PAPER

A Scenario-Based Reliability Analysis Approach for Component-Based Software

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

SUMMARY With the rise of component-based software development, its reliability has attracted much attention from both academic and industry communities. Component-based software development focuses on architecture design, and thus it is important for reliability analysis to emphasize software architecture. Existing approaches to architecture-based software reliability analysis don't model the usage profile explicitly, and they ignore the difference between the testing profile and the practical profile of components, which limits their applicability and accuracy. In response to these issues, a new reliability modeling and prediction approach is introduced. The approach considers reliability-related architecture factors by explicitly modeling the system usage profile, and transforms the testing profile into the practical usage profile of components by representing the profile with input sub-domains. Finally, the evaluation experiment shows the potential of the approach.

key words: *software reliability, software architecture, scenario, component, profile*

1. Introduction

With the advance of social information processes, computer and software products have been widely used in various industries, especially in safety-critical fields. Software reliability has drawn wide concern. The reliability of a software system is defined as the probability of failure-free operation 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.

In recent years, the rise of component-based software development has changed the nature of software industry. It's becoming an important development pattern for future software to share resources and collaborate with each other in the globally distributed community. Software industry has adopted this more productive and flexible development approach instead of coding from scratch, where software is generated with existing open-source, commercial and proprietary components by assembling them together in an interoperable manner. Thus, software engineering is now more focused on architecture design, component selection and system integration tasks instead of coding. Software architecture is becoming a key factor to measure a software system in component-based software development. As a result, traditional black-box software reliability analysis approaches based on software testing is no longer suitable for a component-based software application. As software development shifts the emphasis to architecture design, architecture-based software reliability analysis attracted wide concern in the community. In contrast to black-box approaches, architecture-based approaches are applicable for reliability analysis at any phase during the software lifecycle, and it is especially necessary at early design phase, which helps software architect to evaluate various architecture designs quantitatively and make a choice among them. Thus, architecture-based software reliability analysis is useful to optimize software architecture design, avoid costly design change, and improve software development process and reliability.

Research on architecture-based software reliability analysis is still in its infancy. There are some problems for the existing approaches needed to be solved. In response to these problems, a new scenario-based reliability analysis approach for component-based software is proposed in this paper. The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes a component-based software architecture model (CSA) in a two-layer structure. Section 4 explains how to predict software reliability based on CSA, which includes solving practical profile of components in the light of system usage profile, and solving a scenario-based model. Section 5 documents the case study before Sect. 6 concludes the paper.

2. Related Work

In the past few decades, software reliability analysis approaches have been developed a lot, most of which are software reliability growth models (SRGMs) [1], which are proposed according to traditional software development. SRGMs take software as a whole and use system failure dataset collected during testing and operational phases to model software reliability growth procedure. SRGMs are a kind of black-box models which take the interactions between software with external environment into account only without considering its internal architecture. Software architecture is crucial for a component-based software system. Thus, SRGMs are not suitable for component-based software reliability analysis.

The existing architecture-based approaches for component-based software reliability analysis can be classified into two broad categories, namely a state-based and path-based

Manuscript received July 11, 2014.
Manuscript revised October 7, 2014.
Manuscript publicized December 4, 2014.

[†]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.2014EDP7241

Copyright © 2015 The Institute of Electronics, Information and Communication Engineers

model. The former maps a software architecture model into a Markov state space model, and analytically combines software architecture with component failure behaviors to predict software reliability [2]. The latter is also called a scenario-based model. Scenario has been widely used to describe the way how a software system reacts to a request. Scenario is a set of component interactions triggered by a specific input stimulus, also called a system execution path. A scenario-based approach models software architecture as all possible paths in a software system and their execution probabilities. The approach solves path reliability, and then averages them with their probabilities as weights to obtain system reliability.

The existing architecture-based approaches mainly suffer from the following shortcomings.

(1) It's inconvenient for software developers to use them since they use some kind of analysis-oriented mathematical models to model software architecture. To overcome this problem, some of the approaches adopted design-oriented high-level notations [4], [11]. They model software architecture based on UML sequence and deployment diagrams annotated with reliability properties such as failure probability. This kind of architecture models need to be transformed into analytical models before they can be solved to obtain system reliability. The aim of these approaches is that software developers can quickly enhance existing design specifications in UML to construct reliability predictions regardless of the complication of underlying analysis technology. However, most of them only choose some aspects of software architecture to build architecture models. For example, the scenario-based approaches only model the interactions between components without considering other factors such as execution environment, deploy information and so on [11]. That hinders effective software reliability analysis at architecture design phase.

(2) They don't model software usage profile explicitly. They usually represent usage profile implicitly with transition probabilities between states or scenarios in a system model, which are dependent on testing dataset [9] or software developers' intuition [10]. However, the former is not available at early design stage of a software system, and the latter is apt to make reliability estimation less objective. Afterwards, a parameter dependency method was proposed to propagate system-level usage profile to all the components in a software system. System inputs influence system control flow, and determine practical profile of the components in a system. Hamlet et al. [5] allow component developers to specify the call propagation of individual components. However, the dependency of the call propagations to input parameters values is not made explicit. Reussner et al. [7] explicitly models the influence of external components. However, the approach assumes fixed transition probabilities between components; therefore its models cannot be reused if the system-level usage profile changes. Brosch et al. [2] explicitly model system-level usage profile and the interactions between components to solve parameter dependencies which are used to propagate system profile to all components in a system. However, the approach specifies component failure behavior in terms of program internal actions, which don't conform to software testing specification.

(3) They directly take unit testing results of components as their practical reliability without considering the difference between testing and operational profile. Component developers and users are separate during component-based software development. The developers have no idea how the component will be used in the future when testing it. Thereby, components' testing profile is different from operational profile, and need be converted into operational profile before testing results are used to estimate components' practical reliability. To solve this problem, Hemlet et al. [5] proposed a sub-domain concept. Software input space can be naturally divided into several functional sub-domains, each of which has their own operational profile in uniform distribution. Component practical profile can be represented as a weight vector of sub-domains, which allows software developers to test components without knowledge of their usage profile. However, the approach doesn't apply the concept of sub-domain for component-based software reliability analysis.

(4) They don't consider the influence of execution environment on the reliability of a software system. Lipton et al. [6] and Yacoub et al. [11] take failure probabilities of network connections into account, but neglect the availability of other hardware device, such as processors. Sato et al. [8] combine a system model with a resource availability model. However, they do not consider application-level software failures. Brosch et al. [3] consider availability of network links and hardware devices, and generate Markov chains for all possible cases of hardware resource availability. However, the approach doesn't consider potential state space explosion.

In response to above problems, we propose a scenario-based approach for component-based software reliability analysis. Nowadays, scenario-based approaches have received increasing attentions since scenario concept conforms to software testing specification, where a scenario corresponds to a test case. To the first problem, a two-layer model is suggested to describe component-based software architecture, where the first layer locates a UML-based model for the sake of software developers and the second layer is a formal analytical model transformed from the first-layer model. The second-layer model is invisible for software developers, which is potentially analyzed to obtain software reliability. To the second problem, an algorithm is proposed, which propagates user input profile at system level to build practical profile of the components in a system. To the third problem, sub-domain concept is adopted to map testing profile into operational profile of components. A scenario-based approach has an advantage to overcome the last problem. A component-based software application generally has a finite number of execution paths. A scenario-based approach builds physical state space in the light of system execution paths. In comparison to traditional approaches who consider all possible combinations of hard-

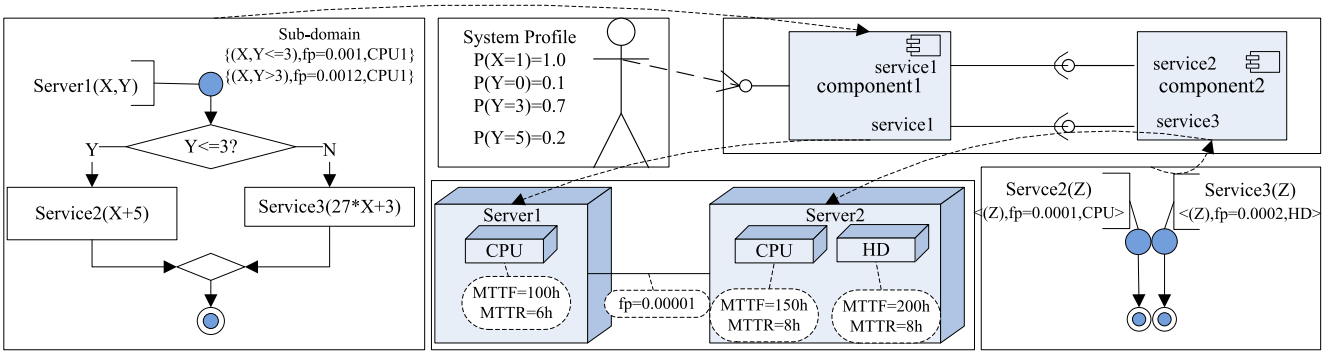

Fig. 1 An example for architecture model.

ware resource states, a scenario-based approach can effectively avoid state space explosion and improve the efficiency of software reliability analysis.

3. Component-Based Software Architecture Model

In this section, we will introduce a two-layer component-based software architecture model called CSA. The first layer of a CSA model situates a UML-based architecture model while the second layer is a formal model. First of all, a simple example is employed to illustrate the first-layer model orienting software developers. And then, we will introduce the second-layer model of a CSA model, which is invisible for software developers, and the target for reliability analysis algorithms.

3.1 UML-Based Architecture Model

Figure 1 shows a simple example of a UML-based architecture model of component-based software. As shown in Fig. 1, the model is composed of three elements: (1) component reliability models; (2) system-level usage profile; (3) execution environment.

Component reliability models are the primary elements of a CSA model, and also crucial for the reliability of a component-based software application. They are built with reliability-related information offered by component developers, including basic properties, testing profile and the interactions between components. Basic properties are made up of interface description, input space, deployment location and so on. Sub-domain concept is adopted to represent testing profile, where unit testing of a component is a procedure that component developers test it in its input sub-domains in a specific execution environment. Testing results are the probabilities that failure occurs in sub-domains whose required hardware is also modeled abstractly in order to take the influence of execution environment failure on software reliability into account. The interactions between components are depicted with call actions and program structures nesting calls based on UML sequence and activity diagrams. Call actions describe parameter dependencies between components, which are useful to pass practical profile of a component to its called component. The structures nesting calls including sequence, loop, branch, fork and so on, explain the way that a component uses other components, and indicate possible direction of control flow in a program.

Control flow direction in a system depends on system-level usage profile. As software reliability theory shows, software reliability is dependent on their usage profile. Thus, a precise and objective description of usage profile is very important for the accuracy of software reliability prediction. Domain experts specify the usage model, which involves the number and order of calls to component interfaces at the system boundaries. The model can contain control flow constructs such as branches, loops. For each called interface, the domain experts also characterize its input parameter values and specific probabilities taking different values. Once the usage model is connected to the system model by the software architect, tools can propagate the parameter values through the parameterized expressions specified by component developers. Because of the parameterization, the usage model can easily be changed at the system boundaries and the effect on the component specifications can be recalculated.

Execution environment characterizes the configuration of the servers used to deploy components, and the links between servers. Communication link failures include loss or damage of message during transport, which results in service failure. Though transport protocols like TCP include mechanisms for fault tolerance, failures can still occur due to overload, physical damage of the transmission link, or other reasons. As such failures are generally unpredictable from the point of view of the system deployer, we treat them like software failures and annotate communication links with a failure probability. System developers can define these failure probabilities either from experience with similar systems or by running tests on the target network. Servers are made up of hardware resources for software execution, e.g. CPU, harddisk, memory and so on. Each kind of hardware has inherent properties, such as processing rate, scheduling strategy and so on. Unavailable hardware causes a service execution to fail. Hardware resource breakdowns mainly result from wear out effects. Typically, a broken-down resource is eventually repaired or replaced by a functionally equivalent new resource. In CSA, hardware resources are annotated with Mean Time To Failure

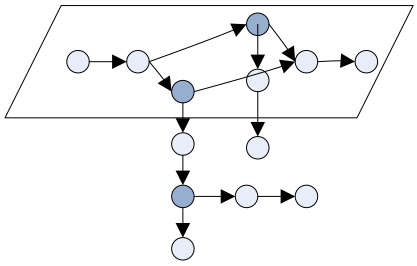

Fig. 2 CSA formal model.

(MTTF) and Mean Time To Repair (MTTR). System deployers have to specify these values. Hardwar vendors often provide MTTF values in specification documents. System deployers can refine these values on experience. MTTR values can depend on hardware support contracts. Hardware reliability models are generated with their properties related to software reliability. When a component is deployed on a specific server, reliability models of hardware in the server can be mapped directly to abstract hardware models in component sub-domains.

3.2 Formal Architecture Model

According to the UML-based CSA model, we adopt a multi-layer directed graph to represent the formal CSA model, as shown in Fig. 2. Nodes in Fig. 2 are composed of solid and hollow nodes which denote call actions and the actions nesting calls respectively. The former represents a call to an interface, which launches a directed graph for the reliability model of called interface on the next layer. The latter represents program structures nesting calls, such as loop, branch and so on. We will define the formal CSA model in detail as follows.

Definition 1. (CSA) A CSA is a component-based software architecture model and defined by the tuple ⟨Server *server*[], Interface *inf*[], Usage *usage*[], Interface **first*⟩, where *server* is a set of servers in a software system; *inf* is a set of component interfaces used to constitute a software application; *usage* is a system usage profile composed of the usage profiles belonging to different kinds of users; and pointer *first* points to start node of user interface in a software system.

Definition 2. (Server) A server is defined by the tuple ⟨string *name*, Hardware *hw*[], Link *link*[]⟩, where *name* is the name of a server; *hw* is a set of hardware resources in a server; and *link* is a set of communication links connecting to a server.

Definition 3. (Hardware) Hardware models a kind of hardware resources in a server. It is annotated by the tuple ⟨int *type*, int *MTTF*, int *MTTR*, long *fp*⟩, where *type* is the type of hardware, for example, CPU, hardisk, memory and so on; *MTTF* is mean time to failure; *MTTR* is mean time to repair; and *fp* is hardware failure probability.

Definition 4. (Link) A link models a communication link, and is defined as the tuple ⟨string *name*, long *fp*⟩, where *name* is the name of a link, and *fp* is link failure probability.

Definition 5. (Interface) An *Interface* is a service provided by a component. It is annotated by the tuple ⟨string *name*, Component *cp*, Input *input*[], Sdomain *sd*[], Action **op*, long *R*⟩, where *name* is the name of a interface; *component* is the component offering a interface; *Input* describes interface input space; *Sd* is a set of sub-domains constituting interface input space; *op* is a pointer directing to program start node; and *R* is practical reliability of a interface.

Definition 6. (Component) A component is defined as the tuple ⟨string *name*, Server **server*⟩, where *name* is component name, and *server* points to the server where a component runs.

Definition 7. (Input) Input describes the input to a parameter, and is defined as the tuple ⟨string *name*, int *type*, Profile *profile*[]⟩, where *name* is parameter name; *type* is parameter type; and *profile* is parameter input profile.

Definition 8. (Profile) A profile describes a case of parameter input, and is defined by the tuple ⟨short *value*, long *P*, Sdomain **sd*[]⟩, where *value* is discrete input value; *P* is the probability taking the value; and *sd* points to the sub-domain that a profile belongs to.

Definition 9. (Sdomain) A Sdomain models one of the sub-domains constituting input space of a component interface. It is annotated by the tuple ⟨string *exp*, short *weight*, int *hardtype*[], long *fp*, long *R*, Hardware **hw*[]⟩, where *exp* is a Boolean expression with interface input parameters as variables used to define sub-domain input space; *weight* is the probability that a interface input falls into a sub-domain, which is dependent on component practical profile; *hardtype* is a set of abstract hardware models to indicate what kinds of hardware are required by a sub-domain; *fp* is the probability that failure occurs in a sub-domain, which is dependent on component testing profile; *R* is practical reliability of a interface in a sub-domain, and determined by its practical profile; and *hw* points to concrete hardware resources, which is used to realize mapping from abstract to concrete hardware model in a sub-domain.

Definition 10. (Action) An action is a program internal action and defined by the tuple ⟨int *type*, string *exp*, Input *input*[], Action **child*, Action **next*⟩, where *type* is action type including call, branch, loop and so on; *exp* is a arithmetic or Boolean expression to describe parameter dependencies; *input* is the interface input profile after taking a action, which is used to record how a action influences input profile and pass input profile to the next action; *child* points to sub-actions nested by a action; and *next* points to the next action.

Definition 11. (Call) Call is a subclass of Action, which models the interactions between two interfaces. It is annotated by the tuple ⟨Interface **called*, long *rlink*⟩, where *called* points to called interface, and *rlink* is the reliability of communication link between two components.

Definition 12. (Branch) Branch is a subclass of Action, which models a branch structure in an interface program. It is defined by the tuple ⟨string *exp*[], Action **child*[], short *P*[]⟩, where *exp* and *child* corresponding to the same properties in *Action* are redefined as arrays because a branch

```
1   Input: CSA csa //component-based software architecture model
2   void  profile(CSA csa)
3   {  csa.first->input=initProfile(csa.usage);csa.first->op->input=csa.first->input; // user interface profile
4      Stack s; s.push(csa.first->op);
5      While (!s.empty()) do //execute depth-first traversal in csa
6      {  Action *p=s.top();
7         switch (p->type)
8         {  case 0: p->next->input=p->input; s.push(p->next); break;   // program start
9            case 1: p->called->input=passProfile(p->type, p->exp, p->input); // call start
10              p->called->op->input=p->called->input; s.push( p->called->op); break;
11           case 2: for int i=0 to count(p->child) do // branch start
12              if (p->child[i] isn't visited)
13              { p->child[i]->input=passProfile(p->type, p->exp[i], p ->input); s.push(p->child[i]); break; }
14           case 3: p->profile=passProfile(p->type, p->exp, p->input); //loop start
15              p->child->input=p->input; s.push(p->child); break;
16           case 4: s.pop(); p->next->input=s.top()->input; s.pop(); s.push(p->next); break; // call end
17           case 5:p->next->input=p->input; s.pop(); s.pop(); s.push(p->next); break; // loop end
18           case 6: s.pop();//branch end
19              if (all p->child are visited) { p->next->input=s.top()->input; s.pop(); s.push(p->next);} break;
20           case 7: s.pop(); s.pop(); s.push(p->next); //program end
21  }}}
```

Fig. 3 Profile algorithm.

structure has a finite set of nested behaviors; and*P* is a set of branch execution probabilities, namely branch profile.

Definition 13. (Loop) Loop is a subclass of Action, which models a loop structure in an interface program. It is defined by ⟨LoopProfile *profile*[]⟩, where *profile* is a loop profile.

Definition 14. (LoopProfile) A *LoopProfile* models the cases of loop execution, and is defined by the tuple ⟨int *count*, long *P*⟩, where *count* is loop iteration counts, and *P* is the probability that iteration counts occur.

Definition 15. (Usage) Usage is a usage profile and models how a kind of users inputs at the boundaries of a system, and is defined by the tuple ⟨short *P*, Input *input*[]⟩, where *P* is the probability that this kind of users use the system; and *input* describes their input profile.

4. A Scenario-Based Reliability Analysis

Based on CSA model built in the last section, we will propose a profile algorithm in this section, which parsers parameter dependencies between components to build practical profile of the components according to system-level usage profile in a component-based software application. And then a scenario-based approach is applied to analyze CSA model for component-based software reliability.

4.1 Profile Algorithm

We use C++ like language to implement profile algorithm, as shown in Fig. 3.

The algorithm propagates system-level input profile through multi-layered directed CSA model by traversing nodes at different layers in depth first order and at the same layer in width first order. Since domain experts may provide a number of input profiles from different kinds of users, profile algorithm first calls initProfile() to normalize these profiles for input profile of user interface. Function initProfile() averages all input profiles with their input probabilities as weights to obtain user interface profile

$$csa.first \text{ -> } input[i].profile = \sum_{j} csa.usage[j].input[i].profile * csa.usage[j].P \quad (1)$$

Then the algorithm defines a stack for traversal through CSA model composed of nodes which denote call actions and the actions nesting calls. We consider three program structures nesting calls: sequence, loop and branch, since these three fundamental structures are enough to realize any program function [5]. Each kind of action is further divided into action start and end, and thus the algorithm totally handles the following eight kinds of nodes.

(1) **Program Start**. It is a kind of virtual action to indicate the entrance to an interface without effective codes. In the case of this type of nodes, the algorithm only needs to give its input profile to the next node directly, and push the next node into stack.

(2) **Call Start**. Function passProfile() is called to propagate current input profile to called interface. passProfile() uses a syntax parser to gain parameter dependencies between two interfaces, which are expressed as arithmetic expressions with input parameters of current interface as variables to represent target parameters. Input profile of called interface can be solved with input profile of current interface and parameter dependencies between them. After passing profile, the algorithm pushes program start node of called interface into stack, and prepares to traverse the directed graph on the next layer.

(3) **Branch Start**. A branch structure nests a finite number of sub-actions, which are visited in width first order. For each sub-action, passProfile() is called to parse branch transition Boolean expression to solve sub-action profile as a subset of whole branch profile. Branch transition probability can be also obtained with sub-action profile. After profile propagation, the sub-action is pushed into stack.

(4) **Loop Start**. Function passProfile() is called to parse loop condition expression to solve loop profile with input profile of current node. Loop profile is characterized with loop iteration counts and the probabilities that iteration counts occur. Since loop action has no influence on node input profile, the profile can be passed directly to the sub-action which is pushed into stack after that.

(5) **Call End**. It represents an end to call an interface. At this time the node under this top node in stack is call start node because all the nodes between them that form the rounded actions in called interface have been visited and popped out of stack. Before popping these two nodes out of stack, the algorithm passes input profile of call start node to the next node of call end node which is then pushed into stack to make the traversal return to the upper layer.

(6) **Loop End**. It represents an end to a loop call. At this time the first two nodes in stack are loop end and loop start node since the sub-actions nested in loop have been visited and popped out of stack. Before these two nodes are popped out of stack, input profile of loop end node is propagated to the next node.

(7) **Branch End**. It is visited each time after a sub-action nested in a branch structure is visited. This node and branch start node are popped out of stack at the same time only if all branch sub-actions are visited; otherwise only this node is popped out. After visiting all sub-actions, input profile of branch end node is passed from the last sub-action and can't be propagated to the next node. Thus, the algorithm passes input profile of branch start node instead of branch end node to the next node.

(8) **Program End**. It represents an end to an interface program. At this time, the next node under this top node in stack is program start node. After they are popped out of stack, the next node that is call end node to an interface is pushed into stack. Profile propagation is not necessary since the profile of called interface has no influence on the profile of the interface calling it.

Profile algorithm uses system usage profile to build practical profile of all components in a component-based software system with the operations mentioned above. In order to facilitate the following software reliability analysis, it's necessary to map component practical profile into their input sub-domains for the probabilities that components run in each of their sub-domains. In terms of sub-domain concept, the probability in a sub-domain can be obtained with joint probability distribution of all parameter inputs in this sub-domain.

$$inf.sd[i].weight = \sum_{inf.input[j].profile \in sd[i]} \prod inf.input[j].profile.P \quad (2)$$

4.2 CSA-Based Software Reliability Analysis

The profile algorithm in last subsection assigns values to profile-related parameters in CSA model. Based on previous work we will carry out a scenario-based software reliability analysis in this subsection. A scenario, also called a system execution path, is a set of component interactions triggered by specific input stimulus, which can be obtained by traversing CSA model in depth first order. According to the definition of CSA model, a scenario corresponds to a sub-domain of user interfaces (UI) in a component-based software application.

Traditional scenario-based approaches implicitly represent system usage profile as the probabilities that scenarios occur, which are taken as weights to average all scenario reliability for system reliability.

$$R_{sys} = \sum_{i=1}^{n} (R_{path_i} * P_{path_i}) \tag{3}$$

In contrast to traditional approaches, we explicitly model the profile of call actions to solve call reliability, which are used to solve path reliability. With usage profile considered during reliability analysis, software reliability is equal to the sum of all path reliability, which is UI practical reliability in its sub-domains.

$$R_{sys} = UI.reliability = \sum_{i=1}^{n} UI.sd[i].reliability \tag{4}$$

An execution path corresponds to a UI sub-domain, which is composed of several calls. Unit testing on a UI is carried out under the assumption that all call actions in a UI wouldn't fail, whose results are failure probabilities in UI sub-domains. Thus, testing reliability in a UI sub-domain is expressed as

$$\begin{aligned} & 1 - UI.sd[i].fp \\ & = P(\text{UI don't fail in } sd[i] \mid \text{all calls don't fail in } sd[i]) \\ & = \frac{P(\text{UI and all calls don't fail in } sd[i])}{P(\text{all calls don't fail in } sd[i])} \\ & = \frac{UI.sd[i].R}{\prod_j R_{call_j}} \end{aligned} \tag{5}$$

where R_{call_j} is the reliability of a call action in the ith UI sub-domain.

From Eq. (5), path reliability is given by

$$UI.sd[i].reliability = (1 - UI.sd[i].fp) * \prod_j R_{call_j} \tag{6}$$

In addition to UI, other interfaces called by UI directly or indirectly in a component-based software application can be classified into two categories: one is called end interface (EI), which lies at the end of a path and does not call other interfaces; and the other is called middle interface (MI), which lies between FI and EI along a path and includes call actions. EI reliability is dependent on their practical profile.

$$EI.R = \sum_{EI.sd[j] \in UI.sd[i]} ((1 - EI.sd[j].fp) * EI.sd[j].weight) \tag{7}$$

where $EI.sd[j].weight$ is a sub-domain weight given by Eq. (2).

MI reliability is expressed as

$$MI.R = \sum_{MI.sd[j] \in UI.sd[i]} ((1 - MI.sd[j].fp) * \prod_k R_{call_k}) \tag{8}$$

From view of the above analysis, it's necessary to solve call reliability R_{call} in order to obtain path reliability. R_{call} depends on two factors: the practical reliability of called interfaces and call action profile. The former has been obtained by Eqs. (7) and (8). The latter varies with different kinds of call actions, which usually includes sequence call, loop call and branch call. Call reliability under these three call patterns is discussed as following.

(1) **Sequence Call**. It is the simplest way to make a call. It's not necessary to take call profile into account since a sequence call is not nested by other actions. Sequence call reliability is equal to practical reliability of called interface.

$$R_{call} = MI.reliability \text{ or } UI.reliability \tag{9}$$

(2) **Loop Call**. A loop structure has a serial of nested call actions. The loop contains a specification of loop iteration counts as a random variable over a finite domain of iteration counts $loop.profile.count \in \mathbb{N}_0$, each assigned a probability of its occurrence $loop.profile.P$. For loop reliability, we have

$$R_{call} = \sum_i \left(loop.profile.P[i] * \left(\prod_j R_{call_j} \right)^{loop.profile.count[i]} \right) \tag{10}$$

(3) **Branch call**. A branch structure nests a finite number of sub-actions each of which is composed of a serial of call actions. Branch profile is represented as branch transition probability. Thus, branch call reliability on a path is expressed as

$$R_{call} = \sum_{branch.child[j] -> input \in UI.sd[i]} \left(branch.P[j] * \left(\prod_k R^j_{call_k} \right) \right) \tag{11}$$

From view of the above discussion, it is a recursive call procedure for CSA-based reliability analysis, which terminates as arriving at the end of a path to compute EI reliability. EI reliability is directly solved by Eq. (7). Path reliability is obtained by backtrace from EI along a path step by step until arriving at UI. Eventually, component-based software reliability can be obtained with all path reliabilities as Eq. (4).

When taking communication link reliability into account, it's essential to refine the equations to compute call reliability. Since a call action includes message delivery and return, call reliability is given that

$$R'_{call} = R_{call} * (call.rlink)^2 \tag{12}$$

Then we will discuss how hardware failures influence the reliability of a component-based software system. Hardware resources are modeled with the properties of MTTF and MTTR, whose failure probability is given by

$$hw.fp = \frac{hw.MTTR}{hw.MTTF + hw.MTTR} \tag{13}$$

Hardware reliability is expressed as

$$hw.R = 1 - hw.fp = \frac{hw.MTTF}{hw.MTTF + hw.MTTR} \tag{14}$$

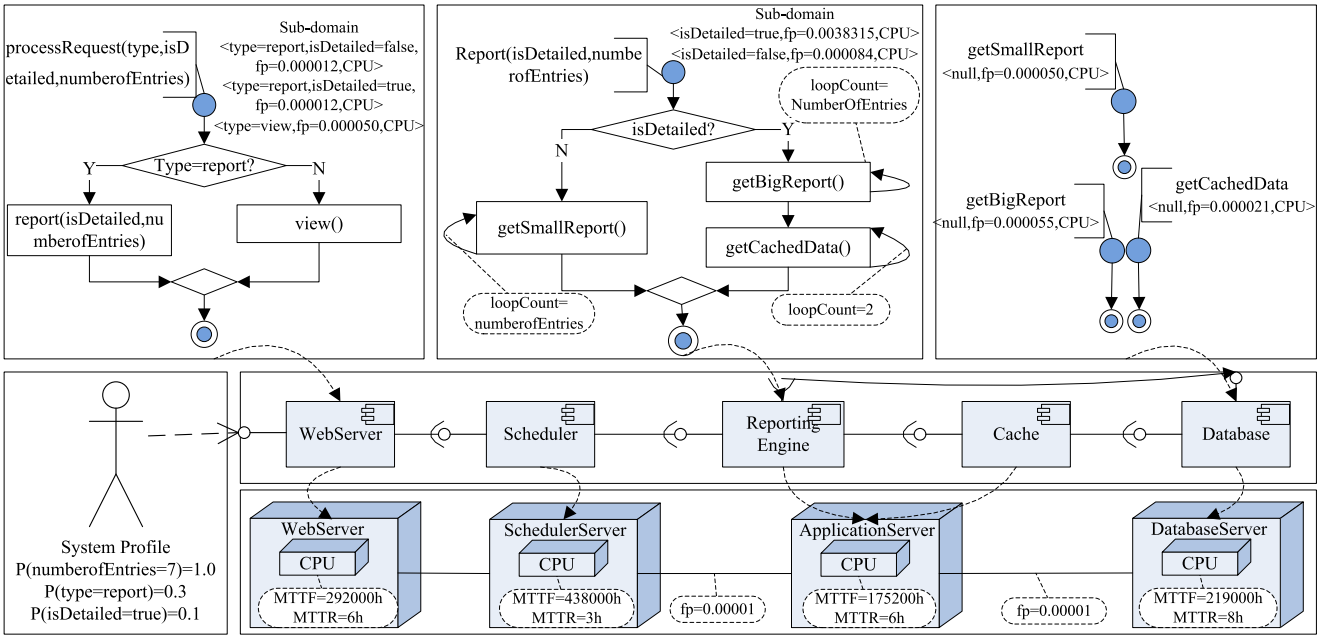

Fig. 4 A CSA model of a business report system.

A component-based software application is generally large and distributed with a good number of system resources. Existing architecture-based approaches solve system reliability in all possible cases of hardware resource availability with exponential size. It's easy for them to cause state space explosion. In contrast to exponential state space, the number of system execution paths is usually lineal. CSA-based approach builds state space with the physical states that hardware resources required by each path are normal, whose size is equal to the number of system paths.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be the physical state space, where n is the number of system paths, and $s_i \in S$ is the state probability that the hardware required by path i is normal.

$$\begin{aligned} s_i &= \prod_{inf.sd[j] \in UI.sd[i]} (1 - inf.sd[j].hw{->}fp) \\ &= \prod_{inf.sd[j] \in UI.sd[i]} \left(\frac{inf.sd[j].hw{->}MTTF}{inf.sd[j].hw{->}MTTF + inf.sd[j].hw{->}MTTR} \right) \end{aligned} \tag{15}$$

With the physical state space S, path reliability at corresponding physical state is given by $UI.sd[i].R * s_i$. Thus, component-based software reliability with physical resource states considered is expressed as

$$R_{sys}{}' = \sum_{i=1}^{n} (UI.sd[i].R * s_i) \tag{16}$$

5. Case Study Evaluation

In this section CSA-based software reliability analysis approach is applied to evaluate the reliability of a distributed component-based software system [3]. Evaluation result is compared with the results in [3] to demonstrate the prediction capabilities of our approach.

5.1 Software System Introduction

The component-based software system in [3] is called Business Reporting System (BRS), which is the basis for our case study evaluation. BRS generates management reports from business data collected in a database. First of all, we build a CSA model of this system as shown in Fig. 4.

Users can query the system via web browsers. They can simply view the currently collected data or generate different kinds of reports (coarse or detailed) for a configurable number of database entries. The usage model provided by the domain expert shows that a user requests a report in 30 percent of the cases, from which 10 percent are detailed reports. An average user requests reports for 7 database entries.

On a high abstraction level, the system consists of five independent software components running on four servers. The web server propagates user requests to a scheduler component, which dispatches them to possibly multiple application servers. The application servers host a reporting engine component, which either directly accesses the database or queries a cache component.

Table 1 Node profile in CSA.

node	profile
Interface processRequest()	P(numberOfEntries=7)=1
	P(type=report)=0.3; P(type=view)=0.7
	P(isDetailed=true)=0.1;
	P(isDetailed=false)=0.9
Interface Report()	P(numberOfEntries=7)=1
	P(isDetailed=true)=0.1;
	P(isDetailed=false)=0.9
Branch in processRequess()	left branch: P=0.3; right branch: P=0.7
Branch in report()	left branch: P=0.9; right branch: P=0.1
Loop nesting getSmallReport()	count=7, P=1
Loop nesting getBigReport()	count=7, P=1
Loop nesting getCachedData()	count=2, P=1

Table 2 Hardware reliability.

Hardware	Reliability
CPU_1	0.9999795
CPU_2	0.9999932
CPU_3	0.9999658
CPU_4	0.9999635

5.2 CSA-Based Software Reliability Analysis

Before evaluating the reliability of BRS, we have to compute its operational profile at first. The domain experts specify system level profile, as shown in the bottom left side of Fig. 4, and the software designers provides the information about how the components interact, as shown in the upper side of Fig. 4. With the knowledge of the system level profile and software architecture, we can execute the Profile algorithm in Fig. 3 to propagate system input profile to all the components. The obtained operational profiles are shown in Table 1.

There are three execution paths in BRS obtained by depth-first traversal, which correspond to three input subdomains of user interface processRequest(). First of all, we solve path reliability without consideration of hardware availability by Eq. (6). The results are 0.2698796, 0.0298719, and 0.699965. In this situation BRS reliability is

$$R_{sys} = UI.R = \sum_{i=1}^{3} UI.sd[i].R = 0.9997165 \tag{17}$$

In order to consider link reliability, we refresh call reliability on three paths by Eq. (12) to obtain three path reliabilities as 0.2698364, 0.0298660, and 0.6999510.

We annotate the CPU in WebServer, SchedulerServer, ApplicationServer, DatabaseServer with CPU_1, CUP_2, CPU_3, CPU_4 for convenience to discuss reliability analysis considering hardware availability, whose reliabilities are solved by Eq. (14) as shown in Table 2.

Based on the execution paths, we build system physical state space as $S = \{s_1, s_2, s_3\}$, where each item represents the probability that hardware resources required by corresponding path are normal. The solution by Eq. (15) is {0.9999018, 0.9999018, 0.9999795}.

Table 3 Hardware reliability.

Approach	Reliability
CSA-based approach	0.99960959
PCM-based approach	0.99960837
Simulation	0.99960658

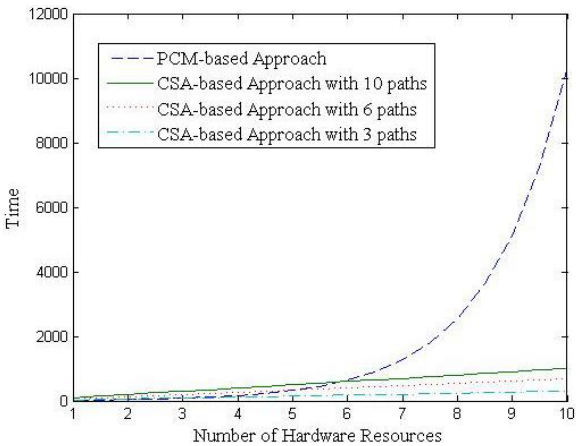

Fig. 5 Time complexity comparison.

With the above analysis results, software reliability considering hardware availability is solved by Eq. (16).

$$R_{sys}{}' = \sum_{i=1}^{3} (FI.sd[i].R * s_i) = 0.99960959 \tag{18}$$

We compare the reliability evaluation by CSA-based approach with that by PCM-based approach and the simulation result in [3], as shown in Table 3. It can be seen that three results are very close and the deviation is less than 0.001 percent.

Then we will further compare time complexity of CSA-based and PCM-based reliability analysis approach. Let the number of nodes in software architecture model be n and the number of 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 evaluates software reliability at each physical state, and the time complexity is $O(2^m \cdot n)$. The size of physical state space built by CSA-based approach is equal to the number of system execution paths obtained by depth-first traversal through CSA model. CSA-based approach employs a scenario-based method to evaluate software reliability at each physical state, and time complexity is $O(e \cdot m \cdot n)$, where e is the number of paths. Figure 5 shows the time complexity of the two approaches in the case of $n = 10$. CSA-based approach defines a loop action to a path by the way of modeling loop profile, which makes the approach not necessary to consider the situation of infinite paths due to loop. As a result, the number of execution paths in a software system is finite and meets $e \leq n$. Figure 5 illustrates

the time complexity of CSA-based approach with $e = 10$, 6, and 3. It can be seen that CSA-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.

6. Conclusion

In this paper we propose a new approach to model and analyze component-based software reliability, which improves traditional architecture-based approaches to overcome some of their problems. The approach builds a two-layer model to describe software architecture for the sake of software developers, where reliability-related factors are well modeled to improve the accuracy of software reliability evaluation. A scenario-based reliability analysis approach is employed to avoid state space explosion and improve the efficiency of reliability analysis. Taking software reliability prediction as a reference, software architects can constantly improve architecture design to achieve expected reliability requirements.

Acknowledgments

This research was supported by the National Natural Science Foundation of China under Grand No.61402333, No. 61402242, No. 61272450 and No.61202381.

References

[1] A. Amin, L. Grunske, and A. Colman, "An approach to software reliability prediction based on time series modeling," J. Systems and Software, vol.86, no.7, pp.1923–1932, 2013.
[2] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the palladio component model," IEEE Trans. Software Engineering, vol.38, no.6, pp.1319–1339, 2011.
[3] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," in Research into Practice–Reality and Gaps, pp.36–51, Springer, 2010.
[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. Software Engineering, vol.29, no.10, pp.946–960, 2003.
[5] D. Hamlet, "Tools and experiments supporting a testing-based theory of component composition," ACM Trans. Software Engineering and Methodology (TOSEM), vol.18, no.3, pp.12–52, 2009.
[6] M.W. Lipton and S.S. Gokhale, "Heuristic component placement for maximizing software reliability," in Recent Advances in Reliability and Quality in Design, pp.309–330, Springer, 2008.
[7] R.H. Reussner, H.W. Schmidt, and I.H. Poernomo, "Reliability prediction for component-based software architectures," J. Systems and Software, vol.66, no.3, pp.241–252, 2003.
[8] N. Sato and K.S. Trivedi, "Accurate and efficient stochastic reliability analysis of composite services using their compact Markov reward model representations," Proc. 2007 IEEE International Conference on Services Computing (SCC 2007), pp.114–121, Salt Lake City, Utah, USA, 2007.
[9] V.S. Sharma and K.S. Trivedi, "Quantifying software performance, reliability and security: An architecture-based approach," J. Systems and Software, vol.80, no.4, pp.493–509, 2007.
[10] W. Wang, T.L. Hemminger, and M. Tang, "A moving average non-homogeneous Poisson process reliability growth model to account for software with repair and system structures," IEEE Trans. Reliab., vol.56, no.3, pp.411–421, 2007.
[11] 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.

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.

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.

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.

Kai Shi received the PhD degree in computer science and technology from Tianjin University in 2010. Currently, he is working as an associate professor in school of computer and communication engineering, Tianjin University of Technology. His main research interests include wireless network and flow control.