TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 31/32 pp908–925
DOI: 10.26599/TST.2024.9010101
Volume 30, Number 2, April 2025

Software Reliability Assessment: An Architectural and Component Impact Analysis

Saleh Alyahyan*, Mohammed Naif Alatawi, Mrim M. Alnfiai, Shoayee Dlaim Alotaibi, Abdullah Alshammari, Zaid Alzaid, and Hathal Salamah Alwageed

Abstract: In the software landscape, understanding component impacts on system reliability is pivotal, especially given the unique complexities of modern software systems. This paper presents a model tailored for software reliability assessment. Our approach introduces the "component influence" to measure a single component's effect on overall system reliability. Additionally, we adapt a state transition model to cater to the diverse architectures of software systems. Using a discrete-time Markov chain, we predict software reliability. We test our model on an actual software system, finding it notably accurate and superior to existing methods. Our work offers a promising direction for those venturing into software reliability enhancement.

Key words: software reliability; architectural analysis; state transition model

1 Introduction

Understanding software reliability is more than just imperative. It pertains to the probability that softwares perform as expected in a computing environment for a specified period[1–3]. As computing ushers us into a new era of computational power and potential, ensuring the reliability of software right from design, through research, to operation becomes paramount. They are especially considering the critical roles of the systems anticipated to play in advanced aeronautics, deep-space exploration, next-generation automotive systems, ultra-precise industrial automation, and the backbone of future critical infrastructures.

The focal point of our investigation is on the early-phase reliability of software, designed using an ensemble of components. There is a plethora of methodologies for predicting the reliability of software. Some revolve around "black-box" models relying on test data[4–14], while others, termed "white-box" models, delve deep into a software's intrinsic architecture[15–19]. However, there is an observed void: these methodologies might not holistically consider the importance of each gate or bit[20–33]. Given the principles inherent in computing, the failure of one bit can have non-trivial and cascading impacts on a system. A limited number of works[16, 17, 22, 28, 32, 33] have broached this topic, but the intricacies demand deeper exploration.

• Saleh Alyahyan is with Applied College in Dwadmi, Shaqra Univerity, Shaqra 11961, Saudi Arabia. E-mail: salyahyan@su.edu.sa.
• Mohammed Naif Alatawi is with Department of Information Technology, University of Tabuk, Tabuk 47713, Saudi Arabia. E-mail: Alatawimn@ut.edu.sa.
• Mrim M. Alnfiai is with Department of Information Technology, Taif University, Taif 21944, Saudi Arabia. E-mail: m.alnofiee@tu.edu.sa.
• Shoayee Dlaim Alotaibi is with Department of Artificial Intelligence and Data Science, University of Hail, Hail 55431, Saudi Arabia. E-mail: s.alotaibi@uoh.edu.sa.
• Abdullah Alshammari is with College of Computer Science and Engineering, University of Hafr Albatin, Hafar Albatin 31991, Saudi Arabia. E-mail: dr.abdullah@uhb.edu.sa.
• Zaid Alzaid is with Department of Computer Science, Islamic University of Madinah, Madinah 42351, Saudi Arabia. E-mail: zsalzaid@iu.edu.sa.
• Hathal Salamah Alwageed is with the College of Computer and Information Sciences, Jouf University, Sakaka 42421, Saudi Arabia. E-mail: hswageed@ju.edu.sa.
* To whom correspondence should be addressed.
Manuscript received: 2024-04-07; revised: 2024-05-16; accepted: 2024-05-31

Charting our path in this research, we are keen to offer a groundbreaking model tailored for the software domain. This model not only gauges the reliability of software, but intricately intertwines the significance of each component, and the diversified architectures software can manifest[7, 8]. We introduce a metric that accurately reflects the impact of individual components on the overall system's reliability. In tandem, we have conceived a state transition model that aligns seamlessly with the variegated designs of software. To extrapolate our reliability findings, we lean on the discrete-time Markov chain, adapted for software systems.

Understanding software reliability is crucial. It gauges the chance that a software system will function without glitches within a certain environment for a predetermined span[21]. Recognizing its pivotal role, software reliability should be a cornerstone consideration, right from its design inception, through research phases, and into its operational lifespan. This is especially pertinent as we witness a sweeping integration of software in fields spanning aeronautics, astronautics, the automotive industry, industrial automation, and pivotal infrastructures[9–11].

The epicenter of our exploration revolves around gauging the reliability of software that is intricately constructed using an array of components, predominantly in the nascent stages of software development. Delving into the methods of predicting the steadfastness of such software reveals an interesting landscape. Certain models are inclined towards harnessing data from tests, commonly labeled as "black-box" models[12]. In contrast, others are more predisposed to meticulously dissecting the internal skeletal structure of the software, heralded as "white-box" models.

However, a discernible lacuna emerges: a majority of these assessment models rather egregiously sidestep the nuanced role each software component plays[13]. It is indisputable that distinct components are entrusted with disparate roles. A malfunction in one segment could have a cascading effect of varying magnitudes on the entire system. A handful of studies[16, 17] have brushed upon this facet, yet the integration of this understanding into the reliability assessment paradigm remains conspicuously absent.

In our foray into this domain, we intend to bridge this void. We are unfolding a model that does not merely estimate software reliability in isolation but integrates the inherent weight each component holds and the multifaceted designs software can assume. Our methodology introduces a calibrated metric to quantify the potential reverberations of individual components on holistic system reliability. Supplementing this, we have architected a transition model that melds gracefully with diverse software design blueprints. To crystallize our projections on reliability, the discrete-time Markov chain serves as our analytical compass.

The rest of the paper is structured as follows: Section 2 reviews related academic endeavors, Section 3 presents our pioneering model, Section 4 demonstrates our methodology's real-world applications, Section 5 reports the implications of this study finddings, and Section 6 concludes with insights and prospects for future research.

2 Related Work

We now summarize the related work studies. For instance, Singh and Tomar[1] proposed a reliability estimation model for Component-Based Software (CBS), which calculates overall system reliability by integrating the reliabilities of individual components and the architecture. The model uses path propagation probability and component impact factor to estimate reliability, considering the activation of components during execution paths. The path propagation probability accounts for the likelihood of specific execution paths, while the component impact factor evaluates the influence of individual components on overall reliability. The model is implemented in JAVA and validated through an adapted case study, demonstrating its effectiveness in estimating CBS reliability and its applicability in early stages of software development. In other study, Yacoub et al.[2] introduced a Scenario-Based Reliability Analysis (SBRA) technique for component-based software, leveraging scenarios of component interactions to construct a probabilistic model called the Component-Dependency Graph (CDG). This model and its accompanying algorithm analyze system reliability based on the reliabilities of its architectural components. The approach is extended to accommodate distributed software systems by incorporating link and delivery channel reliabilities. Key benefits include assessing the impact of variations and uncertainties in component and link reliabilities,

identifying critical components and interfaces, and facilitating early reliability analysis at the architecture level. This early detection is crucial for resource allocation in later development phases, particularly when using off-the-shelf components.

Palviainen et al.[3] presented a comprehensive approach to evaluating the reliability of component-based software systems during design and implementation phases. Their methodology integrates predicted and measured reliability values with heuristic estimates to streamline the reliability evaluation process. The approach combines component-level activities—heuristic reliability estimation, model-based reliability prediction, and model-based reliability measurement—with system-level reliability prediction to support the iterative and incremental development of reliable software. The effectiveness of this approach is demonstrated through a case study, and the paper concludes by summarizing the lessons learned from these case studies, highlighting the practical benefits of the proposed reliability evaluation method and supporting toolchain. Similarly, Gokhale and Trivedi[4] developed an accurate hierarchical model to predict the performance and reliability of component-based software systems based on their architecture. This model improves on previous composite models by accounting for the variance in the number of visits to each module, thus providing more accurate predictions. The approach helps identify performance and reliability bottlenecks and includes expressions for analyzing the sensitivity of these predictions to changes in individual module parameters. Additionally, the hierarchical model is used to assess the impact of workload changes on the system's performance and reliability, with the techniques demonstrated through practical examples.

In summary, this work differs from the existing studies by introducing a novel metric, "component influence", which specifically measures the impact of a single component on the overall system reliability. Unlike Singh and Tomar's model[1], which focuses on path propagation probability and component impact factors in a CBS context, our approach integrates a state transition model to accommodate diverse software architectures. While Yacoub et al.[2] used a CDG for scenario-based reliability analysis, our model leverages a discrete-time Markov chain to predict software reliability. Additionally, unlike Palviainen et al.[3], who combined heuristic estimates with predicted and measured reliability values, our method provides a more precise reliability prediction validated on an actual software system, showing superior accuracy. Furthermore, Gokhale and Trivedi's hierarchical model[4] accounts for variance in module visits, whereas our model uniquely adapts state transitions for comprehensive reliability assessment.

3 Proposed Reliability Assessment Model

In this section, we introduce a software reliability assessment model, building upon the interplay of software component impact and architecture analysis. Initially, we leverage a directed graph to represent the software system that incorporates multiple components. Following this, a novel metric, software component influence, is introduced to elucidate the varied impacts these components have on overall system reliability. We then present the global state transition model, facilitating the efficient transition from components across diverse software architectural styles to system states[34, 35]. Using this foundation, we formulate the global state model for software system based on these components, and subsequently compute the reliability of each state, amalgamating it with component influence. The discrete-time Markov chain is subsequently applied to appraise the reliability of the software[36, 37].

3.1 Component impact computation based on directed graph of software

Software system is constructed using numerous components, each termed as C. A component in this context is a reusable software element, designed to execute a specific function within a software environment, endowed with robust encapsulation and considered as a black box in this study. Embracing the principles of software engineering, the application of graph theory greatly enhances this framework[38]. As a starting point, we sculpt the directed graph representation of component-based software. Figure 1 illustrates this directed graph, comprised of six components. Each component is denoted as node C_i. The inter-relationships between components are captured as a set E of edges, allowing the software to be abstractly represented as a directed graph model $G = \langle C, E \rangle$. To elucidate this model further, we present the subsequent definitions:

Definition 1 Each node C_i in the directed graph G represents an independent component, and its reliability can be regarded as r_i.

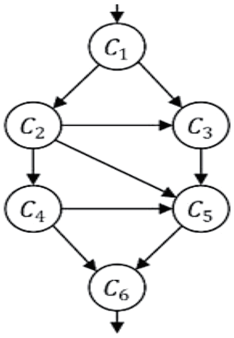

Fig. 1 Directed graph model of component-based software.

Definition 2 The directed edge e_{ij} indicates that the information can flow from component C_i to C_j, which is denoted as $C_i \rightarrow C_j$. In this study, components C_i and C_j are called a requester and provider, respectively. The transition from component C_i to C_j indicates that requester C_i calls to provider C_j, and the transition probability of $C_i \rightarrow C_j$ is $P_{i,j}$, where $\sum_{j=1}^{m} P_{i,j} = 1$, m is the total number of software system components.

Definition 3 Requester set A indicates a component set of all requesters' call to the component C_i, which can be expressed as $A = \left\{C_j \middle\lfloor \exists\, C_j \rightarrow C_i,\ j = 1, 2, \ldots, m\right\}$. Provider set B represents a component set of all providers called by component C_i, which can be expressed as $B = \{C_k \lfloor \exists\, C_i \rightarrow C_k,\ k = 1, 2, \ldots, m\}$.

Definition 4 The number of all directed edges starting from component C_i is called the out-degree of C_i, which can be regarded as $d_{\text{out}}(i)$. The number of all directed edges ending at component C_i is called the in-degree of C_i, which can be regarded as $d_{\text{in}}(i)$.

3.2 Component impact computation model

The positioning and function of each component within a software structure play pivotal roles in influencing the software's reliability. As such, each component's potential to affect the entire system's reliability varies based on its role and function. This relationship is particularly emphasized when we delve deeper into the intricacies of component-based software as it scales in size and complexity.

For a clearer perspective, consider Fig. 1, which delineates a component-based software ecosystem containing six distinct components. Among them, C_1 acts as the input, while C_6 functions as the output. This illustration paves the way for the following observations:

Observation 1 Should the input component C_1 or the output component C_6 falter, the software system may directly experience a malfunction. Thus, the pivotal nature of C_1 and C_6 bestows upon them a more substantial influence on system reliability than other auxiliary components.

Observation 2 Component C_5 caters to the needs of three distinct Components: C_2, C_3, and C_4. On the other hand, C_4 relies solely on C_2. This pattern suggests that C_5 malfunction could ripple more extensively through the system compared to C_4 failure. Consequently, C_5 influence quotient would surpass that of C_4.

Observation 3 Component C_2 interacts with a trio of Components C_3, C_4, and C_5, whereas C_3 interaction remains limited to C_5. Given the extensive pathways for fault propagation stemming from C_2, its potential system impact outstrips that of C_3.

In essence, the influence exerted by different components is heterogenous, and understanding these differences is paramount for a nuanced reliability analysis of CBS. Two critical aspects highlighted in Observations 2 and 3 are the intrinsic failures of components and the fault proliferation pathways they initiate. Achieving optimal system reliability mandates both a reduction in component-specific failures and curbing the adverse effects emanating from inter-component fault propagation. Additionally, each component's frequency of access varies, which in turn influences its reliability. Drawing inspiration from methods like PageRank[1], we dissect component importance through three pivotal lenses: failure impact, fault proliferation impact, and intrinsic significance,

$$\lambda_i = \alpha_1 \text{In}_F(i) + \alpha_2 \text{In}_P(i) + (1-\varphi)\text{In}_S(i) \qquad (1)$$

where parameter λ_i embodies the component impact associated with C_i. The indicators $\text{In}_F(i)$, $\text{In}_P(i)$, and $\text{In}_S(i)$ correspondingly delineate the failure impact, fault propagation impact, and inherent influence of C_i. To strike a balance among these impacts, we deploy weight coefficients α_1 and α_2. Furthermore, φ encapsulates the cumulative value of α_1 and α_2, oscillating within the interval [0, 1].

In contrast to the conventional PageRank algorithm, our proposed methodology places emphasis on ascertaining component significance grounded in three foundational pillars: inherent influence, failure impact, and fault propagation implications. This approach especially hones in on components that exhibit the following characteristics:

(1) Components routinely called upon by other

components.

(2) Components that frequently initiate calls to other constituents.

(3) Components characterized by extensive execution durations.

Typically, each component in a software system has varying visitation frequencies. This variability underpins the intrinsic significance of component C_i. Specifically, C_i inherent influence is gauged by its visitation count over a singular, exhaustive operational cycle of the software system, as depicted in the following:

$$\mathrm{In}_S(i) = E(\mu_i) \tag{2}$$

where $E(\mu_i)$ signifies the anticipated count of visits to the component C_i. Gleaning this value is feasible through an in-depth analysis of the system's operational profile in tandem with the transition likelihoods among components, as documented in Ref. [5].

As highlighted in the second phenomenon, a component C_i that consistently garners calls from a diverse range of components intrinsically wields a pronounced influence on the overarching system reliability. This study introduces the term "failure influence" to characterize this particular impact of component C_i. If one visualizes this through the lens of the directed graph model, it becomes evident that myriad edges originate from a multitude of nodes, all converging at node C_i. Consequently, within the confines of a CBS framework, the failure influence borne by a component C_i can be articulated as

$$\mathrm{In}_F(i) = \sum_{C_j \in A}^{n_1} \frac{E(\mu_j)}{d_{\mathrm{out}}(j)} \tag{3}$$

where n_1 typifies the count of members within the requester set A pertaining to component C_i. Additionall $E(\mu_j)$ denotes the execution frequency of the component C_j, while $d_{\mathrm{out}}(j)$ captures the out-degree for the same component C_j.

Under the observations made in the third phenomenon, if a component C_i routinely initiates calls to a vast array of other components, it inherently has an expansive array of fault propagation pathways coupled with an extensive fault propagation spectrum. Such a component, due to its intricate connectivity, invariably exerts a more profound impact on the entire system's reliability. This study introduces the term "propagation influence" to describe this unique attribute of component C_i. When viewed through the directed graph model's prism, it is evident that a multitude of edges emanate from the node C_i, each terminating at different nodes. Thus, within the framework of a CBS structure, the propagation influence wielded by a component C_i can be articulated as

$$\mathrm{In}_P(i) = \sum_{C_k \in B}^{n_2} \frac{E(\mu_k)}{d_{\mathrm{in}}(k)} \tag{4}$$

where n_2 denotes the count of members within the provider set B associated with component C_i. $E(\mu_k)$ illustrates the execution frequency of component C_k, whereas $d_{\mathrm{in}}(k)$ highlights the in-degree of the same component C_k.

Referring to the first observation, it is evident that the software system initially receives external data via its input components. Subsequently, the processed results are relayed to the external environment through the output components. This dynamic underscores the pivotal role of both input and output components in a CBS paradigm. They essentially serve as the gateway and exit points of the system. Yet, a directed graph does not inherently differentiate between input and output nodes. Distinctions among nodes in such a graph are typically based solely on their edge connections.

In the context of a software system characterized by a single point of entry and exit, all data pathways initiate from a distinct starting component and culminate at a singular terminating component. As a result, the execution frequency invariably remains at unity, emphasizing that a malfunction in either the input or output component translates to a systemic failure.

Conversely, in a software structure with multiple entry and exit points, the number of such components exceeds one. This implies that the execution frequency is less than unity. Thus, a failure in just one of these components does not necessarily precipitate a system-wide breakdown. Notably, the significance of self-influence surpasses the other two parameters, underscoring its paramountcy. Therefore, determining the influence of input/output components is primarily anchored on computing $\mathrm{In}_S(i)$.

3.3 Global state transition model based on architecture analysis

Considering the software architecture style is the high abstraction of the software structure and can accurately

describe the work mechanism, topology, and constraints between components, the architecture style is chosen to characterize the composition of the components, which will help to reflect the complex interactions between components better. Meanwhile, the diversity of software architectural styles is always observed in actual systems, especially in large and complex ones[39, 40]. Hence, a state mapping mechanism, which provides the flexible transition from components to system states to solve the diversity problem of software architectural styles, is established by analysing the characteristics of component interactions under five basic software architectural styles. To simplify the description, each component in the software system is independent of other components in this research.

3.4 Reliabilities of architectural styles

(1) Sequence batch

Sequence batch is a common and simple architecture in CBS. Figure 2a shows an example of this architecture, where $C_1, C_2, \ldots, C_5$ are software components. In a sequence batch architecture, components are executed sequentially, once a particular component concludes its execution, the next in the sequence is initiated. This progression from individual components to comprehensive system states can be visualized as a one-to-one mapping of each component to its corresponding state. This relationship is depicted in Fig. 2b, where $S_1, S_2, \ldots, S_5$ are system states. The reliability of the system state in sequence batch architecture can be expressed as

$$R_i = r_i^{\lambda_i} \tag{5}$$

where R_i is the reliability of state S_i considering the component influence. Equation (5) elucidates that when the influence of component C_i on software reliability, denoted by $r_i^{\lambda_i}$, is low, its effect tends towards 1. Conversely, when the influence is closer to 0, the impact of C_i on the integral functionality of the CBS system becomes negligible. Let us consider that $P_{i,j}$ symbolizes the likelihood of transitioning from system state S_i to system state S_j, the reliability of the sequence batch architecture is calculated as follows:

$$R_{\text{seq-bra}} = R_1 (P_{1,2} R_2 P_{2,3} + P_{1,4} R_4 P_{4,5} R_5 P_{5,3}) R_3.$$

(2) Parallel batch

Figure 3a shows an example of a parallel architecture. In a parallel architecture, multiple

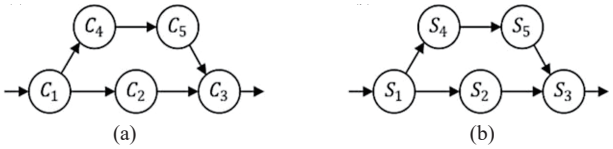

Fig. 2 Sequence batch. (a) Architecture and (b) state view.

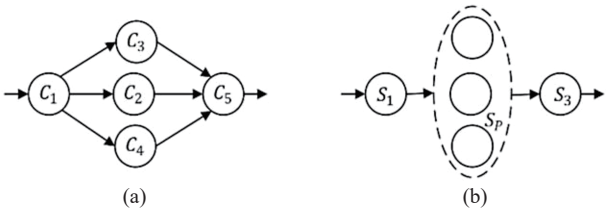

Fig. 3 Parallel batch. (a) Architecture and (b) state view.

components work in parallel to accomplish the identical task in the same time. Therefore, the transition from components to system states can be viewed as a mapping of multiple components in parallel to one state, as shown in Fig. 3b, where S_1, S_P, and S_5 are system states. The reliability of parallel state S_P can be expressed as

$$R_P = \prod_{i=2}^{4} r_i^{\lambda_i} \tag{6}$$

which indicates that during concurrent operations, if any component encounters a failure, the entire parallel state will fail. This is the characteristic of parallel systems where every component must operate correctly for the entire system to function. Given this behavior, the reliability of a system employing a parallel architecture can be derived as

$$R_{\text{par}} = R_1 R_P R_5.$$

(3) Fault tolerance

Fault-tolerant architectures are commonly found in scenarios demanding high reliability and real-time performance[25]. By using functionally identical components, the reliability of CBS can be enhanced, making it more resilient to component failures[33]. Figure 4 illustrates three prominent fault-tolerance methodologies: recovery block, N-version programming, and parallel. Each of these methods is further detailed below, accompanied by formulas to help analyze their respective reliability contributions to the overarching system.

(a) Recovery block: As depicted in Fig. 4a, a recovery block organizes redundant components in a specific manner. If the main component experiences a failure, the backup components are called into action

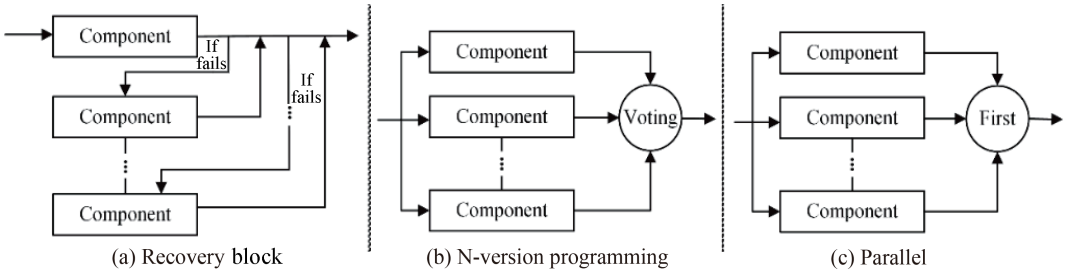

Fig. 4 Fault tolerance strategies.

sequentially[33]. A recovery block is only deemed to have failed if every single redundant component within it breaks down.

(b) N-version programming: As illustrated in Fig. 4b, N-version programming is a technique for software fault-tolerance in which several functionally equivalent components operate concurrently and independently. The ultimate outcome is derived through a majority vote[33]. This architecture is generally believed to have no failures when at least M components are executed successfully, where M is often $[n/2]+1$ based on the experience, Where n is typifies the count of members within the requester set.

(c) Parallel: As depicted in Fig. 4c, the parallel strategy activates all the n functionally equivalent components simultaneously, and the initial response received is used as the final result[33]. An parallel fault-tolerance architecture fails when the set of fault tolerance components fail.

Considering different fault-tolerance strategies have different features, the fault-tolerance architectures are further divided in two categories based on the reliability calculating formulas.

The first one is called the interrupt fault tolerance architecture, which means that the fault tolerance state fails only if all the fault tolerance components fail, such as recovery block and parallel. Figure 5a shows an example of this architecture. In an interrupt fault tolerance architecture, the backup component will timely replace the main component to ensure a successful system operation when the main component fails. Therefore, the transition from components to system states can be viewed as a mapping of multiple components comprising the main and all backup components to one state, as shown in Fig. 5b, where S_1, S_T, and S_5 are system states. The reliability of interrupt fault tolerance state S_T can be expressed as

$$R_T = 1 - \prod_{i=2}^{4} \left(1 - r_i^{\lambda_i}\right) \tag{7}$$

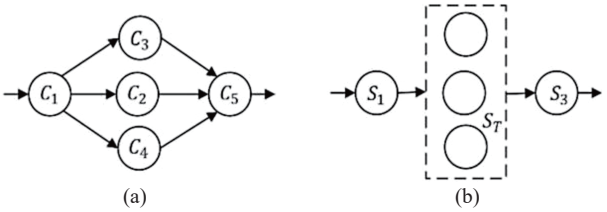

Fig. 5 Interrupt fault tolerance. (a) Architecture and (b) state view.

where R_T is the reliability of interrupt fault tolerance state S_T considering the component influence. On the basis of Eq. (7), the reliability of the interrupt fault tolerance architecture is calculated as follows:

$$R_{\text{fau-tol}} = R_1 R_T R_5.$$

The second one is called the non-interrupt fault tolerance architecture, which means that the fault tolerance state fails when less than M components are executed successfully, like N-version programming. Figure 6a shows an example of this architecture. The uninterrupted fault-tolerance architecture combines the features of both parallelism and redundancy. In this setup, n identical components operate concurrently to accomplish a shared task, with successful components serving as backups for those that falter. Consequently, transitioning from components to system states can be visualized as mapping all identical components to a singular state, as illustrated in Fig. 6b, where S_1, S_N, and S_5 are system states. The reliability of S_N considering the component influence can be regarded as the sum of all probabilities successfully executed by

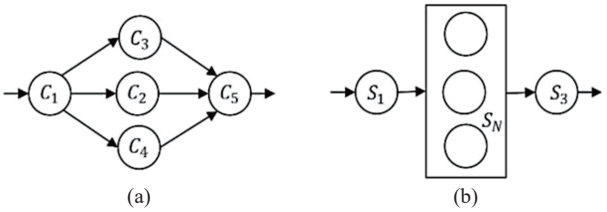

Fig. 6 Non-interrupt fault tolerance. (a) Architecture and (b) state view.

the software system,

$$R_N = \sum_{k=M}^{3} r_i^{k\lambda_i}\left(1 - r_i^{\lambda_i}\right)^{3-k} \tag{8}$$

where R_N is the reliability of non-interrupt fault tolerance state S_N considering the component influence. On the basis of Eq. (8), the reliability of the non-interrupt fault tolerance architecture is calculated as follows:

$$R_{\text{non-int}} = R_1 R_N R_5.$$

(4) Cycle

Figure 7a depicts a cyclic architectural model with C_1, C_2, and C_3 as software components. Within this structure, a given component is executed repetitively to complete a designated task. The shift from components to system states is represented as a one-to-one mapping from a component to a state, as demonstrated in Fig. 7b, where S_1, S_2, and S_3 symbolize system states. The system state's reliability can be determined using Eq. (5). The recurring component in this layout is referred to as a "cycler" in our discussion. As an illustration, C_2 serves as a cycler in Fig. 7a. If $P_{i,j}$ symbolizes the transition probability from system state S_i to S_j, then a cyclic architecture can be perceived as a unique sequential batch structure, and its reliability is deduced as follows:

$$R_{\text{cyc}} = \frac{R_1 P_{1,2} R_2 P_{2,3} R_3}{1 - P_{2,2} R_2}.$$

(5) Call and return

Figure 8a shows an example of the call and return architecture. In this architecture, In this structure, the present component relies on invoking the functionalities of other components to accomplish a designated task. Once the results are received from the other components, the current component resumes its operation. This process of transitioning from components to system states is represented as a one-to-one correspondence between a component and a state, as illustrated in Fig. 8b. Here, S_1, S_2, and S_3 stand for system states. The reliability of each system state can be deduced by employing the formula given in Eq. (5). Components in this architecture are divided into two types. The component that calls the functions of other components is named as caller, and the called component that provides the services to callers are named as responder in this paper. Responders are invoked by callers only. For example, a caller C_1 calls to a responder C_2 in Fig. 8a. Suppose that $P_{i,j}$ represents the probability of the transition from system state S_i to system state S_j, a call and return architecture is regarded as a special sequence batch architecture, and its reliability can be calculated as follows:

$$R_{\text{cal-ret}} = \frac{R_1 P_{1,3} R_3}{1 - P_{1,2} R_2}.$$

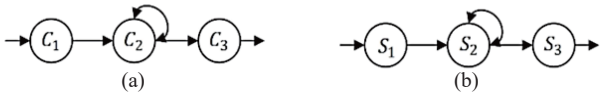

Fig. 7 Cycle. (a) Architecture and (b) state view.

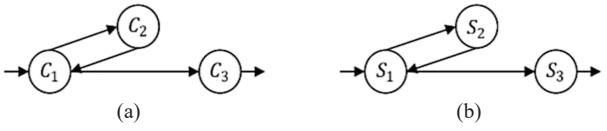

Fig. 8 Call and return. (a) Architecture and (b) state view.

3.5 State mapping mechanism based on architecture analysis

Upon analyzing the interaction patterns of components across five distinct software architectural frameworks—including sequence batch, parallel, fault tolerance, cycle, and call and return—we have formulated a state mapping mechanism. This involves computing the reliability of each state, taking into account the specific influence of the components. Using this foundation, we have devised a holistic state modeling approach to amalgamate the overarching state model for component-driven software. For clarity on the construction process of this encompassing state model, we outline a series of notations in Table 1.

Step 1: For a given directed graph model of the component-based software, identify and classify the architectures by analysing the characteristics of components interactions.

Step 2: Transform the components in the identified architectural styles to system states in accordance with the state mapping mechanism.

Step 3: Repeat the previous transitions until all system states satisfy the Markov properties and independent failure condition. Hence, the final result of conversions is that each transition from state S_i to state S_j in the software meets the basic transition types.

In this study, the transition from state S_i to state S_j is regarded as $S_i \to S_j$, and all state transitions are classified into seven basic types to simplify the calculation process.

Table 1 Notations of different components and state sets.

Parameter	Definition
C	Aggregate of all components within the software framework G
B	Aggregate of all components falling under the sequence batch architectural styles
P	Aggregate of all components falling under the parallel architectural styles
T	Aggregate of all components in the interrupt fault tolerance architectural styles
N	Aggregate of all components in the non-interrupt fault tolerance architectural styles
Q	Aggregate of all initiating components in the call and return architectural styles
S	Aggregate of all responding components in the call and return architectural styles
D	Aggregate of all repeating components in the cycle architectural styles
δ	Aggregate of system states derived from components in the software framework G
$\delta(B)$	Aggregate of system states derived from components under sequence batch architectural styles
$\delta(P)$	Aggregate of system states derived from components under parallel architectural styles
$\delta(T)$	Aggregate of system states derived from components in interrupt fault tolerance architectural styles
$\delta(N)$	Aggregate of system states derived from components in non-interrupt fault tolerance architectural styles
$\delta(Q)$	Aggregate of system states derived from initiating components in the call and return architectural styles
$\delta(S)$	Aggregate of system states derived from responding components in the call and return architectural styles
$\delta(D)$	Aggregate of system states derived from repeating components in the cycle architectural styles

Type 1: A caller S_i calls to a responder S_j, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(Q),\ S_j \in \delta(S)\right\}$.

1.1: The caller is in parallel, where $S_i \in \delta(Q) \cap \delta(P)$.

1.2: The caller is in interrupt fault tolerance, where $S_i \in \delta(Q) \cap \delta(T)$.

1.3: The caller is in non-interrupt fault tolerance, where $S_i \in \delta(Q) \cap \delta(N)$.

1.4: The caller is not in parallel, interrupt fault tolerance and non-interrupt fault tolerance, where $S_i \in \delta(\mathrm{Q}) - [\delta(P) \cup \delta(T) \cup \delta(N)]$.

Type 2: A responder S_i returns to a caller S_j, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(S), S_j \in \delta(Q)\right\}$.

2.1: The responder is in parallel, where $S_i \in \delta(S) \cap \delta(P)$.

2.2: The responder is in interrupt fault tolerance, where $S_i \in \delta(S) \cap \delta(T)$.

2.3: The responder is in non-interrupt fault tolerance, where $S_i \in \delta(S) \cap \delta(N)$.

2.4: The responder is not in parallel, interrupt fault tolerance and non-interrupt fault tolerance, where $S_i \in \delta(S) - [\delta(P) \cup \delta(T) \cup \delta(N)]$.

Type 3: A state in sequence batch calls to a non-responder, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(B),\ S_j \in \delta - \delta(S)\right\}$.

Type 4: A non-caller/non-responder in parallel calls to a non-responder, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(P) - [\delta(Q) \cup \delta(S)],\ S_j \in \delta - \delta(S)\right\}$.

Type 5: A non-caller/non-responder in interrupt fault tolerance calls to a non-responder, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(T) - [\delta(Q) \cup \delta(S)],\ S_j \in \delta - \delta(S)\right\}$.

Type 6: A non-caller/non-responder in non-interrupt fault tolerance calls to a non-responder, which is regarded as $\left\{S_i \to S_j \middle| S_i \in \delta(N) - [\delta(Q) \cup \delta(S)],\ S_j \in \delta - \delta(S)\right\}$.

Type 7: S_i is a cycler, which is regard as $\{S_i \to S_i | S_i \in \delta(D)\}$.

3.6 Reliability assessment model based on component impact and architecture analysis

By employing the matrix representation of a directed graph, coupled with computational tools such as MATLAB, the significance of each individual component can be determined. Following this evaluation, components, as they are represented under a range of architectural paradigms, are translated into distinct system states. This allows for the generation of a comprehensive state model, which draws its foundations from the overarching state transition model. Subsequent to this modeling, the results extracted can be employed to evaluate the overall reliability of the software system[41, 42]. Underpinning the reliability assessment model for the software are some foundational assumptions:

(1) Every component present within the software system operates independently, without reliance or impact on other components.

(2) The mechanism of control transition between components is effectively captured and represented by the Markov Process.

A pivotal phase in the creation of the reliability assessment model for component-based software revolves around the formulation of the one-step transition probability matrix. Suppose Q' represents an $m \times m$ transition probability matrix, regarded as $Q' = \{S(i, j)\}$, where $S(i, j)$ indicates the probability of the successful transition from system state S_i to system state S_j. The rules for constructing the one-step transition probability matrix Q' are as follows:

(1) Rule 1: If no transition is observed from system state S_i to system state S_j, which is regarded as $S_i \nrightarrow S_j$, then transition probability $S(i, j)$ is equal to 0.

(2) Rule 2: If a transition is observed from system state S_i to system state S_j belonging to type 1, then transition probability $S(i, j)$ is equal to $P_{i,j}$.

(3) Rule 3: If a transition is observed from system state S_i to system state S_j belonging to one of the types from Type 2 to 7, then transition probability $S(i, j)$ is equal to $R_i P_{i,j}$.

Meanwhile, the formal description of the aforementioned rules for constructing transition probability matrix Q' can be expressed as

$$S(i, j) = \begin{cases} 0, & S_i \nrightarrow S_j; \\ P_{i,j}, & \text{Type 1}; \\ R_i P_{i,j}, & \text{otherwise} \end{cases} \tag{9}$$

where $\sum_{j=1}^{m} P_{i,j} = 1$. The value of $P_{i,j}$ can be calculated by

$$P_{i,j} = \frac{n(i, j)}{\sum_{k=1}^{m} n(i, k)} \tag{10}$$

where $n(i, j)$ denotes the total number of execution times from system state S_i to system state S_j during a complete operation of the component-based software system[43, 44].

Rule 3 can be further subdivided into the following four sub-rules on the basis of the different computations of the reliability of system state S_i.

(1) Sub-rule 3.1: If system state S_i is in the sequence batch style, a responder, or a cycler, including Types 2.4, 3, and 7, then the reliability of state S_i is $R_i = r_i^{\lambda_i}$.

(2) Sub-rule 3.2: If system state S_i is in the parallel style, including Types 2.1 and 4, then the reliability of state S_i is $R_i = \prod_{i=1}^{n} r_i^{\lambda_i}$, where n represents the total number of all components in the parallel state.

(3) Sub-rule 3.3: If system state S_i is in the interrupt fault-tolerance style, including Types 2.2 and 5, then the reliability of state S_i is $R_i = 1 - \prod_{i=1}^{n} \left(1 - r_i^{\lambda_i}\right)$, where n represents the total number of all components in the interrupt fault-tolerance state.

(4) Sub-rule 3.4: If system state S_i is in the non-interrupt fault-tolerance style, including Types 2.3 and 6, then the reliability of state S_i is $R_i = \sum_{k=M}^{n} r_i^{k\lambda_i} \left(1 - r_i^{\lambda_i}\right)^{n-k}$, where n represents the total number of all components in the non-interrupt fault-tolerance state.

Similarly, the formal description of the aforementioned sub-rules for calculating the reliability of state S_i can be described as

$$R_i = \begin{cases} \prod_{i=i}^{n} r_i^{\lambda_i}, & S_i \in \delta(P); \\ 1 - \prod_{i=1}^{n} \left(1 - r_i^{\lambda_i}\right), & S_i \in \delta(T); \\ \sum_{k=M}^{n} r_i^{k\lambda_i} \left(1 - r_i^{\lambda_i}\right)^{n-k}, & S_i \in \delta(N); \\ r_i^{\lambda_i}, & \text{otherwise} \end{cases} \tag{11}$$

Utilizing the established rules previously discussed, we can derive the one-step transition probability matrix, denoted as Q'. Once the matrix Q' is fully defined and completed, it enables the use of the discrete time Markov chain for the purpose of forecasting the reliability of software that is component-based. Consequently, the subsequent stages involve crafting the software reliability assessment model in the following manner:

$$R_S = (-1)^{m+1} R_m^{+} \frac{|E'|}{|I - Q'|} \tag{12}$$

where E' is the remaining matrix excluding the last row and first column of matrix $(I - Q')$ and R_m^{+} represents the reliability of end state S_m considering the component influence. Finally, the reliability of the component-based software system can be evaluated on the basis of Eq. (12) by using MATLAB.

4 Experiment and Analysis

In this section, a practice bank transaction software system[20, 27, 45] is evaluated to demonstrate the effectiveness of the proposed model. First, the directed graph model of the bank transaction system, which comprises 10 components, is developed, as shown in Fig. 9. C_1 is the input component, which is triggered by an external event. C_{10} is the output component, which

is mainly used to complete the bill printing function. Components C_2 and C_3 provide currency transaction and exchange, respectively. Components C_4 and C_5 are used for task scheduling, and C_5 is a standby component of the main component C_4. Components C_6 and C_7 are in parallel and are simultaneously executed to pass output to C_{10}. In particular, components C_6, C_7, and C_8 are requesters to C_9. Component C_9 can provide the exchange rate of the current currency. Reliabilities, average execution times, and transition probabilities of components are from Refs. [27, 46] and listed in Table 2.

4.1 Reliability assessment

Considering this case for a detailed analysis, the process of the proposed model is shown as follows.

(1) In accordance with the directed graph model of the bank transaction software system, the component influence is calculated on the basis of Eq. (1). In this case, the degree distribution of nodes is uniform. To simplify calculations, we set $\alpha_1 = \alpha_2 = 1/3$ to balance the impact of failure influence, propagation influence, and self-influence. The obtained results are normalized, as listed in Table 3.

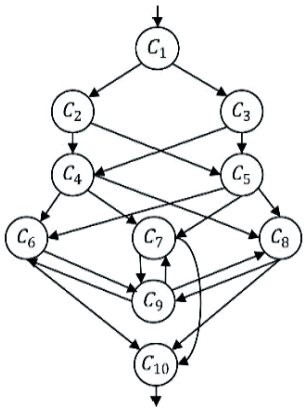

Fig. 9 Directed graph model of bank transaction system.

Table 2 Reliabilities, execution time, and transition probabilities.

C_i	R_i	$E(\mu_i)$	$P_{i,j}$
C_1	0.999	1	$P_{1,2} = 0.49$, $P_{1,3} = 0.51$
C_2	0.974	2	$P_{2,4} = P_{2,5} = 1.00$
C_3	0.970	1	$P_{3,4} = P_{3,5} = 1.00$
C_4	0.982	2	$P_{4,6} = P_{4,7} = 0.80$, $P_{4,8} = 0.20$
C_5	0.960	2	$P_{5,6} = P_{5,7} = 0.80$, $P_{5,8} = 0.20$
C_6	0.999	1	$P_{6,9} = 0.75$, $P_{6,10} = 0.25$
C_7	0.985	1	$P_{7,9} = 0.75$, $P_{7,10} = 0.25$
C_8	0.992	2	$P_{8,9} = 0.75$, $P_{8,10} = 0.25$
C_9	0.975	3	$P_{9,6} = P_{9,7} = P_{9,8} = 1.00$
C_{10}	0.964	1	–

(2) On the basis of the global state transition model of component-based software in Section 2, the components under various architectural styles are transformed into specific system states. For example, components C_4 and C_5 are merged into an interrupt fault-tolerance state S_T, and components C_6 and C_7 are regarded as a parallel state S_p. Meanwhile, the reliability of each state is evaluated with the analysis of component influence, as shown in Table 3.

(3) These independent states are integrated into the global state model of the bank transaction software system, and Fig. 10 depicts the obtained global state model of the bank system[47, 48].

(4) After constructing the transition probability matrix, we evaluate the system reliability on the basis of Eq. (12) by using MATLAB. The reliability assessment result of the bank transaction software system is 0.853.

We select six other software reliability analysis methods for comparison, namely, the models of Cheung[15], Xie et al.[21], Li and Hoang[26], Xue et al.[27], Lo et al.[30], and Wang et al.[41], to demonstrate the

Table 3 Process for computation.

C_i	λ_i	S_i	R_i	$P_{i,j}$
C_1	1.0000	S_1	0.9990	$P_{1,2} = 0.49$, $P_{1,3} = 0.51$
C_2	0.7767	S_2	0.9797	$P_{2,T} = 1.00$
C_3	0.7115	S_3	0.9786	$P_{3,T} = 1.00$
C_4	0.7930	S_T	0.9995	$P_{T,P} = 0.80$, $P_{T,8} = 0.20$
C_5	0.7930	S_T	0.9995	$P_{T,P} = 0.80$, $P_{T,8} = 0.20$
C_6	0.7851	S_P	0.9874	$P_{P,9} = 0.75$, $P_{P,10} = 0.25$
C_7	0.7851	S_P	0.9874	$P_{P,9} = 0.75$, $P_{P,10} = 0.25$
C_8	0.8256	S_8	0.9934	$P_{8,9} = 0.75$, $P_{8,10} = 0.25$

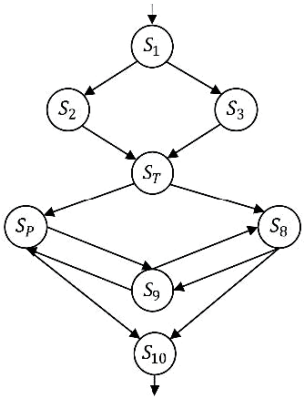

Fig. 10 State view of bank transaction software system.

effectiveness of the proposed model. Table 4 lists the evaluation results.

According to Ref. [27], the estimated result of the bank transaction system through software reliability testing is 0.865, which can be regarded as the actual reliability of this system. Figure 11 shows the accuracy of the aforementioned reliability analysis methods.

As shown in Table 4, the evaluated reliabilities of models Cheung[15], Xie et al.[21], and Lo et al.[30] are 0.821, 0.803, and 0.707, respectively, which are lower than the actual reliability of the software system. This phenomenon can be explained by two main reasons. First, the aforementioned models do not consider the diversity of software architectural styles in large and complex systems. The model of Cheung[15] is only suitable for predicting the reliability of the software with simple sequence batch architecture. The models of Li and Hoang[26] and Lo et al.[30] evaluate system reliability without analysing various software architectural styles. Furthermore, none of the aforementioned methods discusses the reliability enhancement brought by fault-tolerance mechanisms. Therefore, the assessment results of the aforementioned methods are pessimistic. Moreover, the evaluated reliability of the model of Wang et al.[41] is 0.831, which is higher than the assessment result of the model of Cheung[15]. The reason is that the model of Wang et al.[41] analyses the reliabilities under different software architectural styles and considers the effects of fault-tolerance architecture. However, this model assumes that the reliability of each component is constant without considering the influence of the operational profile, which affects the accuracy of the evaluation result. In addition, the evaluated reliabilities of the methods of Lo et al.[30] and Wang et al.[41] are 0.893 and 0.892, respectively. Both methods also consider the diversity problem of software architectural styles and use the average values execution time of components to describe the influence of the operational profile. However, the two methods ignore the impacts of failure correlation and fault propagation between components. Thus, considering the effects of different components in reliability evaluation is necessary because failures of different components have different impacts on system reliability. The assessment results of the two methods are higher than the actual reliability of the software system[47, 48]. On the basis of component failure analysis, the proposed model introduces the component influence to describe the effects of different components on software reliability. The model also establishes the global state transition model of the system to address the diversity problems of software architectural styles flexibly, which can obtain the scientific and reasonable assessment result of software reliability. The deviation between the reliability evaluation result and the actual software reliability of the proposed model is extremely small, at only 1.39%. The comparison results from Fig. 11 also show that the

Table 4 Comparison of results on methods for software reliability assessment.

Assessment method	Effects of component	Software architecture	R_S
Cheung[15]	–	Sequence batch	0.821
Xie et al.[21]	Obtained by calculating the impact factor	–	0.803
Li and Hoang[26]	Execution times	Sequence batch Cycle Parallel	0.893
Xue et al.[27]	Execution times	Sequence batch Parallel Fault tolerance Call and return	0.892
Lo et al.[30]	Execution times	–	0.707
Wang et al.[41]	–	Sequence batch Parallel Fault tolerance Call and return	0.831
Our proposed	Obtained by calculating the component influence	Sequence batch Parallel Fault tolerance Call and return Cycle	0.853

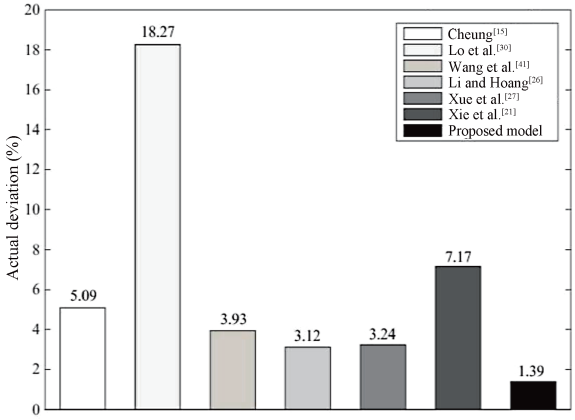

Fig. 11 Accuracy of software reliability evaluation methods.

proposed model is significantly accurate in reliability assessment compared with similar methods.

4.2 Impact of weight coefficient

The determination of component importance in our model is influenced by weight coefficients α_1 and α_2, which guide the integration of self-influence, failure influence, and fault propagation influence. To gain insight into the influence of these coefficients, we conduct a comparative experiment. Considering three scenarios:

(1) With $\alpha_1 = 1$ and $\alpha_2 = 0$, component importance is purely based on failure influence.

(2) When $\alpha_1 = 0$ and $\alpha_2 = 1$, the emphasis is entirely on fault propagation influence.

(3) Setting both α_1 and α_2 to 0 shifts the focus solely to self-influence.

Our experimental data presented in Fig. 12 delineate the variance in component importance under different weight coefficients. Notably, components C_2 and C_3 have the least significance when measured by either the failure influence or self-influence. In stark contrast, their importance peaks when evaluated through fault propagation influence. In a similar vein, components C_6 and C_7 display higher importance through failure or self-influence. Yet, their values of importance diminish when the lens is shifted to fault propagation influence.

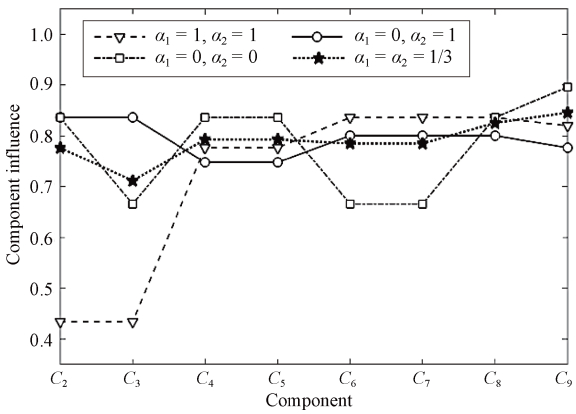

Fig. 12 Impact of weight coefficient.

In our model, the calculations associated with failure and fault propagation influences are intrinsically tied to a component's in-degree and out-degree. The observed patterns suggest that components C_2 and C_3 have a considerably higher number of outgoing edges compared to incoming ones. On the contrary, components C_6 and C_7 possess fewer outgoing edges relative to their incoming connections. Relying solely on either the failure or fault propagation influence can skew the perceived importance, especially in software with unbalanced component degree distributions. Additionally, given that self-influence encapsulates the execution frequency of components, it remains a pivotal factor in software reliability analysis.

In summation, the selection of weight coefficient values should take into account the degree distribution of components within a software system. Our future endeavors will further probe this relationship, seeking to solidify the conditions for the most accurate representation of component importance.

4.3 Reliability optimization

Methods to boost system reliability can be categorized into three main strategies:

(1) Enhancing the reliability of individual component.

(2) Integrating redundant components in a parallel configuration.

(3) Refining component reliability while also incorporating redundant parallel components[14], which is a synthesis of the first two approaches.

The model we propose offers an analytical insight into the influence exerted by individual component on the overall software reliability. By identifying components with a pronounced impact on reliability, this model furnishes crucial data that can be leveraged to strategically enhance the comprehensive reliability of the software system. This means that, rather than adopting a generic approach, developers and engineers can focus their reliability improvement efforts on components that matter most, optimizing resources and ensuring a more robust system outcome. Table 3 shows that component C_9 has the largest influence, and

component C_3 has the smallest influence among all the ordinary components. To present this phenomenon further intuitively, Fig. 13 shows the change in system reliability as the component influence of C_3 or C_9 increases. Each curve represents the derivatives of system reliability with different component influences in Fig. 13. The software reliability continues to decrease with a faster speed than C_3 as the component influence of C_9 increases. The analysis unveils that in the context of the bank transaction software system, component C_9 plays a pivotal role in contrast to C_3. The overarching influence of C_9 suggests its extensive interconnections with other components, thereby having a profound impact on the system's overall reliability. As a strategic move, focusing on enhancing the reliability of C_9 would consequently yield a more substantial elevation in system reliability as compared to concentrating on C_3.

To substantiate the veracity of these findings, we orchestrate a series of verification experiments grounded in the methodologies outlined in literature[13]. We initiate our experiment by setting a baseline reliability of 0.95 for both components C_3 and C_9. Following this, we incrementally elevate their reliabilities from this benchmark to a higher reliability of 0.99. The objective is to discern the repercussions of such improvements on the global system reliability. For a comprehensive evaluation, we employ three renowned reliability assessment models: Cheung's model[15], Wang et al.'s framework[41], and the model by Lo et al.[30]. Figures 14 – 16 visually represent the outcomes of these analyses. Consistently across these figures, it is evident that accentuating the reliability of component C_9 yields a more pronounced boost in software reliability than similar improvements to C_3.

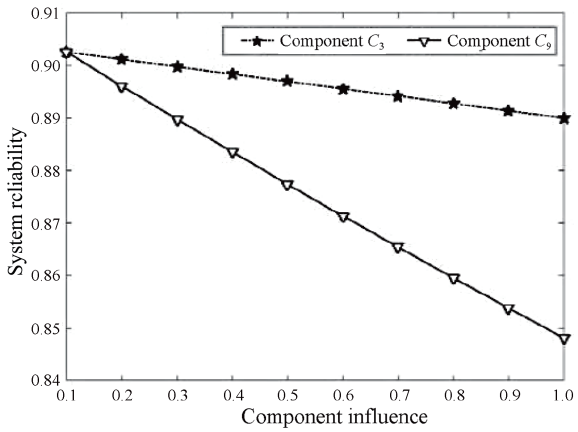

Fig. 13 Variability by component influence.

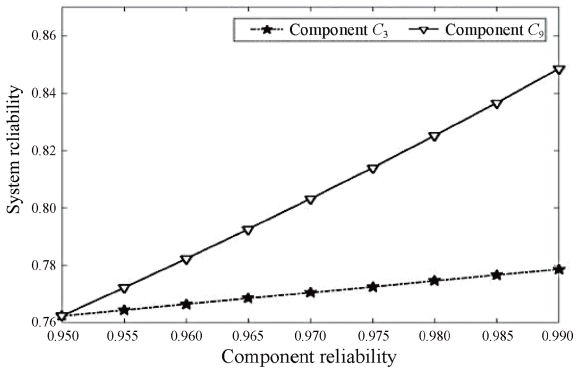

Fig. 14 Increase of system reliability tested by Cheung's model[15].

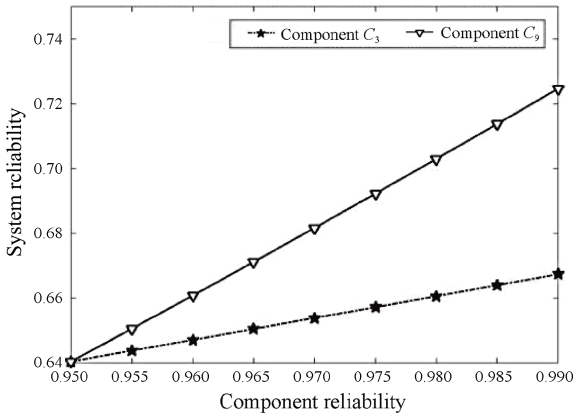

Fig. 15 Increase of system reliability tested by the model of Lo et al.[30]

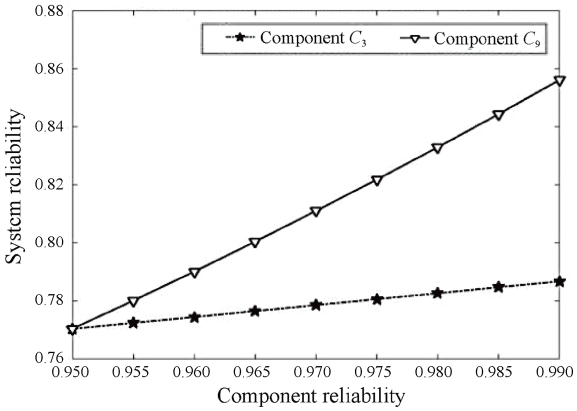

Fig. 16 Increase of system reliability tested by the model of Wang et al.[41]

This consistency corroborates the findings posited by our proposed model.

Conclusively, this underscores an imperative insight: components that exert significant influence within a system can be strategic levers to augment system reliability. Leveraging the insights from the proposed model can empower software architects to pinpoint

these key components. By doing so, resources can be judiciously allocated to ensure that system reliability enhancements are achieved in a cost-effective manner.

4.4 Threats to validity

Following are the potential threats to the validity of the study findings:

4.4.1 Internal validity

• **Influence of component weights:** The weight coefficients α_1 and α_2 are used to calculate component importance based on self-influence, failure influence, and fault propagation influence. However, they may not represent all software systems uniformly.

• **Assumptions:** The study makes certain assumptions, such as components being independent, which might not hold true for all software systems.

4.4.2 External validity

• **Generalization:** While the model has been validated on certain types of software systems, its applicability across a wide range of systems, especially those with unique architectures or workflows, remains untested.

• **Comparative analysis:** Though the model's results are compared with three classic reliability assessment models, there might be newer models or those not considered which might yield different results.

4.4.3 Construct validity

• **Modeling constraints:** The transformation of components under architectural styles into specific system states is based on the global state transition model. The accuracy and completeness of this transformation process can influence the outcomes.

• **Metric selection:** The selection of in-degree and out-degree of components as the only parameters for failure influence and fault propagation influence might overlook other potential influential factors.

4.4.4 Conclusion validity

• **Reliability enhancement methods:** The study suggests that improving the reliability of a high-influence component will enhance system reliability. However, it does not consider the potential trade-offs or limitations of doing so.

• **Predictive power:** While the proposed model has shown to predict reliability under specific conditions, its overall predictive power across various scenarios and system complexities needs further evaluation.

5 Implication

Now we report the industrial and research implications of the study findings as follows.

5.1 Industrial implications

For industries venturing into software development, the emphasis on component reliability can guide resource allocation and enhance overall system performance. Our model's methodology of identifying and optimizing critical components can be applied to pinpoint key algorithms or functions, significantly boosting software reliability. This approach is particularly relevant for industries integrating software in fields like cryptography, drug discovery, and optimization problems, where reliability is paramount. Additionally, as commercial computing platforms become more accessible, industries can use our reliability assessment models to develop and refine their own assessment frameworks, ensuring that their software platforms are both efficient and trustworthy.

5.2 Research implications

Our study provides a foundational guide for academics to develop advanced software reliability theories. By introducing the "component influence" metric and adapting a state transition model, we offer new avenues for research into the probabilistic models needed to cater to the complexities of modern software systems. The interaction between software components and their state transitions can inspire further research, particularly in understanding and mitigating reliability bottlenecks. Collaborative partnerships between industry and academia can leverage these insights, fostering joint ventures to develop and enhance software reliability models, ultimately combining theoretical expertise with practical applications.

6 Conclusion and Future Avenue

In the evolving landscape of software, as algorithms and systems become increasingly intricate, the necessity to consider the reliability of each component becomes evident, particularly given the inherent probabilistic nature of certain elements. Adapting the idea from our study, the "component influence" is extrapolated to encompass the unique impacts of various elements on the overall reliability of software. This adaptation aims to offer enhanced precision and efficacy in the realm of software reliability analysis. Algorithms and circuits naturally showcase a wide range of architectural complexities. This intricacy is reminiscent of the diverse architectural styles observed in software. To navigate these challenges, we introduce

a global state transition model tailored for various settings. This model provides an in-depth method to transition various components into states, addressing the diverse architecture intrinsic to computations.

Integrating the nuanced component influence with the global state transition model, we present a refined reliability assessment model specific to software. Using a simulated environment, the potential effectiveness of this model is demonstrated. The early results suggest that the model significantly enhances reliability assessment accuracy compared to its counterparts. Moreover, this model has the potential to identify pivotal components that greatly influence overall software reliability. Such insights are invaluable, enabling focused improvements during the preliminary phases of software design.

Future research avenues will prioritize the validation of the proposed model across genuine projects and comprehensive experimentation on real platforms. This effort aims to solidify the model's robustness and to discern exact quantitative conditions for its application. Given the dynamism and rapid advancements in computing, the model will be continuously refined to account for the changing nature of states and architectures.

Additionally, we will expand our case studies to enhance the practical relevance and robustness of our findings. While our current study provides initial validation through a simulated environment, future research will focus on incorporating a broader range of case studies and real-world applications. These additional case studies will include various software systems operating in different domains, such as cryptography, optimization, and machine learning. By applying our model to these diverse scenarios, we aim to further demonstrate its applicability and reliability in real-world computing environments. This will not only strengthen the practical utility of our methodology but also provide valuable insights into the unique challenges and performance considerations of software in various applications.

Acknowledgment

This research was funded by Taif University, Saudi Arabia (No. TU-DSPP-2024-41).

References

[1] S. Yacoub, B. Cukic, and H. H. Ammar, A scenario-based reliability analysis approach for component-based software, *IEEE Trans. Reliab.*, vol. 53, no. 4, pp. 465–480, 2004.

[2] S. Yacoub, B. Cukic, and H. H. Ammar, A scenario-based reliability analysis approach for component-based software, *IEEE Transactions on Reliability*, vol. 53, no. 4, pp. 465–480, 2004

[3] M. Palviainen, A. Evesti, and E. Ovaska, The reliability estimation, prediction and measuring of component-based software, *J. Syst. Software*, vol. 84, no. 6, pp. 1054–1070, 2011.

[4] S. S. Gokhale and K. S. Trivedi, Reliability prediction and sensitivity analysis based on software architecture, in *Proc. 13th Int. Symp. Software Reliability Engineering,* Annapolis, MD, USA, 2002, pp. 64–75.

[5] Y. Wang, H. Liu, H. Yuan, and Z. Zhang, Comprehensive evaluation of software system reliability based on component-based generalized G-O models, *PeerJ Comput. Sci.*, vol. 9, p. e1247, 2023.

[6] R. Mirandola, P. Potena, E. Riccobene, and P. Scandurra, A reliability model for service component architectures, *J. Syst. Software*, vol. 89, pp. 109–127, 2014.

[7] S. Oveisi, A. Moeini, S. Mirzaei, and M. A. Farsi, Software reliability prediction: A survey, *Qual. Reliab. Eng. Int.*, vol. 39, no. 1, pp. 412–453, 2023.

[8] F. H. C. Ferreira, E. Y. Nakagawa, and R. P. dos Santos, Towards an understanding of reliability of software-intensive systems-of-systems, *Inf. Software Technol.*, vol. 158, p. 107186, 2023.

[9] K. J. Chen and C. Y. Huang, Using modified diffusion models for reliability estimation of open source software, *IEEE Access*, vol. 11, pp. 51631–51646, 2023.

[10] J. Faraji, M. Aslani, H. Hashemi-Dezaki, A. Ketabi, Z. De Grève, and F. Vallée, Reliability analysis of cyber-physical energy hubs: A Monte Carlo approach, *IEEE Trans. Smart Grid*, vol. 15, no. 1, pp. 848–862, 2024.

[11] E. S. Seddigh, M. R. Haghifam, and H. R. Baghaee, Reliability assessment of multi-microgrids considering collaborations at PCC, *IEEE Trans. Ind. Appl.*, vol. 60, no. 2, pp. 2357–2365, 2024.

[12] S. Brin and L. Page, Reprint of: The anatomy of a large-scale hypertextual web search engine, *Comput. Netw.*, vol. 56, no. 18, pp. 3825–3833, 2012.

[13] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, Architecture-based reliability prediction with the palladio component model, *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1319–1339, 2012.

[14] A. Chatzigeorgiou, N. Tsantalis, and G. Stephanides, Application of graph theory to OO software engineering, in *Proc. Int. Workshop on Interdisciplinary Software Engineering Research*, Shanghai, China, 2009, pp. 29–36.

[15] R. C. Cheung, A user-oriented software reliability model, *IEEE Trans. Software Eng.*, vol. SE-6, no. 2, pp. 118–125, 1980.

[16] W. Dong, H. Ning, and Y. Ming, Reliability analysis of component-based software based on relationships of components, in *Proc. IEEE Int. Conf. Web Services*, Beijing, China, 2008, pp. 814–815.

[17] A. Filieri, C. Ghezzi, V. Grassi, and R. Mirandola, Reliability analysis of component-based systems with multiple failure modes, in *Proc. 13th Int. Symp. Component-Based Software Engineering*, Prague, Czech

Republic, 2010, pp. 1–20.
[18] R. E. Kharboutly and S. S. Gokhale, Efficient reliability analysis of concurrent software applications considering software architecture, *International Journal of Software Engineering and Knowledge Engineering*, vol. 24, no. 1, pp. 43–60, 2014
[19] K. Goseva-Popstojanova, A. P. Mathur, and K. S. Trivedi, Comparison of architecture-based software reliability models, in *Proc. 12th Int. Symp. Software Reliability Engineering*, Hong Kong, China, 2001, pp. 22–31.
[20] K. Goševa-Popstojanova and K. S. Trivedi, Architecture-based approach to reliability assessment of software systems, *Perform. Eval.*, vol. 45, nos. 2 & 3, pp. 179–204, 2001.
[21] J. Y. Xie, J. X. An, and J. H. Zhu, NHPP software reliability growth model considering imperfect debugging, (in Chinese), *J. Software*, vol. 21, no. 5, pp. 942–949, 2010.
[22] V. E. Johnson, A. Moosman, and P. Cotter, A hierarchical model for estimating the early reliability of complex systems, *IEEE Trans. Reliab.*, vol. 54, no. 2, pp. 224–231, 2005.
[23] K. Li, L. Liu, J. Zhai, T. M. Kosgoftaar, M. Shao, and W. Liu, Reliability evaluation model of component-based software based on complex network theory, *Qual. Reliab. Eng. Int.*, vol. 33, no. 3, pp. 543–550, 2017.
[24] K. Li, M. Yu, L. Liu, J. Zhai, and W. Liu, A novel reliability analysis approach for component-based software based on the complex network theory, *Software Test., Verif. Reliab.*, vol. 28, no. 6, p. e1674, 2018.
[25] W. Kuo and V. R. Prasad, An annotated overview of system-reliability optimization, *IEEE Trans. Reliab.*, vol. 49, no. 2, pp. 176–187, 2000.
[26] Q. Li and P. Hoang, NHPP software reliability model considering the uncertainty of operating environments with imperfect debugging and testing coverage, *Appl. Math. Modell.*, vol. 51, pp. 68–85, 2017.
[27] L. Xue, Z. Zhang, and D. Zuo, DCR: Double component ranking for building reliable cloud applications, *Telecommun. Comput. Electron. Control*, vol. 14, no. 4, pp. 1565–1574, 2016.
[28] L. Xue, D. Zuo, Z. Zhang, and N. Wu, A novel component ranking method for improving software reliability, *IEICE Trans. Inf. Syst.*, vol. E100.D, no. 10, pp. 2653–2658, 2017.
[29] J. H. Lo, S. Y. Kuo, M. R. Lyu, and C. Y. Huang, Optimal resource allocation and reliability analysis for component-based software applications, in *Proc. 26th Annu. Int. Computer Software and Applications*, Oxford, UK, 2002, pp. 7–12.
[30] J. H. Lo, C. Y. Huang, I. Y. Chen, S. Y. Kuo, and M. R. Lyu, Reliability assessment and sensitivity analysis of software reliability growth modeling based on software module structure, *J. Syst. Software*, vol. 76, no. 1, pp. 3–13, 2005.
[31] S. Malek, M. Mikic-Rakic, and N. Medvidovic, A style-aware architectural middleware for resource-constrained, distributed systems, *IEEE Trans. Software Eng.*, vol. 31, no. 3, pp. 256–272, 2005.
[32] J. D. Musa, *Software Reliability Engineering*. New York, NY, USA: McGraw-Hill, 1998.
[33] N. Wu, D. Zuo, Z. Zhang, P. Zhou, and Y. Zhao, FSCRank: A failure-sensitive structure-based component ranking approach for cloud applications, *IEICE Trans. Inf. Syst.*, vol. E102.D, no. 2, pp. 307–318, 2019.
[34] T. T. Pham and X. Défago, Reliability prediction for component-based systems: Incorporating error propagation analysis and different execution models, in *Proc. 12th Int. Conf. Quality Software*, Xi'an, China, 2012, pp. 106–115.
[35] T. T. Pham, F. Bonnet, and X. Défago, Reliability prediction for component-based software systems with architectural-level fault tolerance mechanisms (extended version), *J. Wireless Mobile Netw., Ubiquitous Comput., Dependable Appl.*, vol. 5, no. 1, pp. 4–36, 2014.
[36] T. T. Pham, X. Défago, and Q. T. Huynh, Reliability prediction for component-based software systems: Dealing with concurrent and propagating errors, *Sci. Comput. Program.*, vol. 97, pp. 426–457, 2015.
[37] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, Hora: Architecture-aware online failure prediction, *J. Syst. Software*, vol. 137, pp. 669–685, 2018.
[38] Q. Wang, Y. Lu, H. Fang, and X. L. Zhu, Complex software reliability evaluation method based on architecture analysis, (in Chinese), *J. Syst. Eng.*, vol. 28, no. 2, pp. 271–284, 2013.
[39] A. P. Singh and P. Tomar, A new model for reliability estimation of component-based software, in *Proc. 3rd IEEE Int. Advance Computing Conf.*, Ghaziabad, India, 2013, pp. 1431–1436.
[40] H. A. Stieber, L. Hu, and W. E. Wong, Estimation of the total number of software failures from test data and code coverage: A Bayesian approach, in *Proc. IEEE Int. Symp. Software Reliability Engineering Workshops*, Toulouse, France, 2017, pp. 234–238.
[41] W. L. Wang, D. Pan, and M. H. Chen, Architecture-based software reliability modeling, *J. Syst. Software*, vol. 79, no. 1, pp. 132–146, 2006.
[42] W. Lu, F. Xu, and J. Lv, An approach of software reliability evaluation in the open environment, (in Chinese), *Chin. J. Comput.*, vol. 33, no. 3, pp. 452–462, 2010.
[43] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, FTCloud: A component ranking framework for fault-tolerant cloud applications, in *Proc. IEEE 21st Int. Symp. Software Reliability Engineering*, San Jose, CA, USA, 2010, pp. 398–407.
[44] Z. Zheng, T. C. Zhou, M. R. Lyu, and I. King, Component ranking for fault-tolerant cloud applications, *IEEE Trans. Serv. Comput.*, vol. 5, no. 4, pp. 540–550, 2012.
[45] F. Wang, H. Zhu, G. Srivastava, S. Li, M. R. Khosravi, and L. Qi, Robust collaborative filtering recommendation with user-item-trust records, *IEEE Trans. Comput. Soc. Syst.*, vol. 9, no. 4, pp. 986–996, 2022.
[46] Y. Liu, Y. Zhang, X. Mao, X. Zhou, J. Chang, W. Wang, P. Wang, and L. Qi, Lithological facies classification using attention-based gated recurrent unit, *Tsinghua Science and Technology*, vol. 29, no. 4, pp. 1206–1218, 2024.
[47] T. Zhang, X. Zhou, J. Liu, B. Cheng, X. Xu, L. Qi, Q. Tian, and Z. Wan, QoE-driven data communication

framework for consumer electronics in Tele-healthcare system, *IEEE Trans. Consum. Electron.*, vol. 69, no. 4, pp. 719–733, 2023.

[48] J. Huang, B. Ma, M. Wang, X. Zhou, L. Yao, S. Wang, L. Qi, and Y. Chen, Incentive mechanism design of federated learning for recommendation systems in MEC, *IEEE Trans. Consum. Electron.*, vol. 70, no. 1, pp. 2596–2607, 2024.

Saleh Alyahyan is working as an assistant professor at Shaqra Univerity, Saudi Arabia. He received the PhD degree from University of East Anglia, UK. His research interests are artificial intelligence, machine learning, and classification ensemble methods.

Mohammed Naif Alatawi is currently working as an associate professor at University of Tabuk, Saudi Arabia. He received the MEng degree from Florida Institute of Technology, USA in 2015, and the PhD degree in computer information systems from Nova Southeastern University, USA in 2019. His research interests include computer security, software engineering, machine learning, web services, cloud computing, IoT, and E-learning.

Mrim M. Alnfiai is an associate professor of information technology at Taif University, Saudi Arabia. Her research interests are in assistive technology, human computer interaction, accessibility, usable security, AI, and machine learning. She has published several papers in ISI journals and participates in assistive technology, HCI, and accessibility conferences, including ASSETS, ANT, FNC, CIST, JAIHC, and ICCA. Currently, her research focuses on designing accessible tools for visually impaired people, including people with no or low vision. She has conducted several studies and experiences to understand visually impaired abilities and behaviors, and designed accessible systems that help them interact easily with technology. She has also published several papers related to accessibility and authentication mechanisms for visually impaired users. She has also published papers related to using Near Field Communication (NFC) technology and machine learning to enhance the healthcare system.

Shoayee Dlaim Alotaibi is an assistant professor at Department of Artificial Intelligence and Data Science, University of Hail, Saudi Arabia. Her research interests include smart city, internet of things, and machine learning techniques.

Abdullah Alshammari received the BEng degree from Tennessee State University, USA in 2016, and the MEng and the PhD degrees from Howard University, USA in 2018 and 2020, respectively. He is currently working as an assistant professor at College of Computer Science and Electrical Engineering, University of Hafr Albatin, Saudi Arabia. He has delivered keynotes and invited speeches at international conferences and workshops. He is engaged in research and teaching in the areas of cybersecurity, machine learning, big data analytics, and wireless networking for emerging networked systems, including cyber-physical systems, IoTs, smart cities, edge computing, cognitive city, mobile computing, network security, and artificial intelligence. Formerly, he led cybersecurity innovation, research, and development at NEOM and NEOM AUTHORITY, Saudi Arabia. In addition, he served as assistant professor of cybersecurity and artificial intelligence on the Royal Commission for Jubail and Yanbu.

Zaid Alzaid is a highly acclaimed computer science scholar, received the BEng degree from King Abdul-Aziz University, Saudi Arabia, the MEng degree in advanced internet applications from Heriot-Watt University, UK, and the PhD degree from Florida State University, USA from 2017 to 2020. He served as a teaching assistant and research assistant at Florida State University, USA, honing his academic and research skills. His academic journey began as a computer lecturer at Hail College of Technology, Saudi Arabia, where he taught from 2006 to 2014, playing a significant role in shaping the minds of aspiring technologists. Currently, he is an assistant professor at Islamic University of Madinah, Saudi Arabia, where he also serves as the dean of the IT Deanship, leading the way in innovation and research. His extensive work in High-Performance Computing (HPC), systems optimization, and AI underscores his impressive research portfolio. His contributions have been instrumental in the development of AI and HPC systems, enhancing both academic inquiry and the educational landscape within these domains.

Hathal Alwageed received the PhD degree in computer engineering from Stevens Institute of Technology, USA in 2019. He has been at College of Computer and Information Sciences, Jouf University, Saudi Arabia since 2019. He is currently an assistant professor. His current research interests include machine learning, deep learning, IoTs, and computer networks.