller in parallel or in fault tolerance.
  - 2.4 $s_i \in \delta_S \cap \delta_F$, $s_j \in \delta_C - (\delta_P \cup \delta_F)$, the callee in fault tolerance returns to the caller not in parallel and fault tolerance.
  - 2.5 $s_i \in \delta_S - (\delta_P \cup \delta_F)$, $s_j \in \delta_C \cap (\delta_P \cup \delta_F)$, the callee not in parallel and fault tolerance returns to the caller in parallel or in fault tolerance.
  - 2.6 $s_i \in \delta_S - (\delta_P \cup \delta_F)$, $s_j \in \delta_C - (\delta_P \cup \delta_F)$, the callee not in parallel and fault tolerance returns to the caller also not in parallel and fault tolerance.
- Type 3: from a batch-sequential state to a non-callee, $s_i \in \delta_B$ and $s_j \in \delta - \delta_S$, $s_i \rightarrow s_j$.
  - 3.1 $s_i \in \delta_B$, $s_j \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$, the non-callee excludes parallel and fault tolerance.
  - 3.2 $s_i \in \delta_B$, $s_j \in (\delta_P \cup \delta_F) - \delta_S$, the non-callee is in parallel or in fault tolerance.
- Type 4: from a non-caller/callee parallel state to a non-callee, $s_i \in \delta_P - (\delta_C \cup \delta_S)$ and $s_j \in \delta - \delta_S$, $s_i \rightarrow s_j$.
  - 4.1 $s_i \in \delta_P - (\delta_C \cup \delta_S)$, $s_j \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$, the non-callee excludes parallel and fault tolerance.
  - 4.2 $s_i \in \delta_P - (\delta_C \cup \delta_S)$, $s_j \in (\delta_P \cup \delta_F) - \delta_S$, the non-callee is in parallel or in fault tolerance.
- Type 5: from a non-caller/callee fault tolerant state to a non-callee, $s_i \in \delta_F - (\delta_C \cup \delta_S)$ and $s_j \in \delta - \delta_S$, $s_i \rightarrow s_j$.
  - 5.1 $s_i \in \delta_F - (\delta_C \cup \delta_S)$, $s_j \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$, the non-callee excludes parallel and fault tolerance.
  - 5.2 $s_i \in \delta_F - (\delta_C \cup \delta_S)$, $s_j \in (\delta_P \cup \delta_F) - \delta_S$, the non-callee is in parallel or in fault tolerance.

Step 4: *Construct the transition matrix M based on the global state model of the system and compute the reliability of the system.*

The construction of this transition matrix M is to make use of the Markov model to compute software reliability. For this transition matrix with size $m \times m$, as shown in Eq. (10), the computations of its entries are corresponding to the conditions determined in *Step 3*. Intuitively, if there is no transition from one state to another, denoted as "$\nrightarrow$", the transition probability in that entry is simply a 0. However, if the conditions are from 1 to 6, meaning the relationship of a caller calling a callee, the transition probability is modeled as P_{ij}. In such a case, R_i, the reliability of the one time execution of caller, remains considered when transiting to a non-callee. For the other conditions, the transition probability will need to consider R_i, thus modeled as R_iP_{ij}.

$$M(i,j) = \begin{cases} 0, & s_i \nrightarrow s_j, \text{ or } i = m, \\ P_{ij}, & \text{conditions 1–6}, \\ R_iP_{ij}, & \text{otherwise}, \end{cases} \quad 1 \leqslant i,j \leqslant m \tag{10}$$

For the computation of R_i, there are three formulas for conditions from 7 to 18, as shown in Eq. (11). It mainly relies on the style and the number of components running in state s_i. Conditions 11–14 all have only one component c_α running; thus R_i is equal to the reliability of c_α, i.e., r_α. Note that conditions 13 and 14 are for a batch-sequential transition, while 11 and 12 are for a callee returning back to its caller. For conditions 7, 8, 15 and 16, multiple components are running concurrently. Therefore, R_i is equal to the product of the reliabilities of those running components. Similarly, conditions 9, 10, 17 and 18 have multiple fault tolerant components running. Consequently, R_i is computed as 1 minus the failure probability of all those fault tolerant components.

$$R_i = \begin{cases} r_\alpha, & \text{only } c_\alpha \text{ in } s_i, \text{ conditions 11–14}, \\ \prod\limits_{\alpha} r_\alpha, & \forall c_\alpha \text{ in } s_i, \text{ conditions } 7,8,15 \text{ and } 16, \\ 1 - \left(\prod\limits_{\alpha}(1-r_\alpha)\right), & \forall c_\alpha \text{ in } s_i, \text{ conditions } 9,10,17 \text{ and } 18, \end{cases} \quad 1 \leqslant i \leqslant m \tag{11}$$

The computation of P_{ij} also corresponds to the previously identified conditions and can be derived from function $n(i,j)$, as shown in Eq. (12), which returns the number of transition occurrences from state s_i to s_j.

$$P_{ij} = n(i,j) \Big/ \sum_{k=1}^{m} n(i,k), \quad 1 \leqslant i,j \leqslant m \tag{12}$$

For different conditions, the function $n(i,j)$ will take into account distinctive circumstances, e.g., component(s) in state s_i and s_j can be concurrent, fault tolerant, or others. If a state has multiple components running, only one of them will be chosen for modeling because a control transfer to one is basically to all. Note that function $n(i,j)$ is for the transition occurrences between states in the global state model, whereas function $t(i,j)$ in Section 3 was for the transition occurrences between components in the architecture view. The following shows the computation of $n(i,j)$ from function $t(i,j)$ as shown in Eq. (13).

$$n(i,j) = \begin{cases} t(\alpha,\beta), \text{ only } c_\alpha \text{ in } s_i, \text{ only } c_\beta \text{ in } s_j, \\ \quad \text{condition } 6, 12 \text{ or } 13 \\ t(x,\beta), \text{ any } c_x \in \forall c_\alpha \text{ in } s_i, \text{ only } c_\beta \text{ in } s_j, \\ \quad \text{condition } 2, 4, 8, 10, 15 \text{ or } 17 \\ t(\alpha,y), \text{ only } c_\alpha \text{ in } s_i, \text{ any } c_y \in \forall c_\beta \text{ in } s_j, \\ \quad \text{condition } 5, 11 \text{ or } 14 \\ t(x,y), \text{ any } c_x \in \forall c_\alpha \text{ in } s_i, \text{ any } c_y \in \forall c_\beta \text{ in } s_j, \\ \quad \text{condition } 1, 3, 7, 9, 16 \text{ or } 18 \end{cases} \tag{13}$$

Eqs. (12) and (13) show an approach to compute P_{ij} by considering the transition occurrences between states in the state model from the transition occurrences between components in the architecture view. In summary, the architecture-based reliability model computes software reliability based on a transition matrix derived from a global state model. This global state model is the integration of a number of state models, which are transformed from the architectural styles identified in a software system.

5. An example

This section demonstrates an example of our architecture-based reliability modeling approach for a heterogeneous software system. The architecture of this system is shown in Fig. 6(a) with a total of 16 components. In this architecture, components in the dotted oval run in parallel, whereas those in the dotted rectangle run in fault tolerance. In addition, components $c_1, c_2, \ldots, c_8$ have caller/callee relationships with c_1 as the initial caller. For the rest of the components, they run in a batch sequential manner.

Step 1:

The first step is to identify the architectural styles in the given system; there are four styles. Components c_1, c_9, c_{14}, c_{15}, and c_{16} are in batch-sequential. Components c_3, c_4, c_{10} and c_{11} are in parallel. Components c_6, c_7, c_{12} and c_{13} are in fault tolerance. There are several components in call-and-return styles. Component c_1 is a caller to callee c_2, or simultaneously to both c_3 and c_4. Both c_3 and c_4 are a caller to c_5, or to both c_6 and c_7. Both c_6 and c_7 are a caller to c_8. Furthermore, some components belong to multiple styles. Component c_1 is in a batch-sequential style, but is also a caller. Components c_3 and c_4 are a callee and run in parallel. Likewise, components c_6 and c_7 are a callee and run as fault tolerance.

Based on Table 1 and the above, we have

$$B = \{c_1, c_9\} \cup \{c_{14}, c_{15}, c_{16}\} = \{c_1, c_9, c_{14}, c_{15}, c_{16}\}$$
$$P = \{c_3, c_4\} \cup \{c_{10}, c_{11}\} = \{c_3, c_4, c_{10}, c_{11}\}$$
$$F = \{c_6, c_7\} \cup \{c_{12}, c_{13}\} = \{c_6, c_7, c_{12}, c_{13}\}$$
$$C = \{c_1\} \cup \{c_3, c_4\} \cup \{c_6, c_7\} = \{c_1, c_3, c_4, c_6, c_7\}$$
$$S = \{c_2\} \cup \{c_3, c_4\} \cup \{c_5\} \cup \{c_6, c_7\} \cup \{c_8\} = \{c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$$
$$G = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7 c_8, c_9, c_{10}, c_{11}, c_{12}, c_{13}, c_{14}, c_{15}, c_{16}\}$$

Step 2:

The second step is to transform those identified styles into state models. Since the component-to-state mapping can be many-to-one or one-to-one mapping, the total number of states in the state models can be different from the total number of components in the architecture. For example, parallel components c_3 and c_4 are only mapped to one state s_3 as shown in Fig. 6(b). Consequently, the transformed state models have a total of 12 states out of a total of 16 components. The following shows the corresponding transformation results from the component sets in *Step 1*.

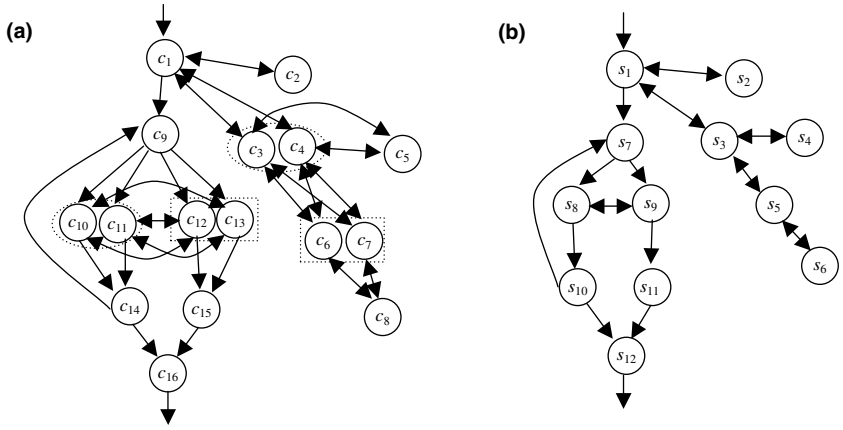

Fig. 6. (a) An architecture view and (b) a state view.

$\delta_B = \{s_1, s_7\} \cup \{s_{10}, s_{11}, s_{12}\} = \{s_1, s_7, s_{10}, s_{11}, s_{12}\}$
$\delta_P = \{s_3\} \cup \{s_8\} = \{s_3, s_8\}$
$\delta_F = \{s_5\} \cup \{s_9\} = \{s_5, s_9\}$
$\delta_C = \{s_1\} \cup \{s_3\} \cup \{s_5\} = \{s_1, s_3, s_5\}$
$\delta_S = \{s_2\} \cup \{s_3\} \cup \{s_4\} \cup \{s_5\} \cup \{s_6\} = \{s_2, s_3, s_4, s_5, s_6\}$
$\delta = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\}$

Step 3:

The integration is to simultaneously address different component interactions among associated state models, in order to model the whole system and derive a global state model. In this example, we assume the interaction behaviors are the same as those in *Step 3* of Section 4. Therefore, there are five types of state transitions with 18 conditions. The following lists the state transitions in this example, and categorizes for each one of them its type and condition.

- Type 1:

$s_1 \to s_2,\ \ s_1 \in \delta_C - (\delta_P \cup \delta_F),\ \ s_2 \in \delta_S - (\delta_P \cup \delta_F)$ condition 6
$s_1 \to s_3,\ \ s_1 \in \delta_C - (\delta_P \cup \delta_F),\ \ s_3 \in \delta_S \cap (\delta_P \cup \delta_F)$ condition 5
$s_3 \to s_4,\ \ s_3 \in \delta_C \cap \delta_P,\ \ s_4 \in \delta_S - (\delta_P \cup \delta_F)$ condition 2
$s_3 \to s_5,\ \ s_3 \in \delta_C \cap \delta_P,\ \ s_5 \in \delta_S \cap (\delta_P \cup \delta_F)$ condition 1
$s_5 \to s_6,\ \ s_5 \in \delta_C \cap \delta_F,\ \ s_6 \in \delta_S - (\delta_P \cup \delta_F)$ condition 4

- Type 2:

$s_2 \to s_1,\ \ s_2 \in \delta_S - (\delta_P \cup \delta_F),\ \ s_1 \in \delta_C - (\delta_P \cup \delta_F)$ condition 12
$s_3 \to s_1,\ \ s_3 \in \delta_S \cap \delta_P,\ \ s_1 \in \delta_C - (\delta_P \cup \delta_F)$ condition 8
$s_4 \to s_3,\ \ s_4 \in \delta_S - (\delta_P \cup \delta_F),\ \ s_3 \in \delta_C \cap (\delta_P \cup \delta_F)$ condition 11
$s_5 \to s_3,\ \ s_5 \in \delta_S \cap \delta_F,\ \ s_3 \in \delta_C \cap (\delta_P \cup \delta_F)$ condition 9
$s_6 \to s_5,\ \ s_6 \in \delta_S - (\delta_P \cup \delta_F),\ \ s_5 \in \delta_C \cap (\delta_P \cup \delta_F)$ condition 11

- Type 3:

$s_1 \to s_7,\quad s_1 \in \delta_B,\quad s_7 \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 13
$s_7 \to s_8,\quad s_7 \in \delta_B,\quad s_8 \in (\delta_P \cup \delta_F) - \delta_S$ condition 14
$s_7 \to s_9,\quad s_7 \in \delta_B,\quad s_9 \in (\delta_P \cup \delta_F) - \delta_S$ condition 14
$s_{10} \to s_7,\quad s_{10} \in \delta_B,\quad s_7 \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 13
$s_{10} \to s_{12},\quad s_{10} \in \delta_B,\quad s_{12} \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 13
$s_{11} \to s_{12},\quad s_{11} \in \delta_B,\quad s_{12} \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 13

- Type 4:

$s_8 \to s_9,\ \ s_8 \in \delta_P - (\delta_C \cup \delta_S),\ \ s_9 \in (\delta_P \cup \delta_F) - \delta_S$ condition 16
$s_8 \to s_{10},\ \ s_8 \in \delta_P - (\delta_C \cup \delta_S),\ \ s_{10} \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 15

- Type 5:

$s_9 \to s_8,\ \ s_9 \in \delta_F - (\delta_C \cup \delta_S),\ \ s_8 \in (\delta_P \cup \delta_F) - \delta_S$ condition 18
$s_9 \to s_{11},\ \ s_9 \in \delta_F - (\delta_C \cup \delta_S),\ \ s_{11} \in \delta - (\delta_P \cup \delta_F \cup \delta_S)$ condition 17

Step 4:

This final step is to construct a transition matrix M, and compute software reliability using the Markov model. *Step 3* lists a series of types and conditions, upon which the transition probability from one state to another can be computed. Based on the computation Eqs. (10)–(13), the transition matrix M can be derived as follows:

$$\begin{bmatrix}
0 & P_{1,2} & P_{1,3} & 0 & 0 & 0 & R_1P_{1,7} & 0 & 0 & 0 & 0 & 0 \\
R_2P_{2,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
R_3P_{3,1} & 0 & 0 & P_{3,4} & P_{3,5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & R_4P_{4,3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & R_5P_{5,3} & 0 & 0 & P_{5,6} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & R_6P_{6,5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & R_7P_{7,8} & R_7P_{7,9} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R_8P_{8,9} & R_8P_{8,10} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & R_9P_{9,8} & 0 & 0 & R_9P_{9,11} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & R_{10}P_{10,7} & 0 & 0 & 0 & 0 & R_{10}P_{10,12} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & R_{11}P_{11,12} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

where

$P_{1,2} = n(1,2)/(n(1,2)+n(1,3)+n(1,7))$
$= t(1,2)/(t(1,2)+t(1,3)+t(1,9))$ // condition 6
or $t(1,2)/(t(1,2)+t(1,4)+t(1,9))$ c_3 and c_4 are in parallel
$P_{1,3} = n(1,3)/(n(1,2)+n(1,3)+n(1,7))$
$= t(1,3)/(t(1,2)+t(1,3)+t(1,9))$ // condition 5
$P_{1,7} = n(1,7)/(n(1,2)+n(1,3)+n(1,7))$
$= t(1,9)/(t(1,2)+t(1,3)+t(1,9))$ // condition 13
$P_{2,1} = n(2,1)/n(2,1) = 1$ // condition 12
$P_{3,1} = n(3,1)/(n(3,1)+n(3,4)+n(3,5))$
$= t(3,1)/(t(3,1)+t(3,5)+t(3,6))$ // condition 8

$$P_{3,4} = n(3,4)/(n(3,1)+n(3,4)+n(3,5)) = t(3,5)/(t(3,1)+t(3,5)+t(3,6)) \quad // \text{condition 2}$$
$$P_{3,5} = n(3,5)/(n(3,1)+n(3,4)+n(3,5)) = t(3,6)/(t(3,1)+t(3,5)+t(3,6)) \quad // \text{condition 1}$$
$$P_{4,3} = n(4,3)/n(4,3) = 1 \quad // \text{condition 11}$$
$$P_{5,3} = n(5,3)/(n(5,3)+n(5,6)) = t(6,3)/(t(6,3)+t(6,8)) \quad // \text{condition 9}$$
$$P_{5,6} = n(5,6)/(n(5,3)+n(5,6)) = t(6,8)/(t(6,3)+t(6,8)) \quad // \text{condition 4}$$
$$P_{6,5} = n(6,5)/n(6,5) = 1 \quad // \text{condition 11}$$
$$P_{7,8} = n(7,8)/(n(7,8)+n(7,9)) = t(9,10)/(t(9,10)+t(9,12)) \quad // \text{condition 14}$$
$$P_{7,9} = n(7,9)/(n(7,8)+n(7,9)) = t(9,12)/(t(9,10)+t(9,12)) \quad // \text{condition 14}$$
$$P_{8,9} = n(8,9)/(n(8,9)+n(8,10)) = t(10,12)/(t(10,12)+t(10,14)) \quad // \text{condition 16}$$
$$P_{8,10} = n(8,10)/(n(8,9)+n(8,10)) = t(10,14)/(t(10,12)+t(10,14)) \quad // \text{condition 15}$$
$$P_{9,8} = n(9,8)/(n(9,8)+n(9,11)) = t(12,10)/(t(12,10)+t(12,15)) \quad // \text{condition 18}$$
$$P_{9,11} = n(9,11)/(n(9,8)+n(9,11)) = t(12,15)/(t(12,10)+t(12,15)) \quad // \text{condition 17}$$
$$P_{10,7} = n(10,7)/(n(10,7)+n(10,12)) = t(14,9)/(t(14,9)+t(14,16)) \quad // \text{condition 13}$$
$$P_{10,12} = n(10,12)/((n(10,7)+n(10,12)) = t(14,16)/(t(14,9)+t(14,16)) \quad // \text{condition 13}$$
$$P_{11,12} = n(11,12)/n(11,12) = 1 \quad // \text{condition 13}$$
$$R_1 = r_1 \quad // \text{condition 13}$$
$$R_2 = r_2 \quad // \text{condition 12}$$
$$R_3 = r_3 r_4 \quad // \text{condition 8}$$
$$R_4 = r_5 \quad // \text{condition 11}$$
$$R_5 = 1-(1-r_6)(1-r_7) \quad // \text{condition 9}$$
$$R_6 = r_8 \quad // \text{condition 11}$$
$$R_7 = r_9 \quad // \text{condition 14}$$
$$R_8 = r_{10} r_{11} \quad // \text{conditions 15 and 16}$$
$$R_9 = 1-(1-r_{12})(1-r_{13}) \quad // \text{conditions 17 and 18}$$
$$R_{10} = r_{14} \quad // \text{condition 13}$$
$$R_{11} = r_{15} \quad // \text{condition 13}$$
$$R_{12} = r_{16} \quad // \text{condition 13}$$

After the transition matrix M is constructed, software reliability can be computed, based on Eq. (2) in Section 2.

6. A case study

We conducted a case study to investigate the validity of the assumptions and the analysis of our model on real systems. An empirical study was conducted on an industrial real time component-based system,[1] which has been used by more than 100 companies and 4000 individual users over the past several years. This system provides a set of statistical models to help traders and fund managers analyze the stock market's historical data and catch future movement.

The system is composed of four sub-units including data unit, business rule unit, utility unit, and presentation unit. These units serve as mathematical libraries and were implemented using C and C++. In this study, we focused on the data unit, which contains 54 classes, 13,846 lines of code, and 921 functions. It has a total of 15 components in three identified architectural styles, including batch-sequential, parallel, and client–server. The database components run concurrently with the evaluation components so that modeling and data retrieval can operate simultaneously as in parallel. Two components, calculator and matrix, serve as server components to provide complex mathematics calculations for the client components as in call-and-return. The other components are running in sequential manner with looping and branching conditions as in batch-sequential. A test pool of this system was provided by Quality Assurance (QA) team with a total of 13,596 test cases, and out of those test cases 121 failures were observed.

To apply our model, three attributes are required as discussed in Section 3. With the above system architecture, the component reliabilities and the transition probabilities among components also need to be available. To measure the reliabilities of the constituent components, we collected data recorded by the QA team during component testing and computed the number of successful executions without crash over the total number of executions for each individual component. The next step is to compute the transition probabilities among components. We computed the transition probabilities among components from the transition occurrences among components collected during the path traversals of those 13,596 test cases. The collection of transition occurrences was an automatic process through the instrumented code to an operational system copy.

Finally, we constructed the state machine for the system based on the methodologies described in Sections 3 and 4 and constructed a Markov model to compute reliability by Eq. (2). The computation yields 0.994001, with 99.4% of executions potentially being failure free. This reliability result was considered to be acceptable after consultation with the QA team members, who verified that failures were mainly caused by some infrequently executed functions and the chance of running them had been pretty low. In addition, we compared our model result with the Nelson model (Nelson, 1978), which had the result computed as the total number of passed test cases over the total number of test cases. The difference between the reliability measures of the two models is around 0.3%.

7. Related work

Software Reliability Growth Models (SRGMs) employ test history to predict *Mean Time To Failure* of the

[1] The system is proprietary and the owners requested that the system not be identified.

software (Farr, 1996; Musa et al., 1987). Using a black-box approach, these models require retesting the whole software if a change or an update occurs to the structure. Today, software components can easily plug-and-play and upgrade, which eases the software development but hinders the use of SRGMs due to repeated testing efforts.

A number of studies adopt Markov models to measure the reliability of modular software. Cheung (1980) proposed a user-oriented reliability model to measure the reliability of service that a system provides to a user community. A discrete Markov model was formulated based on the knowledge of individual module reliability and inter-module transition probabilities. In Littlewood's (1975, 1979) reliability model, a modular program is treated as transfers of control between modules following a semi-Markov process. Each module is failure-prone, and the different failure processes are assumed to be Poisson. Laprie et al. (1991) evaluate the reliability of multi-component systems by introducing a knowledge-action transformation (KAT) model, which accounts for reliability growth phenomena, and enables estimation and prediction of the reliability of multi-component systems. These models are limited to modeling sophisticated structures and heterogeneous architecture, but can handle homogeneous software with common sequential and branching transitions.

In addition to analytic models, experiments and simulations were also developed to predict and measure software reliability. Krishnamurthy and Mathur (1997) conducted an experiment to evaluate a method, *Component Based Reliability Estimation* (CBRE), to estimate software reliability using software components. CBRE involves computing path reliability estimates based on the sequence of components executed for each test input and the system reliability is the average over all test runs. Gokhale et al. (1998a,b,c) proposed a discrete-event simulation to capture a detailed system structure and to study the influence of separate factors in a combined fashion on dependability measures. Li et al. (1997) presented a methodology and accompanying toolset, W2S, for generating a simulator from a semi-formal architecture description, which allows an analysis of the system's reliability based on its simulated behavior and performance. Gokhale et al. (1998a,b,c) predicted architecture-based software reliability using a testing-based approach, which parameterized the analytic model of the software using measurements obtained from the regression test suite, and coverage measurements. The above approaches are able to model certain system structures, but are usually expensive and time-consuming especially when applied to structure changes, and the application domains can be limited.

8. Conclusions

We introduce an architecture-based approach for software reliability modeling, which addresses various software infrastructures and can compute reliability for different component configurations in different phases of a software process. Pragmatically, software with heterogeneous architecture that behaves non-uniformly across different portions of the system will be analyzed and then modeled by using a state machine. The state machine is constructed to become a stochastic matrix of a discrete-time Markov model, and software reliability is computed through matrix computations. This approach can facilitate quality assessment at the architecture level for deciding component reuse, thus benefiting decision-making on software architecture design. It can also support component changes at later phases without a complete system retest.

The architecture-based reliability modeling is a top–down approach, in which software is first examined based on the design requirements and classified into different architectural styles. Based on the study of reliability modeling for each individual architectural style, the style-based models are used as the building blocks for the architecture-based reliability model to measure overall system reliability. This architecture-based approach takes software architecture into account and the white-box feature only needs to retest the affected components or changed structures during software evolution. Therefore, it is useful in guiding the selections of architectural styles and components for reuse in new designs. For future component changes, the model can compute a new reliability measure by only examining the affected portions. An empirical study of this model on a real-time system suggests its great potential for use in modern software infrastructures.

The limitation of this model is the lack of consideration of execution history due to the Markov property, regarding the transitions from a component to others as being probabilistic. In many situations, they can be deterministic involving no randomness. For example, a component calls several components in sequence, and a call to the first guarantees calls to the others. To broaden the applicability of this model on different application domains, our future work intends to address deterministic behaviors and take history into account.

Appendix A

A Markov model is a finite state machine with probabilities for each transition, and a transition probability to the next state will depend on the current state only. For a discrete-time Markov model, the transitions occur only at discrete intervals of time or at discrete events, and the transition probabilities follow a discrete distribution. Accordingly, a discrete-time Markov model consists of

(1) a finite set of n states $S = \{s_1, \ldots, s_n\}$,
(2) an $n \times n$ stochastic matrix $T = (P_{ij})$, $P_{ij} \geqslant 0$ and $\sum_{j=1}^{n} P_{ij} = 1$, where P_{ij} is the transition probability that the system will move to state s_j, given only that the system is in state s_i, and
(3) a vector $\pi^0 = (\pi_1^0, \ldots, \pi_n^0)$ where π_i^0 denotes the probability that the system is initially in state s_i.

These Markov-based software reliability models associated with the defined stochastic matrix T are two essential lemmas, as shown below. They ensure that the denominator in the reliability formula is not equal to zero, thus yielding a measurable result. The detailed proofs of these two lemmas are available in (Berman and Plemmons, 1979).

Lemma 1. *If a non-negative stochastic matrix T is standardized, the spatial radius $\rho(M) < 1$.*

$$T = \begin{bmatrix} D_1 & 0 & \cdots & 0 & 0 \\ 0 & D_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & D_r & 0 \\ B_1 & B_2 & \cdots & B_r & M \end{bmatrix} \quad \textit{standardized}$$

where D_i is an $n_i \times n_i$ irreducible state transition matrix associated with its ergodic equivalence class; M is a $k \times k$ square matrix and all the states corresponding to M are transient.

Lemma 2. *If M is a $k \times k$ matrix with $\rho(M) < 1$, then $(I - M)^{-1}$ is non-singular and $(I - M)^{-1} = \lim_{j\to\infty}\sum_{i=0}^{j} M^i = \sum_{i=0}^{\infty} M^i$, where I is an identity matrix with the same size as M. When $(I - M)^{-1}$ is non-singular, the determinant of I minus M, denoted as $|I - M|$, is not equal to 0.*

Included below are the related definitions and theorems for these two lemmas. For further information, please reference the book "Non-negative Matrices in the Mathematical Sciences" (Berman and Plemmons, 1979).

Definition 1. A state s_i is called *transient* if $s_i \to s_j$ but not $s_j \to s_i$, that is, s_i has access to some s_j that has no access to s_i. Otherwise, the state s_i is called *ergodic*. Thus s_i is ergodic if and only if $s_i \to s_j$ implies $s_j \to s_i$.

Definition 2. The *classes* of a Markov chain are the equivalence classes induced by the communication relation on the set of states. A class α has access to a class β if $s_i \to s_j$ for some $s_i \in \alpha$ and $s_j \in \beta$. A class is called *final* if it has access to no other class. An *ergodic equivalence class* is a class that is final and contains all ergodic states.

Theorem 1. *Let $a_{ij}^{(q)}$ denote the (i,j) element of A^q. A non-negative matrix A is irreducible if and only if for every (i,j) there exists a natural number q such that $a_{ij}^{(q)} > 0$.*

To illustrate, the following depicts Cheung's user-oriented reliability model (Cheung, 1980). Cheung's model consists of a standardized $n \times n$ stochastic matrix T, where $n = k + 2$, as shown in (A.1). The inner matrix M in T is a $k \times k$ transition matrix with state s_1 as the entry state and s_k as the exit state, as shown in (A.2). Note that $M(i,j)$ is the entry of ith row and jth column, R_i is the reliability of software component c_i, and P_{ij} is the reaching probability from component c_i to c_j, where $1 \leqslant i,j \leqslant k$. Two ergodic states S and F represent a successful state and a failure state, respectively. Consequently, a successful execution of software will eventually reach the ergodic state S; otherwise, reach F instead. Since S and F are ergodic, the probabilities from S to S and from F to F are both equal to 1. In addition, there are two inner vectors B_1 and B_2. B_1 has only one entry not equal to 0 that is the probability of reaching state S from the exit state s_k. B_2 stores the failure probabilities that reach state F from states s_1 to s_k. Based on (A.1) and (A.2), this finite state machine is a discrete-time Markov model with each row sum in the standardized matrix T equal to 1.

$$T = \begin{array}{c} \\ S \\ F \\ s_1 \cdots s_k \end{array} \begin{array}{c} \begin{array}{ccc} S & F & s_1 \cdots s_k \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ B_1 & B_2 & M \end{bmatrix} \end{array}, B_1 = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ R_k \end{bmatrix}, \; B_2 = \begin{bmatrix} 1-R_1 \\ \vdots \\ 1-R_{k-1} \\ 1-R_k \end{bmatrix} \tag{A.1}$$

$$M(i,j) = \begin{cases} R_i P_{ij}, & \text{state } s_i \text{ reaches state } s_j \text{ and } i \neq k \\ 0, & \text{otherwise,} \end{cases}$$
$$\text{for } 1 \leqslant i,j \leqslant k \tag{A.2}$$

In Cheung's (1980) model, software reliability R is computed as

$$R = (-1)^{k+1} R_k \frac{|E|}{|I - M|} \tag{A.3}$$

$|I - M|$ is the determinant of matrix $(I - M)$, while $|E|$ is the determinant of the remaining matrix excluding the last row and the first column of $(I - M)$. $D_1 = [1]$ and $D_2 = [1]$, both belonging to an ergodic equivalence class, are irreducible. Since T is standardized, this implies that the determinant $|I - M|$ will not be equal to 0, so that software reliability R has a solution.

References

Agnes, M., 2000. Webster's New World College Dictionary, fourth ed. Macmillan, USA, p. 1399.

Allen, R., Garlan, D., 1994. Formalizing architectural connection. In: Proceedings of the Sixteenth International Conference on Software Engineering, Sorrento, Italy, 16–21 May, pp. 71–80.

Bass, L., Clements, P., Kazman, R., 1998. Software Architecture in Practice. Addison-Wesley Longman, Inc.

Berman, A., Plemmons, R.J., 1979. Nonnegative Matrices in the Mathematical Sciences. Academic Press Inc, New York.

Chen, M.H., Garg, P., Mathur, A.P., Rego, V.J., 1995. Investigating coverage-reliability relationship and sensitivity of reliability estimates to errors in the operational profile. Computer Science and Informatics Journal 25 (3), 165–170, Special Issue on Software Engineering.

Chen, M.H., Tang, M.H., Wang, W.L., 1998. Effect of software architecture configuration on the reliability and performance estimation. In: Proceedings of the 1998 IEEE Workshop on Application Specific Software Engineering and Technology, 26–28 March, Richardson, TX, pp. 90–95.

Chen, M.H., Lyu, M.R., Wong, E.W., 2001. Effect of code coverage on software reliability measurement. IEEE Transactions on Reliability 50 (2), 165–170.

Cheung, R.C., 1980. A user-oriented software reliability model. IEEE Transactions on Software Engineering 6 (2), 118–125.

Clements, P.C., 1996. Coming attractions in software architecture, Technical Report CMU/SEI96 TR008 and ESCTR96008, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, USA, January.

Farr, W., 1996. Software reliability modeling survey. In: Lyu, M.R. (Ed.), Handbook of Software Reliability Engineering. McGraw-Hill Publishing Company, IEEE Computer Society Press, New York, pp. 71–117.

Garlan, D., 1995. What is Style? In: Proceedings of Dagshtul Workshop on Software Architecture, Saarbruecken, Germany, February.

Garlan, D., Shaw, M., 1993. An Introduction to Software Architecture. In: Ambriola, V., Tortora, G. (Eds.), Advances in Software Engineering and Knowledge Engineering, 1. World Scientific Publishing Company.

Garlan, D., Allen, R., Ockerbloom, J., 1994. Exploiting style in architectural design environments. In: Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, 19(5), New Orleans, Louisiana, December.

Goel, A.L., Okumoto K., 1979. A Markovian model for reliability and other performance measures of software systems. In: Proceedings of National Computer Conference, pp. 769–775.

Gokhale, S.S., Lyu, M.R., Trivedi, K.S., 1998a. Reliability simulation of component-based software systems. In: Proceedings of Ninth International Symposium on Software Reliability Engineering (ISSRE), Paderborn, Germany, November, pp. 192–201.

Gokhale, S.S., Lyu, M.R., Trivedi, K.S., 1998b. Software reliability analysis incorporating fault detection and debugging activities. In: Proceedings of Ninth International Symposium on Software Reliability Engineering (ISSRE), Paderborn, Germany, 4–7 November, pp. 202–211.

Gokhale, S.S., Wong, W.E., Trivedi, K.S., Horgan, J.R., 1998c. An analytical approach to architecture-based software reliability prediction. In: Proceedings of IEEE International Computer Performance and Dependability Symposium (IPDS), Durham, North Carolina, September, pp. 13–22.

Hamlet, D., Mason, D., Woit, D., 2001. Theory of software reliability based on components. In: Proceedings of 23rd International Conference on Software Engineering (ICSE), Toronto, Ont., Canada, 12–19 May, pp. 361–370.

Johnson Jr., A.M., Malek, M., 1988. Survey of software tools for evaluating reliability, availability, and serviceability. ACM Computing Surveys 20 (4), 227–269.

Krishnamurthy, S., Mathur, A.P., 1997. On the estimation of reliability of a software system using reliabilities of its components. In: Proceedings of Eighth International Symposium on Software Reliability Engineering (ISSRE), Albuquerque, NM. USA, November, pp. 146–155.

Laprie, J.-C., 1995. Dependability of computer systems: concepts, limits, improvements. In: Proceedings of Six International Symposium on Software Reliability Engineering (ISSRE), Toulouse, France, 24–27 October, pp. 2–11.

Laprie, J.-C., Kanoun, K., 1996. Software reliability and system reliability. Handbook of Software Reliability Engineering. McGraw-Hill, New York, pp. 27–70.

Laprie, J-C., Kanoun, K., Béounes, C., Kaâniche, M., 1991. The KAT approach to the modeling and evaluation of reliability and availability growth. IEEE Transactions on Software Engineering SE-17 (4), 370–382.

Li, J.J., Micallef, J., Horgan, J.R., 1997. Automatic simulation to predict software architecture reliability. In: Proceedings of Eighth International Symposium on Software Reliability Engineering (ISSRE), Albuquerque, NM, USA, November, pp. 168–179.

Littlewood, B., 1975. A reliability model for systems with Markov structure. Applied Statistics 24 (2), 172–177.

Littlewood, B., 1979. Software reliability model for modular program structure. IEEE Transactions on Reliability 28 (3), 241–246.

Lopez-Benitez, N., 1994. Dependability modeling and analysis of distributed programs. IEEE Transactions on Software Engineering 28 (5), 345–352.

Medvidovic N., Oreizy P., Taylor R.N., 1997. Reuse of off-the-shelf components in C2 style architectures. In: Proceedings of 19th International Conference on Software Engineering, 17–23 May, pp. 692–700.

Musa, J., Iannino, A., Okumoto, K., 1987. Software Reliability: Measurement, Prediction, Application. McGraw-Hill, New York.

Nelson, E., 1978. Estimating software reliability from test data. Microelectronics and Reliability 17 (1), 67–73.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Eng. Notes 17 (4), 40–52.

Pinkerton, T.B., 1968. Program behavior and control in virtual storage computer systems. Technical Report 4, University of Michigan.

Ramamoorthy, C.V., 1966. The analytic design of a dynamic look ahead and program segmenting system for multiprogrammed computers. Proceedings of ACM National Conference, 229–239.

Shaw M., 1991. Heterogeneous design idioms for software architecture. In: Proceedings of Sixth International Workshop on Software Specification and Design, Como, Italy, October, pp. 158–165.

Shaw, M., Garlan, D., 1994. Characteristics of higher-level languages for software architecture. Technical Report CMU-CS-94-210, Carnegie Mellon University, Pittsburgh, PA, USA, December.

Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

Shaw, M., DeLine, R., Klein, D., Ross, T., Young, D., Zelesnik, G., 1995. Abstractions for software architecture and tools to support them. IEEE Transactions on software Engineering 21 (4), 314–335.

Taylor, R., Medvidovic, N., Anderson, K.M., Whitehead Jr., E.J., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L., 1996. A component- and message-based architectural style for GUI software. IEEE Transactions on Software Engineering 22 (6), 390–406.

Tracz, W., 1995. DSSA frequently asked questions (FAQ). ACM Software Engineering Notes 19 (2), 52–56.

Wang, W.L., Tang, M.H., Chen, M.H., 1999. Software architecture analysis—a case study. In: Proceedings of 23rd IEEE International Computer Software and Applications Conference, Phoenix, Arizona, 25–26 October, pp. 265–270.

Wang, W.L., Wu, Y., Chen, M.H., 1999. An architecture-based software reliability model. In: Proceedings of Pacific Rim International Symposium on Dependable Computing, Hong Kong, pp. 143–150.

Wen-Li Wang is currently an Assistant Professor in the program of Electrical, Computer and Software Engineering at Penn State Erie, the Behrend College. He earned his Ph.D. and M.S. degrees in Computer Science from State University of New York at Albany and received his B.S. degree in Management Information Systems at National Chengchi University in Taiwan. His research interest is in the areas of Object-Oriented Design and Test, Software Reliability Modeling, Design Patterns, Software Architecture, Signal Processing and Swarm Intelligence.

Dai Pan is a Ph.D. student in the Computer Science Department at the University at Albany, SUNY. He received the M.S. degrees in Physics and in Computer Science from the University at Albany, SUNY. He is currently working at Bear, Stearns & Co., Inc. in the Derivatives Quant Group. His research interests include software architecture analysis, component-based software modeling, testing, and maintenance.

Mei-Hwa Chen received the Ph.D. degree in Computer Science and the M.S. degree in Mathematics from Purdue University. She is currently an associate Professor in the Computer Science Department at the University at Albany, SUNY. Her research interests include software architecture analysis, software testing, change impact analysis, maintenance, and software reliability modeling.