PAPER

Using Hierarchical Scenarios to Predict the Reliability of Component-Based Software

Chunyan HOU[†a)], **Jinsong WANG**[†], ***Nonmembers***, ***and*** **Chen CHEN**[††], ***Member***

SUMMARY System scenarios that derived from system design specification play an important role in the reliability engineering of component-based software systems. Several scenario-based approaches have been proposed to predict the reliability of a system at the design time, most of them adopt flat construction of scenarios, which doesn't conform to software design specifications and is subject to introduce state space explosion problem in the large systems. This paper identifies various challenges related to scenario modeling at the early design stages based on software architecture specification. A novel scenario-based reliability modeling and prediction approach is introduced. The approach adopts hierarchical scenario specification to model software reliability to avoid state space explosion and reduce computational complexity. Finally, the evaluation experiment shows the potential of the approach.

key words: *scenario, software reliability, software architecture, components*

1. Introduction

Software reliability is defined as the probability of failure-free operation of a software system for a specified period of time in a specified environment. Software reliability is one of the most important criteria to measure software quality, and determines whether or not a software system could run in a stable and reliable way. The traditional software development methodology, often represented by Waterfall, ensures high software reliability only through software testing during late development stages. Nowadays, the use of component-based iterative software development methodologies is increasing to satisfy the need to respond to fast moving market demand and for gaining market share [1]. In contrast to traditional methodology, component-based development methodologies evaluate software reliability based on software architecture already during early development stages [2]. This helps software architects to determine the software components mostly affecting system reliability, to study the sensitivity of the system reliability to component reliabilities, and to support decisions between different design alternatives.

Unfortunately, component-based development methodologies do not explicitly provide effective practices for managing and measuring quality and reliability [3]. First of all, the components in a component-based software system are usually developed by different third parties so that there are more reliability problems that could be caused from interaction between them. On the other hand, component providers usually keep the information about software development and testing secrete. Therefore, it is inconvenient for component users to model software failure behavior for their reliability estimation. This deficiency may prevent software development organizations in safety-critical domains from transforming from traditional development to highly iterative component-based development. Moreover, it is a hard task to perform architecture-based software reliability analysis. Software reliability depends not only on the component implementation, but also on operational profile, execution environment, system configuration and so on. Thereby software reliability is a non-isolated attribute, and it makes sense only in some specific context referred to as scenarios.

Scenarios are a popular means for capturing behavioral requirements of software systems early in the lifecycle. Nowadays, the use of scenarios to evaluate architecture from different perspective has been widely acknowledged by the industry and academia [4]. It can provide an insight into system qualities like maintainability, reliability, performance and so on [5]. Scenarios have been widely adopted as a way to specify the target to be evaluated. System scenarios derived from system design specification play an important role in the reliability engineering of component-based software systems. Architecture-based software reliability analysis can be regarded as a process to prove how well software architecture supports a variety of critical scenarios.

There has been some previous work on using scenarios to predict the reliability of component-based software. The main problem with most of the existing approaches is the number of states which known as state explosion problem, exactly, in case of large systems and flat construction of scenarios [6]. In contrast to hierarchical ones based on abstraction and approximation, flat approaches consider all possible execution paths to construct system scenarios. Albeit with better accuracy, they are un-applicable to large systems due to state space explosion problems. In response to this problem, we present a novel architecture-based approach to predict software reliability in this paper, which models scenarios and evaluates scenario reliability hierarchically. Hierarchical scenarios describe different kinds of profile that various software participators concern according to software design specifications. We construct an architecture model to describe profile dependencies between hierarchical scenarios from the perspective of software architects. At last,

Manuscript received April 12, 2017.
Manuscript revised September 20, 2017.
Manuscript publicized November 7, 2017.
[†]The authors are with Tianjin University of Technology, Tianjin, China.
[††]The author is with Nankai University, Tianjin, China.
a) E-mail: chunyanhou@163.com
DOI: 10.1587/transinf.2017EDP7127

reliability analysis based on hierarchical scenarios is performed to evaluate software reliability.

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 presents a software reliability model based on hierarchical scenarios named SRM to describe software architecture. Section 4 explains how to evaluate software reliability based on SRM. Section 5 documents the case study before Sect. 6 concludes the paper.

2. Related Works

During the last decade, researchers proposed several approaches to predict reliability at the design-time utilizing the architectural design of the software system and the reused components data; these approaches address different problems and challenges. In this section, we provide a brief view of scenario-based reliability analysis approaches which are greatest interest to the scope of this work.

Since early 1990s, scenario-based approaches have been widely adopted to evaluate software architecture model at the design time [7]. Software reliability is one of the most important non-functional properties that govern crucial architectural design decisions [8]. Therefore, people proposed special scenario-based reliability approaches to analyze software architecture, especially for component-based software.

In [9], a probabilistic model named component-dependency graph (CDG) is constructed using scenarios of component interactions. Based on CDG, a reliability analysis algorithm is developed to analyze the reliability of the system as a function of reliabilities of its architectural constituents. Reussner et al. [10] and Brosch et al. [11], [12] introduced reliability prediction approach for component-based software architectures that considers the relevant architectural factors of software systems by explicitly modeling the system usage profile and execution environment. Rodrigues et al. [13] further considers the influence that the concurrency of component-based software systems has on system reliability. In [14]–[16], the authors applied dynamic Bayesian networks to build a stochastic reliability model that relies on standard models of software architecture and does not require implementation-level artifacts. However, the above mentioned approaches build global behavior models of the system based on flat construction of scenarios [17], which in the large systems introduce state space explosion problem.

In order to solve this problem, Hou et al. [18] proposed extension to the work in [11] by separately modeling and evaluating each scenario to avoid state space explosion and reduce computational complexity. However, the approach analyzed case scenarios directly, and didn't consider the dependencies between different kinds of scenarios. Cheung et al. [19] and Ali, et al. [6] extended the work in [14] by modeling and calculating system scenarios hierarchically. The hierarchical method in [19] can provide solution in case of large systems, especially when the synchronization nature of the system implementation is taken into account. Ali, et al. [6] proposed a modeling and calculation approach that, pragmatically, models and divides scenarios by a scenario specification languages and finite state machine. However, these two approaches don't explicitly model software operational profile by illustrating the profile dependencies between different kinds of scenarios. Ao, et al. [20] built software operational profile by profile propagation between scenarios. However, the results are used to generate software reliability test cases, and not applicable for reliability analysis.

In summary, extension of a scenario specification toward partial behavior modeling is integral part that should be considered for predicting the reliability based on the architecture [21]. The approaches based on hierarchical scenarios have been applied to software reliability testing. At present, researchers are trying to use scenario-based approaches for software reliability estimation. Although some preliminary results have been obtained, there are a few of problems with existing approaches which significantly affects their practical applicability. In order to solve these problems, we propose a novel reliability analysis approach based on hierarchical scenarios.

3. Reliability Model

In this section a software reliability model named SRM is defined in every detail to describe the architecture of component-based software. A SRM formalizes the factors related to software reliability for the purpose of reliability prediction. A SRM is defined based on hierarchical scenario specification, which divides scenarios into four classes. Each class of scenarios is described with corresponding profile as shown in Fig. 1. The definition of a SRM meets the following assumptions.

(1) Software architecture is static, and each interaction between components is a synchronous communication.
(2) An application scenario is realized by several case scenarios collaborating with one another. The transitions between case scenarios are finite, and occur due to users' subjective decisions.
(3) Components' testing profile is different from their operational profile. Component developers perform testing without the knowledge of how they will be used in the future.

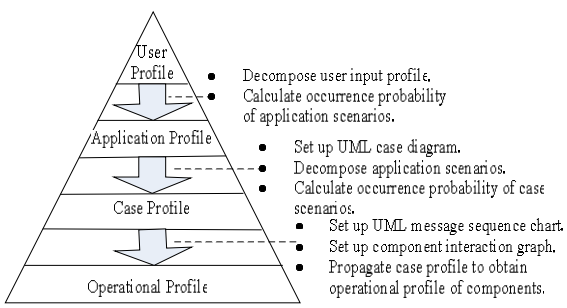

Fig. 1 The process for SRM to propagate profile.

3.1 Reliability Model Based on Hierarchical Scenarios

According to software reliability theory, software reliability significantly depends on operational profile [22]. Therefore, it is essential for software reliability model to describe operational profile accurately and objectively. Operational profile can be eventually constructed by continually refining software input space from top to down and determining input probability. Software reliability model (SRM) defines profile dependencies between different kinds of scenarios, and builds operational profile by the way of profile propagation from top to down as shown in Fig. 1. It can be seen that three steps are taken to obtain operational profile from user profile with the profile of application and case scenarios as intermediates. The process will be explained in detail in the text that follows. SRM is defined as Definition 1.

Definition 1. **(SRM)** A SRM is a software reliability model based on hierarchical scenario specification, and is defined by the tuple <STG[*n*] *stg*, CIG[*n*] *cig*, CDM *cdm*, Usage[] *usage*, double *R*>, where *n* is the number of case scenarios; *stg* is a set of scenario transition graphs each of which describes an application scenario; *cig* is a group of component interaction graphs representing how several components cooperate to realize basic system level functions; *cdm* is a component deployment model with the information about system configuration and deployment; *usage* is a collection of user profile from different types of users; and *R* is the reliability of a software system.

Definition 2. **(Profile)** Profile lists the input values and probability distribution for a parameter, and is defined as the tuple <string *name*, int *type*, <var *value*, double *P*>[] *item*>, where *name* and *type* are the name and type of a parameter respectively; and *item* is a set of input instances, where *value* is a discrete input value, and *P* is the probability that the *value* is input.

In the rest of this sub-section, four kinds of profile in Fig. 1 will be introduced from top to down in accordance with the direction to propagate profile. At first, user profile is defined as Definition 3.

Definition 3. **(Usage)** A Usage models usage profile for a type of users, and is defined by the tuple <UserType *type*, double *P*, <string *name*, double *P*>[] *app*>, where *type* denotes user type; *P* is the occurrence probability of a usage scenario; and *app* is a set of application scenarios triggered by a type of users, where *name* and *P* are the name and occurrence probability respectively.

Secondly, application profile is depicted with case-scenario transition graph (STG). A STG models an application scenario consisting of several case scenarios. Software designers typically employ UML Use Case Diagrams (UCD) to list case scenarios. Figure 2 shows an example of a UCD instance with seven case scenarios for Borrower, and Fig. 3 shows an example of a STG instance. It can be seen that a STG depicts how several cases cooperate with one another to realize a system application function. A STG model is a kind of directed graph with case scenarios as nodes and the transition between cases as edges. The definition of a STG is given as Definition 4.

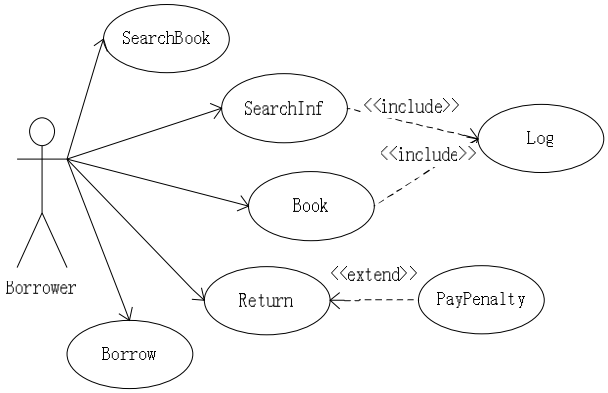

Fig. 2 A UCD example.

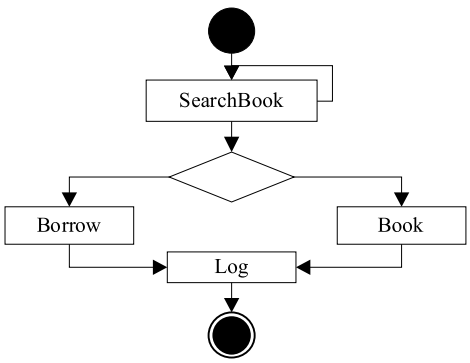

Fig. 3 A STG example.

Definition 4. (STG) A STG is an application scenario model, and is defined by the tuple <string *name*, double *P*, Profile[] *profile*, Case[] *case*, double[,] *trans*>, where *name* and *P* are the name and occurrence probability respectively; *profile* is input profile to user interface; *case* is a collection of included case scenarios; and *trans* denotes the transition probabilities between case scenarios.

Thirdly, case profile is used to describe case scenarios which realize basic system level functions. Case scenarios are defined from the perspective of software designers, whereas application scenarios are from software users. A case scenario is realized with several components to interact with one another. Case profile is defined as Definition 5.

Definition 5. **(Case)** A Case is a case scenario model, and is defined by the tuple < string *name*, CIG *cig*, Profile[] *profile*, double *P* >, where *name* is the name of a case scenario; *cig* is a component interaction graph to depict how the components interact; *profile* denotes input profile to case interface; and *P* are the probability that a case scenario is called.

Finally, operational profile is practical input profile to components, which is dependent on user input and different from testing profile. We construct Component Interaction Graphs (CIGs) to describe operational profile, which are the kernel part of SRM definition. In the next sub-section, CIGs

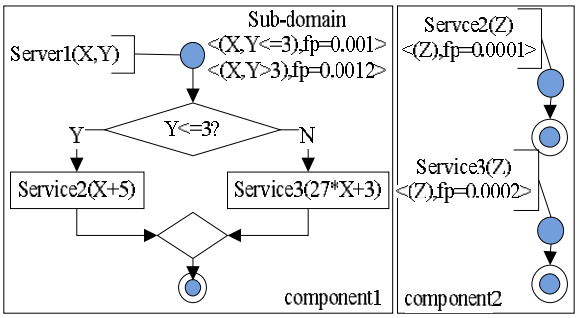

Fig. 4 A sample of CIG.

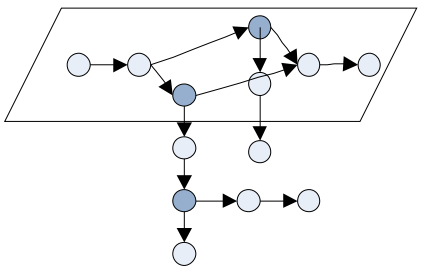

Fig. 5 A sample for formal CIG.

will be introduced in detail.

3.2 Component Interaction Graphs

A component interaction graph (CIG) depicts the architecture of a case scenario, which enables the propagation from case profile to operational profile. Software designers typically use Message Sequence Charts (MSCs) to describe component interactions, but they are not suitable for software reliability analysis. Compared with MSCs, CIGs are special profile models which focus on describing profile dependencies. In order to consider the difference between operational and testing profile, we adopt the concept of sub-domain [23] to define operational profile as input sub-domains and reliability or failure probability on sub-domains. Figure 4 shows an example of a CIG instance.

The CIG as shown in Fig. 4 is descriptive for convenience of software architects, which is formally extracted as a multi-layer directed graph behind the scene as shown in Fig. 5. It can be seen that there are two types of nodes—one denotes call actions while the other denotes the actions nesting calls. A CIG is defined as Definition 6.

Definition 6. **(CIG)** A CIG is a component interaction graph, and is defined by the tuple <string *name*, Chain<Action> *arch*, double *R*, Service[] *ser*>, where *name* is the name of the case scenario realized by a CIG; *arch* is a sequence of interacting actions; *R* is the reliability; and *ser* denotes a set of interfaces called in a case.

Definition 7. (Action) An action is a program internal action, and is defined by the tuple <ActionType *type*, string[] *exp*, Profile[] *profile*, CIG[] *child*, Action **next*, Service **owner*, <int *num*, double *P*>[] *ap*>, where *type* denotes the type of an action; *exp* is a set of arithmetic or Boolean expressions to describe parameter dependencies; *profile* is the input profile to a service after taking a action; *child* denotes sub-actions nested by a action; *next* points to the next action; *owner* points to the service an action belongs to; and *ap* denotes action profile, where *num* and *P* are the occurrence times and probability respectively.

Definition 8. (Service) A *Service* denotes an interface provided by a component. It is annotated by the tuple <string *name*, SType *type*, string *comp*, Sdomain[] *sd*, <string *name*, string *type*>[] *param*, Action **op*, double *R*, Profile[] *profile* >, where *name* is the name of a service; *type* includes atomic and non-atomic service to indicate whether a service calls other services; *component* is the name of the component offering a service; *Sd* is a set of sub-domains constituting input space; *param* are input parameters where *name* and type is parameters' name and *type* respectively; *op* points to the first action; *R* is practical reliability; and *profile* is operational profile.

Definition 9. (Sdomain) A Sdomain models one of the sub-domains constituting the input space of a service. It is annotated by the tuple < string *name*, list<var[*m*]> *item*, double *weight*, HardType[] *hardtype*, double *R*>, where *name* is an identifier for a specific sub-domain; *item* is a list of all input value composition included in a sub-domain, where *m* is the number of input parameters; *weight* is the probability that users' input falls into a sub-domain; *hardtype* indicates what kinds of hardware are required by software when running on a sub-domain; and *R* is testing reliability.

3.3 Component Deployment Model

The environment where a software application runs and the strategy how to deploy an application have a considerable influence on software reliability. A component deployment model (CDM) defined as Definition 10 is constructed to describe these reliability-related factors.

Definition 10. **(CDM)** A CDM is a component deployment model, and is defined by the tuple <Server[] *server*, Component[] *comp*>, where *server* is a set of servers used to deploy components; and *comp* is a collection of components in a software application.

Definition 11. (**Server**) A Server is defined by the tuple <string *name*, Hardware[] *hw* >, where *name* is used to identify a server; and *hw* describes hardware configuration.

Definition 12. (Component) A Component is defined as the tuple <string *name*, Service[] *service*, Server[] *server*>, where *name* is component's name; *service* is a group of external interfaces provided by a component; and *server* is one or more servers used to deploy a component.

Definition 13. (Hardware) Hardware models a kind of hardware in a server. It is annotated by the tuple <HardType *type*, int *MTTF*, int *MTTR*, double *fp*>, where *type* denotes the type of hardware; *MTTF* is mean time to failure; *MTTR* is mean time to repair; and *fp* is failure probability.

4. Reliability Analysis

In this section, we will explain how to evaluate software reliability based on SRM. At first, SRM profile attributes should be calculated. Profile propagation is used to construct operational profile from user profile. Based on the results, scenario-based reliability analysis is performed to analyze software reliability.

4.1 Profile Propagation

As shown in Fig. 1, software profile can be divided into user profile, application profile, case profile and operational profile. The lower profile is propagated from the upper profile. User profile on top is provided by domain experts modeled as Definition 3, which is used to calculate application profile. The occurrence probability of an application scenario is

$$stg.P = \sum_{usage[i].app[j].name=stg.name} (usage[i].P) * (usage[i].app[j].P) \tag{1}$$

An application scenario depicted with a STG is composed of several case scenarios. The transitions between case scenarios are totally dependent on users without respect to objective factors. Therefore, the knowledge of domain experts is applicable to decompose an application scenario into several case scenarios according to UML UCD provided by software designers, and determine transition probabilities. Application profile can be propagated across STG to construct case profile. From a long-term point of view, an application scenario is called repeatedly, which can be regarded as an infinite cyclic process. Thereby, a transition from termination case to start case with probability 1 can be added to a STG, which is then modeled as an irreducible discrete time Markov chain. With that, the occurrence probabilities of cases are expressed as

$$\begin{cases} \begin{pmatrix} stg.case[1].P \\ \vdots \\ stg.case[n].P \end{pmatrix} \\ = (stg.case[1].P \quad \cdots \quad stg.case[n].P) \cdot \\ \begin{pmatrix} stg.trans[1,1] & \cdots & stg.trans[1,n] \\ \vdots & \vdots & \vdots \\ 1 & \cdots & stg.trans[n,n] \end{pmatrix} \\ \text{where } \sum_{i=1}^{n} stg.case[i].P = 1 \end{cases} \tag{2}$$

Case scenarios are depicted with CIGs, which are used to propagate case profile to construct operational profile. Operational profile describes practical input profile to components and the way how components call others. In order to ensure objectivity and adequacy of the profiles which directly determines the accuracy of reliability evaluation, we have proposed an algorithm to automatically obtain operational profile from case profile in the previous work [19]. The algorithm traverses a CIG to propagate case profile to all the components.

Component developers typically perform testing in terms of software basic functional units, whose results are failure probability or reliability on each sub-domain. In order to consider the difference between operational and testing profile, operational profile should be mapped to sub-domains to obtain a group of weight—the probability that a component is called on each sub-domain. Sub-domain weight can be calculated with joint probability distribution of all parameter inputs. Given that input parameters are independent, the probability that an input belongs to a sub-domain is

$$\begin{aligned} &P(service.profile[i] \in service.sd[j]) \\ &= P\left(\bigcap_{k=1}^{m} (service.param[k] = service.profile[i].item[k])\right) \\ &= \prod_{k=1}^{m} P(service.param[k] = service.profile[i].item[k]) \end{aligned} \tag{3}$$

where $P()$ is a probability function.

A sub-domain weight is given by

$$service.sd[i].weight = \sum P(service.profile[j] \in service.sd[i]) \tag{4}$$

4.2 Reliability Analysis

Based on the obtained SRM, we will carry out a scenario-based software reliability analysis in this subsection. On the contrary to the profile propagation from top to down, reliability analysis takes a bottom-up process. Component reliability is estimated at first, then the reliability of case and application scenarios, and software reliability at last.

A case scenario corresponds to a system execution path, which can be represented as a linear sequence of call actions $\{call[1], \cdots, call[n]\}$. Therefore, case reliability is

$$case.R = \prod_{action[i] \in case} call[i].R \tag{5}$$

Call reliability depends on the reliability of called components and the way how to make the calls. Components' practical reliability is different from their testing reliability because of different profile. The mapping between two kinds of profile has been realized, the results of which are sub-domain weights as (4). Based on that, the mapping from testing to practical reliability can also be achieved.

$$service.R = \sum_{i} (service.sd[i].R) * (service.sd[i].weight) \tag{6}$$

With regard to the way how to make calls, we take three

```
1  Input: cig   //scenario model
2  Output: case.R //scenario reliability
3  double   getCR(CIG cig)
4  { case.R=1; Action *action=cig.arch->first; double temp=0;
5  while (action->next!=null)
6  {   switch(action->type)
7     {   case call: temp=0;//sequential call
8  foreach (Sdomain sd in action->called->sd) temp+=sd.R *sd.weight;
9        if (action.child[0].link->first->owner->type== atomic)
10          case.R= case.R*temp;
11       else   case.R= case.R*temp* getCR(action->child[0]); break;
12       case loop: double   r=getCR(action->child[0]); temp=0;
13       for (int i=0; i<((action->ap).account);i++)
14          temp+=action->ap[i].P * (r)^(action->ap[i].num);   case.R
   *=temp; break;
15       case branch: temp=0;
16       for (int i=0; i<((action->ap).account);i++)
17          temp+=action->ap[i].P * getCR(action->child[i]);
   case.R*=temp; break;
18       } action=action->next;
19    } return case.R;
20 }
```

Fig. 6 An algorithm to compute the reliability of a case scenario.

ways into account, namely, sequential, branch, and loop calls with a nested semantics. Sequential calls are not nested by other actions. Call reliability with sequential structure is equal to practical reliability of called components.

$$call.R = call.child.R \tag{7}$$

A branch structure nests a finite number of sub-actions which may be or not be executed in a case scenario. Branch profile is usually represented as branch transition probabilities. Thus, call reliability with branch structure is

$$call.R = \sum_i (call.ap[i].P) * (call.child[i].R) \tag{8}$$

A loop structure has only one nested action. The loop contains a specification of loop iteration counts as a random variable over s finite domain of iteration counts, each assigned a probability of its occurrence. For loop reliability, we have

$$call.R = \sum_i (call.ap[i].P) * (call.child.R)^{call.ap[i].num} \tag{9}$$

Based on the above analysis, an algorithm is proposed to compute the reliability of a case scenario, as shown in Fig. 6.

It can be seen from Fig. 6 that it is a recursive procedure to compute the reliability of a case scenario. Using case reliability, the reliability of application scenarios can be expressed as

$$stg.R = \sum_i (stg.case[i].R) * (stg.case[i].P) \tag{10}$$

Finally, software reliability is given by

$$srm.R = \sum_i (srm.stg[i].R) * (srm.stg[i].P) \tag{11}$$

In the rest of this subsection, we will discuss how well software deployment influences software reliability in the light of CDM, including hardware configuration and deployment scheme. Hardware resources are modeled with the properties of MTTF and MTTR, whose reliability is

$$hw.R = \frac{hw.MTTF}{hw.MTTF + hw.MTTR} \tag{12}$$

We construct physical state space as $S = \{case[1].s, \cdots, case[n].s\}$ in terms of case scenarios, where $case[i].s \in S$ represents the state that all hardware required by the ith case scenario is normal. Since sub-domains are the basic functional units to run a component, they are also adopted to indicate required hardware. The probability that the hardware required by a sub-domain is normal is

$$sd.s = \prod_{server.hw[i].type=sd.hardtype[i]} (server.hw[i].R) \tag{13}$$

For some key components, it may use more than one server to deploy them in order to guarantee their normal execution. In this case, hardware state probability on a sub-domain is

$$sd.s = 1 - \prod_i \left(1 - \prod_{server[i].hw[j].type=sd.hardtype[j]} server[i].hw[j].R\right) \tag{14}$$

The probability that the hardware required by a case scenario is normal is

$$case.s = \prod_{service[i].sd[j] \in case} service[i].sd[j].s \tag{15}$$

Therefore, with hardware reliability considered case reliability is refined as

$$case.R = case.S * \left(\prod_{action[i] \in case} action[i].R\right) \tag{16}$$

5. Case Study Evaluation

In this section SRM-based software reliability analysis approach is applied to evaluate the reliability of a distributed component-based software system [24]. Evaluation result is compared with the results in [11] to demonstrate the prediction capabilities of our approach.

5.1 SRM Construction

Figure 7 illustrates a high-level view on the Business Reporting System (BRS), which generates management reports from business data collected in a database. The model is based on an industrial system. For the purpose of BRS reliability analysis, we need to construct a SRM for BRS at first, including deployment model and profile model.

The BRS system consists of 23 software components

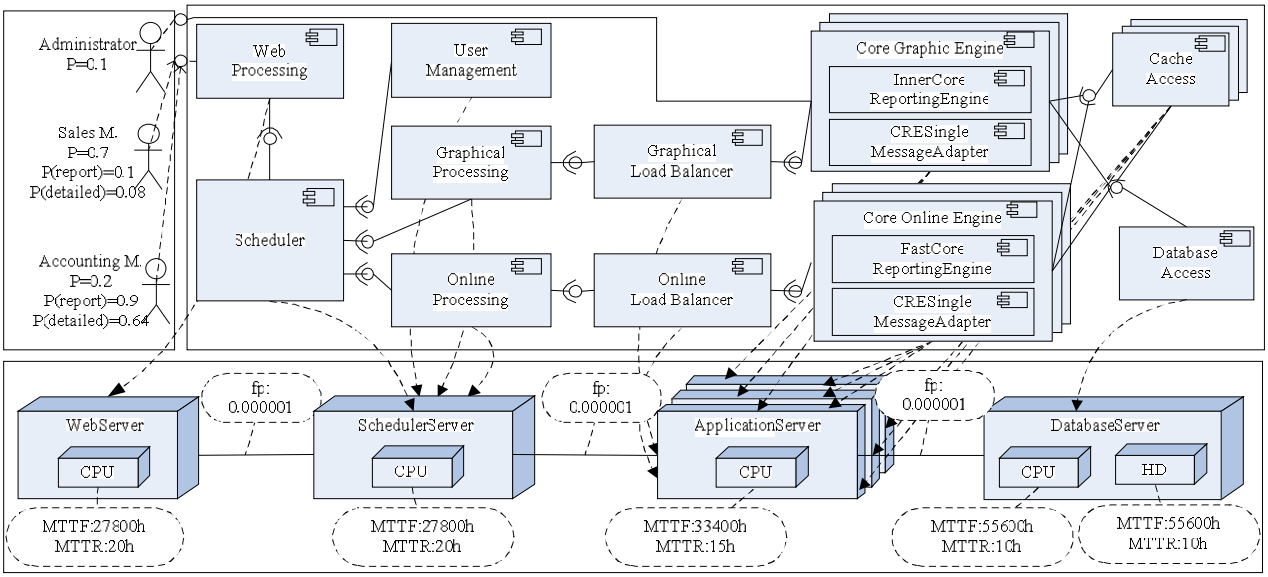

Fig. 7 An overview of a business report system.

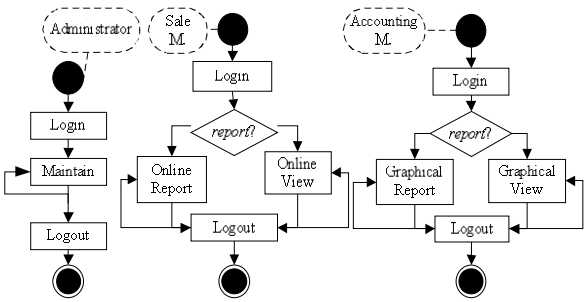

Fig. 8 Application profile of the business reporting system.

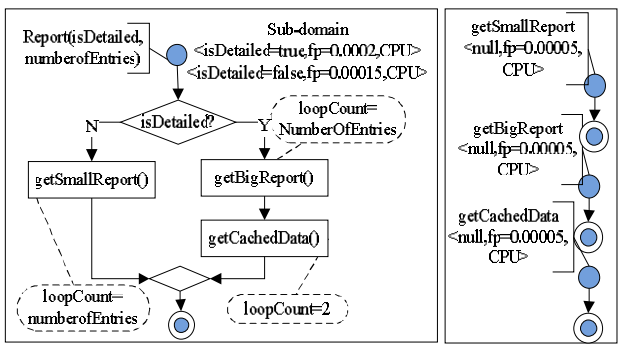

Fig. 9 A case profile instance of the business reporting system.

deployed on six servers. The web server propagates user requests to a scheduler server, which hosts, among others, a scheduler and a user management component. From there, requests reach the main application server and are possibly dispatched to 2 further application servers by two load balancer components. A database server hosts the database and a corresponding data access component. The system includes caches to reduce the need of database accesses.

BRS includes three types of users with three usage scenarios: Sales managers use the system mainly for online viewing of live data, accounting managers request the production of graphical reports, and administrators perform system maintenance activities. Each usage scenario includes only one application scenario as shown in Fig. 8.

It can be seen from Fig. 8 that there are 7 basic case scenarios: Login, Logout, Maintain, Online Report, Online View, Graphical Report, and Graphical View. Next, CIGs for case scenarios need to be constructed. Figure 9 shows a CIG model for Online Report. Due to limited space, we don't list CIG models for other cases. The details of BRS are described at the website [25].

5.2 Reliability Analysis

The upper-left part of Fig. 7 shows that the occurrence probability of three usage scenarios is 0.1, 0.7, and 0.2 respectively. How an application system is used is often recorded in the system log. The scenario probabilities can be easily obtained by means of statistical analysis of the log. Since each usage scenario includes only one application scenario, their occurrence probability is equal to that of corresponding usage scenario. According to application profile model as shown in Fig. 8, Eq. (2) is used to calculate the occurrence probability of case scenarios. The results are shown in the first three rows of Table 1.

The algorithm in Fig. 6 is used to compute case reliability without regard to hardware state. The results are shown in the last but one row of Table 1 as failure probability for ease of viewing. At the same time, Eq. (15) is used to calculate case state probability, and results are show in the last row of Table 1.

Based on the above analytical results about case scenarios, Eq. (13) is used to calculate the reliability of three

Table 1 Occurrence probabilities of scenarios

Scenarios	Admin.	Sales	Account	Failure	State
Login	0.1	0.01	0.01	e-5	0.9993
Logout	0.1	0.01	0.01	e-5	0.9993
Maintain	0.8	0	0	e-5	1-9e-11
O.Report	0	0.098	0	5.28e-5	0.99964
O.View	0	0.882	0	1.56e-5	1-9e-11
G..Report	0	0	0.882	6.14e-5	0.99964
G..View	0	0	0.098	1.56e-5	1-9e-11

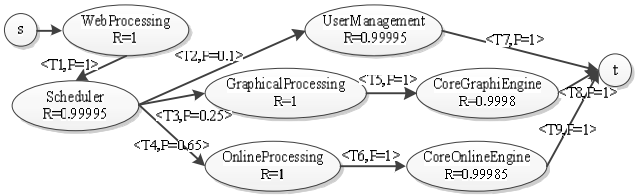

Fig. 10 Component dependency graph of business reporting system.

Table 2 Reliability estimation of BRS system.

Approach	Reliability
SRM-based approach	0.999634
PCM-based approach	0.999652
CDG-based approach	0.999798

application scenarios as 0.99979, 0.99976, and 0.99911. Finally, software reliability can be obtained with Eq. (14) as 0.999634.

In order to prove the efficiency of our SRM-based approach, we compare the analytical result with that of PCM-based [12] and CDG-based [9] approach. All three of them employ scenario-based reliability analysis. Compared with the first two ones, CDG-based method is more coarse-grained, which doesn't consider the parameter propagation between component interfaces and hardware reliability. The CDG for BRS system is shown in Fig. 10.

The reliability evaluation results are shown in Table 2. It can be seen that the result of SRM-based approach is close to that of PCM-based one, and the deviation is less than 0.002%. Since PCM method has been published and proved, it can also suggest the efficiency of our approach. In contrast, the result of CDG-based approach seems to be over-optimistic. There are two reasons. At first, it doesn't consider hardware reliability. Secondly, it mixes all scenarios together to describe component reliability which in fact is different at different scenarios.

Then, we will further compare time complexity of our approach with PCM-based approach in [12]. Let the number of nodes in software architecture model be N and the number hardware resources required by a software system be M. PCM-based approach considers all possible cases of hardware availability. As each resource has two possible states, the size of physical state space is 2^M. PCM-based approach adopts flat construction of scenarios, and the time complexity is $O(2^M \cdot N)$. SRM-based approach adopts hierarchical scenario specification, and the size of physical state space is equal to the number of case scenarios which is assumed to be L. SRM approach performs scenario reliability analysis at each physical state, and the time complexity is $O(M \cdot N \cdot L)$. Generally, the number of case scenarios meets $L < N$. Thus, it can be seen that SRM-based approach improves the efficiency of software reliability analysis compared to exponential time complexity of PCM-based approach for a large component-based software application.

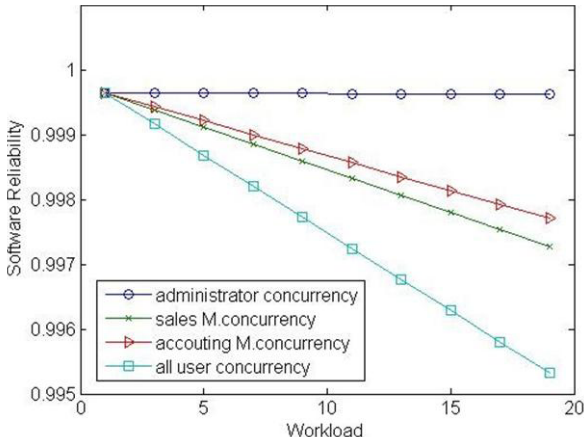

Fig. 11 Sensitivity of software reliability to concurrency.

Component-based software applications are usually distributed Web applications, which can be used simultaneously by multiple users. The number of concurrent users is referred to as workload which considerably affects software performance and reliability. In the future work, we will study in detail how workload influences software reliability. Figure 11 shows how BRS reliability varies with different workload. It can be seen that software reliability gradually decreases as workload increases. Workload is one of the most important factors that should be considered when designing the architecture of component-based software. Key components which significantly affect system reliability should be deployed on multiple or high-performance servers to guarantee high system reliability according to practical workload.

6. Conclusion

In this paper, we have presented a framework to quantitatively assess software reliability using hierarchical scenario specification, thus applicable to early phases of the software life cycle. Our main contribution lies on a reliability analysis approach for component-based software that takes into account the dependencies between different kinds of profile that various participators concerns and the difference between testing and operational profile. Based on the profile dependencies, the approach uses profile propagation to construct operational profile, and adopts the concept of subdomain to realize the mapping from testing to operational profile. With all of that, reliability analysis based on hierarchical scenarios is performed to avoid state space explosion, and at the same time hierarchical scenarios conform to software design specifications. In summary, we can conclude

that the proposed approach can improve the scalability and objectivity of the current scenario-based reliability prediction approaches.

Acknowledgments

This research was supported by the National Natural Science Foundation of China under Grand No.61402333, No. 61402242, and No. 61272450, and Tianjin Municipal Science and Technology Commission under grand No. 15JCQNJC00400. The authors would also like to thank Tianjin Key Lab of Intelligence Computing and Novel Software Technology and Key Laboratory of Computer Vision and System, Ministry of Education, for their support of the work.

References

[1] K. Goševa-Popstojanova and K.S. Trivedi, "Architecture-based approach to reliability assessment of software systems," Performance Evaluation, vol.45, no.2, pp.179–204, 2001.
[2] S.S. Gokhale, "Architecture-based software reliability analysis: Overview and limitations," IEEE Trans. Dependable and Secure Comput., vol.4, no.1, pp.32–40, 2007.
[3] V.S. Sharma and K.S. Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," Journal of Systems and Software, vol.80, no.4, pp.493–509, 2007.
[4] K. Goseva-Popstojanova, A. Hassan, A. Guedem, W. Abdelmoez, D.E.M. Nassar, H. Ammar, and A. Mili, "Architectural-level risk analysis using UML," IEEE Trans. Softw. Eng., vol.29, no.10, pp.946–960, 2003.
[5] S. Anwar, M. Ramzan, A. Rauf, and A.A. Shahid, "Software Maintenance Prediction Using Weighted Scenarios: An Architecture Perspective," 2010 International Conference on Information Science and Applications (ICISA), Seoul, South Korea, pp.1–9, 2010.
[6] A. Ali, D.N.A. Jawawi, and M.A. Isa, "Modeling and calculation of scenarios reliability in component-based software systems," 8th Software Engineering Conference (MySEC), Malaysian, pp.160–165, 2014.
[7] H. Mei, G. Huang, L. Zhang, and W. Zhang, "ABC: a method of software architecture modeling in the whole lifecycle," SCIENCE CHINA Information Sciences, vol.44, no.5, pp.564–587, 2014.
[8] M. Palviainen, A. Evesti, and E. Ovaska, "The reliability estimation, prediction and measuring of component-based software," Journal of Systems and Software, vol.84, no.6, pp.1054–1070, 2011.
[9] S. Yacoub, B. Cukic, and H.H. Ammar, "A scenario-based reliability analysis approach for component-based software," IEEE Trans. Rel., vol.53, no.4, pp.465–480, 2004.
[10] R.H. Reussner, H.W. Schmidt, and I.H. Poernomo, "Reliability prediction for component-based software architectures," Journal of Systems and Software, vol.66, no.3, pp.241–252, 2003.
[11] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the palladio component model," IEEE Trans. Softw. Eng., vol.38, no.6, pp.1319–1339, 2012.
[12] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," In: Research into Practice–Reality and Gaps, Springer, pp.36–51, 2010.
[13] G.I.N. Rodrigues, D. Rosenblum, and S. Uchitel, "Using scenarios to predict the reliability of concurrent component-based software systems," Fundamental Approaches to Software Engineering, pp.111–126, Springer, Edinburgh, 2005.
[14] R. Roshandel, N. Medvidovic, and L. Golubchik, "A Bayesian model for predicting reliability of software systems at the architectural level," Software Architectures, Components, and Applications, pp.108–126, 2007.
[15] H. Singh, V. Cortellessa, B. Cukic, E. Gunel, and V. Bharadwaj, "A bayesian approach to reliability prediction and assessment of component based systems," Proc. 12th International Symposium on Software Reliability Engineering (ISSRE), Hong Kong, pp.12–21, 2001.
[16] Y. Liu, P. Lin, Y.-F. Li, and H.-Z. Huang, "Bayesian Reliability and Performance Assessment for Multi-State Systems," IEEE Trans. Rel., pp.1–16, 2014.
[17] I. Krka, G. Edwards, L. Cheung, L. Golubchik, and N. Medvidovic, "A comprehensive exploration of challenges in architecture-based reliability estimation," Architecting Dependable Systems VI, vol.5835, pp.202–227, 2009.
[18] C.Y. Hou, C. Chen, J.S. Wang, and K. Shi, "A scenario-based reliability analysis approach for component-based software," IEICE Trans. Inf. & Syst., vol.E98-D, no.3, pp.617–626, 2015.
[19] L. Cheung, I. Krka, L. Golubchik, and N. Medvidovic, "Architecture-level reliability prediction of concurrent systems," Proc. 3rd ACM/SPEC International Conference on Performance Engineering (ICPE), Boston, pp.121–132, 2012.
[20] Q. Ao, J. Ai, M. Lu, and F. Zhong, "Scenario-based software operational profile," 9th International Conference on Reliability, Maintainability and Safety (ICRMS), Guiyang, pp.700–704, 2011.
[21] A. Immonen and E. Niemelä, "Survey of reliability and availability prediction methods from the viewpoint of software architecture," Software and Systems Modeling, vol.7, pp.49–65, 2008.
[22] A. Amin, L. Grunske, and A. Colman, "An approach to software reliability prediction based on time series modeling," Journal of Systems and Software, vol.86, no.7, pp.1923–1932, 2013.
[23] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," ACM Transactions on Software Engineering and Methodology (TOSEM), vol.18, no.3, pp.12–52, 2009.
[24] X. Wu and M. Woodside, "Performance modeling from software components," Proc. Fourth Int'l Workshop Software and Performance, vol.29, pp.290–301, 2004.
[25] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Reliability prediction for component-based software architectures," Online Available: http://sdqweb.ipd.kit.edu/wiki/ReliabilityPrediction, 2011.

Chunyan Hou received the master's degree in computer science from Beihang University in 2006, and the PhD degree in computer science from Harbin Institute of Technology in 2011. Currently, she is working as a lecturer in school of computer and communication engineering, Tianjin University of Technology. Her main research interests include software reliability evaluation and software testing.

Jinsong Wang received the PhD degree in computer science and technology from Nankai University in 2005. Currently, he is working as a professor in school of computer and communication engineering, Tianjin University of Technology. His main research interests include computer network and information security.

Chen Chen received the PhD degree in computer science and technology from Harbin Institute of Technology in 2011. Currently, he is working as a lecturer in the College of Computer and Control Engineering, Nankai University. His main research interests include information retrieval and natural language processing.