ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/317773925

Dynamic structure measurement for distributed software

Article *in* Software Quality Journal · September 2018

DOI: 10.1007/s11219-017-9369-3

CITATIONS
9

READS
234

6 authors, including:

Wuxia Jin
Xi'an Jiaotong University
20 PUBLICATIONS **212** CITATIONS
SEE PROFILE

Ting Liu
Xi'an Jiaotong University
132 PUBLICATIONS **2,384** CITATIONS
SEE PROFILE

Yu Qu
University of California, Riverside
27 PUBLICATIONS **307** CITATIONS
SEE PROFILE

Qinghua Zheng
Xi'an Jiaotong University
500 PUBLICATIONS **6,525** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Securing Outsourced Data in Cloud with SGX View project

Project quality of distributed system View project

All content following this page was uploaded by Yu Qu on 08 September 2017.

The user has requested enhancement of the downloaded file.

Software Qual J
DOI 10.1007/s11219-017-9369-3

Dynamic structure measurement for distributed software

Wuxia Jin[1] · Ting Liu[1] · Yu Qu[1] · Qinghua Zheng[1] · Di Cui[1] · Jianlei Chi[1]

© Springer Science+Business Media New York 2017

Abstract With the advent of network technologies and the ultra-fast increasing of computing ability, the distributed architecture has become a necessity for the majority of software systems. However, it is difficult for current architecture measurements to evaluate distributed systems, such as cohesion and coupling. Most current methods focus on the relations among various classes or packages but barely consider the structure at component level, which has a serious impact on change impact analysis, fault diagnosis, or other maintenance activities. In this paper, we propose a dynamic structure measurement for distributed software. The intra-component and inter-component dependencies are introduced into a Calling Network model to further represent distributed software. More importantly, based on the Kieker monitoring framework, the measurement methods are proposed and implemented for distributed software. Two structural quality attributes cohesion factor of component (CHC) and coupling factor of component (CPC) are measured. Finally, case studies are conducted on two open-source distributed systems: *RSS Reader Recipes* and the

✉ Wuxia Jin
wx_jin@stu.xjtu.edu.cn

Ting Liu
tingliu@mail.xjtu.edu.cn

Yu Qu
quyuxjtu@mail.xjtu.edu.cn

Qinghua Zheng
qhzheng@mail.xjtu.edu.cn

Di Cui
cuidi@sei.xjtu.edu.cn

Jianlei Chi
chijianlei7@sei.xjtu.edu.cn

[1] Ministry of Education Key Lab for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China

Published online: 05 June 2017

distributed version of *iBATIS JPetStore*. By applying the proposed methods and comparing with the existing ones, the features of CHC and CPC can be assessed and observed for distributed software.

Keywords Dynamic metric · Calling network · Distributed software · Structure measurement

1 Introduction

In recent years, various new application scenarios are almost impossible to run on a single machine, such as gene analysis, social network, big data applications, etc., due to the increasing difficulty of exploiting higher CPU speed or larger memory of a single machine. Moreover, the size of industrial software has been growing for decades, and the maintenance is often hindered by technical debt (Lin et al. 2016). So it has become too difficult to make changes for industrial software with business requirements. As a consequence, the idea that splitting a big single-server application into smaller cooperating components to form a distributed system is emerging (Thones 2015). Such distributed system can make use of hardware power of more than one computers, also can help software be easier to change, scale, evolve, and maintain. In general, distributed software has been a popular trend as one kind of software paradigms. Because of distributed nature, the structure of distributed software is more complex when compared with single-server software. Such structural complexity makes a serious impact on the development or maintenance activities such as change impact analysis (Cai and Thain 2016), fault diagnosis (Nguyen et al. 2013), etc. Therefore, it is crucial to measure structure quality of distributed software. Structure quality measurement aims at software quality assurance, and software with high quality is likely to be stable and maintainable (Dallal and Briand 2012).

Distributed system is a collection of independent computers that appears to its users as a single coherent system (Tanenbaum and Steen 2002). A distributed system consists of components (i.e., computers) that are autonomous. A component can be a process or any piece of hardware required to run a process.[1] In this paper, regardless of the hardware level, we only focus on *Distributed Software* itself. We define a *Component* of distributed software as one software modular unit corresponding to one autonomous process. Component is an independent and more abstract modular unit than class or package. These components collaborate with others dynamically to achieve a common goal. *Node* is an autonomous computing unit, which can be the physical computer, virtual machine (VM), container, etc. Different components are deployed on different nodes respectively, or more than one components are deployed on the same node. Components interact with remote ones by employing communication techniques including remote method invocation (RMI), and other third party communication framework (Coulouris et al. 2012). It is clear that there are complicated inter-component behaviors in a distributed system which cannot be captured by static analysis.

It is widely accepted that the structure quality of software can be estimated by measuring cohesion and coupling attributes. The *Cohesion* of a module indicates the extent to which the components (here the components refer to elements) of the module are related (Bieman and Ott 1994). *Coupling* is defined as the measure of the strength of association

[1] http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html#Basics

established by a connection from one module to another (Stevens et al. 1974). Well-designed or well-implemented software systems always follow the principle "high cohesion and low coupling." To measure cohesion and coupling attributes in a quantitative way, a variety of corresponding metrics have been designed, such as works of Chidamber and Kemerer (1994), Yacoub et al. (1999), Ying et al. (2004), Counsell et al. (2006), Qu et al. (2015a), etc. However, cohesion and coupling concepts are proposed in the period when distributed software did not exist. It is not surprising that these quality attributes mostly are measured for single-server software, without being assessed for distributed software.

But far beyond that, distributed software has its own unique feature for measuring these attributes. Figure 1 illustrates an example on single-server and distributed versions of a software system. C_i denotes a component. m_i denotes a method. A directed edge from m_i to m_j means a method call. Methods belonging to different components are rendered in different color separately. Also, method call within components and across components are rendered in different color. As shown in Fig. 1, besides intra-component dependencies similar to ones in single-server software, there are inter-component dependencies in distributed software. Both intra-component and inter-component dependencies constitute the structure relations in distributed software. Through the collaborating function of these dependencies, normal or abnormal behaviors in one component often propagate to others. If components are designed or partitioned with proper dependencies shown in Fig. 1b, the ripple effect can be controlled within the root-cause component to a certain extent, reducing the influenced scope. On the contrary, components in Fig. 1c with improper dependencies could be very difficult for understanding and maintenance. So it is critical to estimate structure quality at component level by considering both intra and inter component dependencies for distributed software. Such measurement can estimate the structure quality of a new distributed software, and also can guide the partitioning in the transformation from legacy single-server to distributed version. However, the existing related works mostly estimate software system at class or package level but not component level. Furthermore, no works take account into the inter-component dependencies which cannot be captured by static analysis of source code.

In this paper, we address the issue of structure measurement for distributed software. For distributed system, the structural dependency relationships are introduced from two aspects. One is natural relationship introduced by inter-component communication type and the other is the relationship introduced by designers or developers (Indrajit Wijegunaratnec and Fernandez 1998). Since the dependencies introduced by the former can be finally exposed on software itself, here we focus on the dependencies introduced by the latter. Specific for distributed software, we define structure attributes cohesion factor of component (CHC) and coupling factor of component (CPC) which are consistent with traditional cohesion and coupling concepts.

CHC is the extent to which the elements (methods, classes, etc.) of a component are related in a distributed software system.

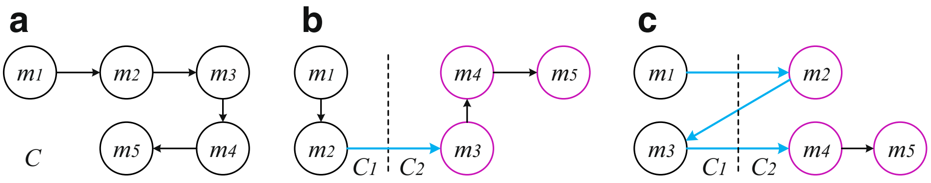


Fig. 1 **a** Single-server version. **b** One distributed version. **c** Another distributed version

CPC is the measure of the strength of association established by a connection from one component to another component in a distributed software system.

In order to measure CHC and CPC attributes in a quantitative way, we design the corresponding metrics by considering the distinguishing characteristics of distributed software. Our work makes the following main contributions:

1. The structure quality measurement is proposed for distributed software, which has its own structure characteristics when compared with single-server software.
2. Based on Kieker monitoring framework (Hoorn et al. 2012), the dynamic structure measurement approaches are proposed and implemented for distributed software. Based on the extended calling network, Growing CN and Partitioned CN models (Jin et al. 2016), CHC and CPC attributes are measured. Information used by the metrics is exposed in method-method interaction, one view of software attributes (Allen et al. 2001; Dallal and Briand 2012). More importantly, we emphasize inter-component dependencies, and take into account both direct and indirect dependency relationships.
3. Case studies are conducted on real-world distributed software systems: *RSS Reader Recipes* and the distributed version of *iBATIS JPetStore*. By employing the proposed methods and comparing with the existing measurements, studies show that our approaches are able to estimate the structure quality of distributed software. The results are consistent with the expected features.

The rest of this paper is organized as follows. The extended calling network model is introduced in Section 2. Section 3 presents the proposed metrics and validates their mathematical properties. Section 4 demonstrates the case studies. Section 5 makes discussion. Section 6 introduces some related works. Make conclusion and discuss future work in Section 7.

2 Extended calling network

Qu et al. (2015b) proposed calling network (CN) model for software system. A lot of works have used this model (Wang et al. 2017; Tian et al. 2017). We extend the model considering distributed software. The extended CN model is given in the following:

Definition 1 Calling behaviors *cb*. *cb* is a behavior record of one method call. $cb_k = (t_k, Caller_k, Callee_k, Param_k, VM_{Caller_k}, VM_{Callee_k})$. Where t_k is the timestamp of the method call. $Caller_k$ and $Callee_k$ are self-descriptive. $Param_k$ is the parameter list of $Callee_k$. VM_{Caller_k} is the name of component to which the method belongs. So as VM_{Callee_k}.

Definition 2 Calling behavior set *CB*. $CB = \{cb_k \mid k \in \mathbb{N}\}$. *CB* is an ordered set. According to the execution timestamp of all method calls, *k* is the sequence number of *cb*.

Definition 3 Calling graph *CG*. $CG = (V, E)$. *CG* is a directed graph, in which *V* stands for the method set and *E* stands for the set of method call relations. Let L^V be the set of vertex labels, and L^E be the set of edge labels. $L^V \neq \emptyset$, $L^E \neq \emptyset$. Let A^D be the set of discrete attribute values and $A^N \subset \mathbb{R}$ be the set of numeric attribute values, such that $A = A^N \cup A^D$. The label-to-value mapping function for vertex is denoted as $f_v : V \times L^V \rightarrow A$.

The label-to-value mapping function for edge is denoted as $f_e : E \times L^E \to A$. The vertex label and edge label can have various meanings in different scenarios. In this paper, vertex label is the name of component to which the method belongs. Edge label is weight, the method call frequency. That is, $f_v : V \times L^V \to A^D$, $f_e : E \times L^E \to A^N$.

Definition 4 Calling graph generation function: $f_{CG\_Gen} : CB \to CG$.

Definition 5 Calling network (CN). CN is an ordered set of *CG*: CN $= \{CG_i \mid i \in \mathbb{N}\}$, Where $CG_i = f_{CG\_Gen}(CB_i), CB_i \subseteq CB$. *CB* can be partitioned into CB_i using some strategies. In this paper, we discuss two strategies. Other strategies also fit extended CN model.

Strategy 1 Use fixed interval and quantity of *cbs* to generate *CG*. Two parameters are needed in this strategy: N_{Itv} and N_{CG}. N_{Itv} represents the interval between two consecutive *CBs*, and N_{CG} represents the number of *cbs* in each CB_i. Then, $CB_i = \{cb_k \mid (i-1) \cdot N_{Itv} < k \leq (i-1) \cdot N_{Itv} + N_{CG}\}$.

Strategy 2 Use time interval to partition *CB*. Two parameters are set: T_i and $\triangle t$. T_i is the i-th time point, $\triangle t$ is the time window for selecting *cbs*. Then, $CB_i = \{cb_k \mid T_i - \triangle t < t_k \leq T_i\}$.

In general, the extended CN model is formalized as (Strategy 1 is included):

$$\begin{cases} CN = \{CG_i \mid i \in \mathbb{N}\} \\ CG_i = f_{CG_{Gen}}(CB_i), CB_i \subseteq CB \\ CB_i = \{cb_k \mid (i-1) \cdot N_{Itv} < k \leq (i-1) \cdot N_{Itv} + N_{CG}\} \\ CG = (V, E), f_v : V \times L^V \to A^D, f_e : E \times L^E \to A^N \\ CB = \{cb_k \mid k \in \mathbb{N}\} \\ cb_k = (t_k, Caller_k, Callee_k, Param_k, L^{V_{Caller_k}}, L^{V_{Callee_k}}) \end{cases}$$

Then two CN generation schemes are proposed by setting different parameters of extended CN model above:

Scheme 1 Growing calling network (Growing CN). Here, use Strategy 1 to partition *CB*. Assuming $N_{Itv} = 0$ and $N_{CG} = i \cdot N_{Const}$, where N_{Const} is a constant value. Then $CB_i = \{cb_k \mid 0 < k \leq i \cdot N_{Const}\}$. Obviously, this sequence of *CGs* represents the growing process of calling graph over time.

Scheme 2 Partitioned calling network (Partitioned CN). Here, use Strategy 2 to partition *CB*. T_i is the end time of one business functionality, and $\triangle t$ is the processing time window. For ith business functionality, $T_i - \triangle t$ is the starting time, and T_i is the end time of processing. $CB_i = \{cb_k \mid T_i - \triangle t < t_k \leq T_i\}$.

Growing CN is a sequence of dynamic *CGs*, and represents the growing process of calling graph over time. Partitioned CN is a dynamic *CG* in a time interval. Through Growing CN and Partitioned CN, the software dynamic behavior evolving over time can be presented and observed intuitively.

3 Distributed software structure measurement

Different consideration reflects different view of measurement, so the underlying hypotheses which drive structure measurement need be specified (Briand et al. 1999). We design metrics for CHC and CPC attributes, and the hypotheses also also will be described.

3.1 Cohesion factor of component

Based on the extended CN model, we design metric CC_i for measuring CHC attribute of component C_i according to the following hypotheses.

Hypothesis H1 *Intra-component dependencies contribute to CHC. More dependencies within one component, more CHC the component will be.*

Hypothesis H2 *Inter-component dependencies make one aspect of contribution in reducing CHC of the involved component.*

Since CHC measures the cohesion inside a component, the inter-component dependency direction is insignificant. Hence, we measure CHC based on undirected extended CN model.

$$\begin{aligned} CG &= (V, E),\, f_v : V \times L^V \rightarrow A^D,\, f_e : E \times L^E \rightarrow A^N \\ P_i &= \{e_k \mid e_k = (v_{k-1}, v_k),\, f_v(L^{v_{k-1}}) = f_v(L^{v_k}) = C_i\} \\ Q_i &= \big\{e_k \mid e_k = (v_{k-1}, v_k),\, \big(f_v(L^{v_{k-1}}) = C_i \; and \; f_v(L^{v_k}) \neq C_i\big) \\ &\quad\; or \; \big(f_v(L^{v_{k-1}}) \neq C_i \; and \; f_v(L^{v_k}) = C_i\big)\big\} \end{aligned}$$

P_i is a set of edges which are within component C_i. Q_i is a set of inter-component edges which are across component C_i. CC_i of component C_i is formulated as:

$$CC_i = \frac{\alpha \cdot |P_i| - \beta \cdot |Q_i|}{|N_i|}, \qquad (\alpha + \beta = 1,\, \alpha > 0,\, \beta > 0,\, |P_i| \geq |Q_i|)$$

In a complex distributed software, component is an independent unit, so intra-component interaction is larger than inter-component interaction. That is, for component C_i, $|P_i| \geq |Q_i|$. N_i denotes the edge set of all possible method interactions within component C_i. The contribution of intra-component dependency to CHC is not equal to that of inter-component dependency, so two weight factors are defined. α is the weight of intra-component dependency, while β is the weight of inter-component one. As $|P_i| \geq |Q_i|$, CC_i value always is smaller than that of not differentiating the weight. By emphasizing the weight factors, CC_i metric is able to consistent with the characteristics of distributed software.

A set of properties are proposed for defining cohesion by Briand et al. (1998). They are widely supported by many related works, such as Allen et al. (2001), Briand et al. (1998), Zhou et al. (2004), and Al Dallal (2010). Consistent with those properties, we reserve CHC and the properties are shown in Table 1.

Then, we validate that the proposed CC_i has the properties in Table 1.

Property 1 *Nonnegativity and normalization.* When there is no interaction ($|P_i| = 0, |Q_i| = 0$), $CC_i = 0$. When all possible method interactions happen within a component, and there is no inter-component interaction, CC_i reaches the MAX. This is obvious and accords with our intuition.

Table 1 The properties of CHC

Concept/Properties
CHC for distributed software:
1. Nonnegativity and normalization. CHC belongs to a specified interval [0, MAX]
2. Null value. CHC is null if the set of intra-component edges of a component is empty
3. Monotonicity. Adding an intra-component edge to a component does not decrease its CHC
4. Merging of components. If two unrelated component C_1 and C_2 are merged to form a new component $C_3 = C_1 \cup C_2$ that replaces C_1 and C_2, then CHC of C_3 is not greater than the maximum CHC of C_1 and C_2

Property 2 *Null value.* If the set of intra-component interaction of component C_i is empty ($|P_i| = 0$), and inter-component interaction of component C_i is empty ($|Q_i| = 0$), then CC_i is null.

Property 3 *Monotonicity.* Add an intra-component edge to component C_i as a new component C_i'.

$$\begin{aligned} CC_i' - CC_i &= \frac{\alpha \cdot |P_i'| - \beta \cdot |Q_i'|}{|N_i'|} - \frac{\alpha \cdot |P_i| - \beta \cdot |Q_i|}{|N_i|} \\ &= \frac{\alpha \cdot (|P_i| + 1) - \beta \cdot |Q_i|}{|N_i| + 1} - \frac{\alpha \cdot |P_i| - \beta \cdot |Q_i|}{|N_i|} \\ &= \frac{\alpha \cdot (|N_i| - |P_i|) + \beta \cdot |Q_i|}{(|N_i| + 1) \cdot |N_i|} \geq 0 \end{aligned}$$

Therefore, it can be concluded CC satisfies Property 3.

Property 4 *Merging of components.* Let two unconnected component C_1 and C_2 be merged to form a new one $C_3 = C_1 \cup C_2$.

Hypothesis $max\{CC_1, CC_2\} = CC_1$, then $CC_1 \geq CC_2$.

$$\begin{aligned} CC_1 \geq CC_2 &\Rightarrow \frac{\alpha \cdot |P_1| - \beta \cdot |Q_1|}{|N_1|} \geq \frac{\alpha \cdot |P_2| - \beta \cdot |Q_2|}{|N_2|} \\ &\Rightarrow (\alpha \cdot |P_1| - \beta \cdot |Q_1|) \cdot |N_2| \geq (\alpha \cdot |P_2| - \beta \cdot |Q_2|) \cdot |N_1| \end{aligned}$$

$$\begin{aligned} CC_3 - max\{CC_1, CC_2\} &= CC_3 - CC_1 \\ &= \frac{\alpha \cdot (|P_1| + |P_2|) - \beta \cdot (|Q_1| + |Q_2|)}{|N_1| + |N_2|} - \frac{\alpha \cdot |P_1| - \beta \cdot |Q_1|}{|N_1|} \\ &= \frac{(\alpha \cdot |P_2| - \beta \cdot |Q_2|) \cdot |N_1| - (\alpha \cdot |P_1| - \beta \cdot |Q_1|) \cdot |N_2|}{(|N_1| + |N_2|) \cdot |N_1|} \leq 0 \end{aligned}$$

It can be concluded $CC_3 \leq max\{CC_1, CC_2\}$. So, CC satisfies Property 4.

3.2 CouPling factor of component

We design a metric CP_i for measuring CPC attribute for component C_i according to the following hypotheses.

Hypothesis H3 *Both direct inter-component dependency and indirect inter-component dependency contribute to CPC.*

Hypothesis H4 *Indirect inter-component dependency is indicated in calling path across components. And the calling path is deeper, the indirect dependency is weaker. When the depth is equal to 1, it becomes direct dependency.*

Hypothesis H5 *In the definitions of direct and indirect dependency, direct dependency is stronger than indirect one.*

CPC measures the coupling between two components, and dependency strength of component i be coupled to component j is not symmetrical with that of the opposite coupled direction. So we make distinction between inbound CPC and outbound CPC. That is, we measure CPC attribute based on directed extended CN model. Metric $CP_{i,j}$ and CP_i are formulated as the following:

$$\begin{aligned}
CG &= (V, E),\ f_v : V \times L^V \to A^D,\ f_e : E \times L^E \to A^N \\
P_i &= \{e_k \mid e_k = (v_k, v_{k+1}),\ f_v(L^{v_k}) = f_v(L^{v_{k+1}}) = C_i\} \\
Q_{i,j} &= \{e_k \mid e_k = (v_k, v_{k+1}),\ f_v(L^{v_k}) = C_i,\ f_v(L^{v_{k+1}}) = C_j\} \\
Q_i &= \{e_k \mid e_k = (v_k, v_{k+1}),\ f_v(L^{v_k}) = C_i,\ f_v(L^{v_{k+1}}) \neq C_i\} \\
Path_{k,l} &= \{e_k, e_{k+1}, \ldots, e_{l-1}\} = \{(v_k, v_{k+1}), (v_{k+1}, v_{k+2}), \ldots, (v_{l-1}, v_l)\} \\
Meth_{k,l} &= f_m(Path_{k,l}) \\
PATH_{i,j} &= \{Path_{k,l} \mid e_k, e_{k+1}, \ldots, e_{l-2} \in P_i,\ e_{l-1} \in Q_{i,j}\} \\
PATH_i &= \{Path_{k,l} \mid e_k, e_{k+1}, \ldots, e_{l-2} \in P_i,\ e_{l-1} \in Q_i\} \\
CP_{i,j} &= \sum_{n=1}^{|PATH_{i,j}|} Meth_{k,l} = \sum_{n=1}^{|PATH_{i,j}|} f_m(Path_{k,l}), \qquad Path_{k,l} \in PATH_{i,j} \\
CP_i &= \sum_{i \neq j} CP_{i,j}
\end{aligned}$$

P_i is a set of intra-component directed edges within component C_i. $Q_{i,j}$ is a set of inter-component directed edges starting from C_i to C_j. $Path_{k,l}$ is an order set of edges, and it means a directed path from $v_k$to v_l. $Meth_{k,l}$ is a mapping result of $Path_{k,l}$, and it means the dependency strength from $v_k$to v_l. f_m is the mapping function. $PATH_{i,j}$ is a set of $Path_{k,l}$, whose last edge is inter-component across C_i and other edges are intra-component within C_i. $CP_{i,j}$ measures CPC from C_i to C_j and it is the sum of dependency strength indicated in $f_m(Path_{k,l})$. CP_i is the total CPC of C_i.

Here, we choose the mapping function $f_m(n) = \frac{1}{n^2}$, $n \in \mathbb{N}$, because it owns two important mathematical properties:

Rigorous decrease of $\frac{1}{n^2}$ Rigorous decrease implies that the dependency strength from v_k to v_l is weaker when the path length from v_k to v_l is larger. When the length is equal to 1 ($n = 1$), the dependency reaches the max ($\frac{1}{n^2} = 1$). This is consistent with hypothesis H4.

The convergence of series $\sum \frac{1}{n^2}$ The convergence implies and makes sure that the sum of indirect dependency strength is less than direct one. $\lim\limits_{n\to\infty} \sum_{n=1} \frac{1}{n^2} = \frac{\pi^2}{6}$, so $\lim\limits_{n\to\infty} \sum_{n=2} \frac{1}{n^2} = \frac{\pi^2}{6} - 1 < 1$.

A set of properties are proposed for defining coupling by Briand et al. (1999). Many coupling works support the properties, such as Allen et al. (2001), Briand et al. (1999), and

Arisholm et al. (2002). Consistent with these properties, we reserve CPC for distributed software, and the properties of CPC are listed in Table 2.

Then we validate that the proposed CP_i has the properties in Table 2.

Property 1 *Nonnegativity.* $|PATH_{i,j}| \geq 0$, so CP_i is nonnegativity. This also reflects our intuitive judgment.

Property 2 *Null value.* If the set of inter-component edges is empty, $Q_{i,j} = \emptyset$. $PATH_{i,j} = \emptyset$, hence $CP_{i,j}$ is null, and CP_i is null.

Property 3 *Monotonicity.* Add an inter-component edge to a component C_i as a new C'_i. $|Q'_{i,j}| \geq |Q_{i,j}|$,hence $|PATH'_{i,j}| \geq |PATH_{i,j}|$. So $CP'_{i,j} \geq CP_{i,j}$, then $CP'_i \geq CP_i$.

Property 4 *Merging of components.* Merge two component C_1 and C_2 to form a new one $C_3 = C_1 \cup C_2$. Because the original inter-component edges from C_1 to C_2 no longer belong to inter-component ones in C_3, $|Q_3| \leq |Q_1| + |Q_2|$, then $|PATH_3| \leq |PATH_1| + |PATH_2|$. So $CP_3 \leq CP_1 + CP_2$.

Property 5 *Disjoint component additivity.* Merge two unconnected components C_1 and C_2 to form a new one $C_3 = C_1 \cup C_2$. Because there is no inter-component edge from original C_1 to original C_2, $|Q_3| = |Q_1| + |Q_2|$, then $|PATH_3| = |PATH_1| + |PATH_2|$. So $CP_3 = CP_1 + CP_2$.

4 Case studies

4.1 Setting

We conduct experiments on two open-source distributed software systems. By using our measurement approaches, the following processes are done for each software system:

Construction of inputs In order to collect execution behaviors, we need design the representative inputs for driving the target software system. Two alternative methods are widely used: designing test cases and designing test scenarios (Arisholm et al. 2002). In our cases, the software system is web-based and exposes the functionality features to users through

Table 2 The properties of CPC

Concept/Properties
CPC for distributed software:
1. Nonnegativity. The CPC between two components is nonnegativity
2. Null value. CPC is null if the set of inter-component edges of a component is empty
3. Monotonicity. Adding an inter-component edge to a component does not decrease its CPC
4. Merging of components. If two component C_1 and C_2 are merged to form a new component $C_3 = C_1 \cup C_2$ that replaces C_1 and C_2, CPC of C_3 is not greater than the sum of CPC of C_1 and C_2
5. Disjoint component additivity. If two unconnected component C_1 and C_2 are merged to be a new component $C_3 = C_1 \cup C_2$ that replaces C_1 and C_2, then CPC of C_3 is equal to the sum of CPC of C_1 and C_2

GUI. Through GUI, it is convenient to collect traces by invoking specific features. In this way, the need to have an accurate test case suite can be avoided. So we choose to design test scenarios for each software system. This method of constructing inputs also has been adopted by Bavota et al. (2013) for dynamic measurements.

Deployment and monitoring We deploy and run the distributed software on VMs. Different component locates in different VM. We use Kieker tool (version 1.12) (Hoorn et al. 2012) to collect execution data. Kieker provides dynamic analysis capabilities including monitoring and analyzing a software system runtime behaviors. It can enable application performance monitoring and architecture discovery. In particular, this tool provides probes for monitoring a distributed system. Driven by above test scenarios as inputs, the execution behaviors can be recorded once the application is running. Figure 2 illustrates partial records when monitoring *RSS Reader Recipes* application, one of the target software systems.

In Fig. 2, each line corresponds to one record. There are ten items for each record, and items are separated by ";". From left to right, the items are *Type*, *Time*, *Method*, *SessionID*, *TraceID*, *Tin*, *Tout*, *HostName*, *EOI*, and *ESS*. *Type* item means the type of this record. In the following experiments, we only focus on the "\$1" about calling behaviors. *Time* item means the timestamp when this execution behavior happens. *Method* item denotes the full signature of an invocated method, including modifier, class name, method name, and parameter list. *SessionID* item is a globally unique number representing each session. *TraceID* is a globally unique number representing each trace. *Tin* is the timestamp just before entering this method. *Tout* is the timestamp just after finishing execution. *HostName* item is the node in which the method is running. In the following experiments, one component is deployed on the corresponding node, so *HostName* item can be used to differentiate different components. *EOI* item is the calling order of the method. *ESS* is the depth of the calling stack of the method. Based on *SessionID*, *TraceID*, *EOI*, and *ESS* items of all records generated by Kieker, intra-component interactions can be obtained, and inter-component interactions can be associated from one component to others. Finally, a set of representative dynamic data can be collected.

Modeling Analyzing the set of collected data using Python script, we adopt two schemas in Section 2 to generate models: Growing CN and Partitioned CN. These models are two different views of abstract presentation of the distributed software.

CHC and CPC measurements According to the formulations in Section 3, we measure CHC and CPC attributes by computing *CC* and *CP* metrics based on the above models. And Networkx 1.11[2] Library is used.

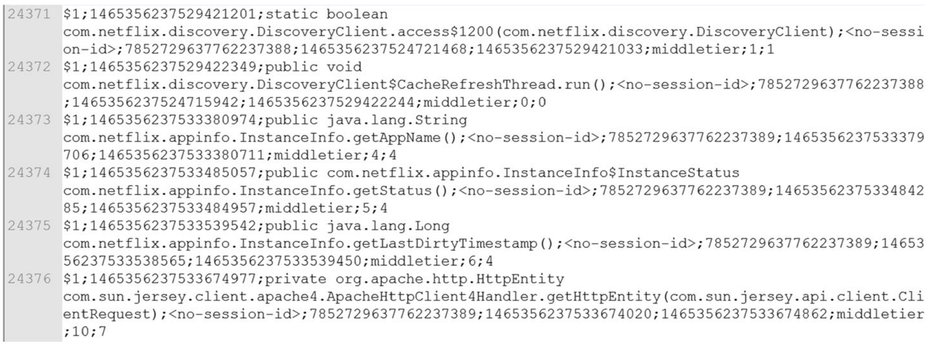
4.2 Target software systems

4.2.1 RSS reader recipes application

RSS Reader Recipes[3] is a distributed enterprize application developed by Netflix. It provides web service for users to get, add, and delete RSS feeds. The lines of code is about 20K

[2]http://networkx.github.io

[3]https://github.com/Netflix/recipes-rss



```
$1;1465356237529421201;static boolean
com.netflix.discovery.DiscoveryClient.access$1200(com.netflix.discovery.DiscoveryClient);<no-sessi
on-id>;7852729637762237388;1465356237524721468;1465356237529421033;middletier;1;1
$1;1465356237529422349;public void
com.netflix.discovery.DiscoveryClient$CacheRefreshThread.run();<no-session-id>;7852729637762237388
;1465356237524715942;1465356237529422244;middletier;0;0
$1;1465356237533380974;public java.lang.String
com.netflix.appinfo.InstanceInfo.getAppName();<no-session-id>;7852729637762237389;1465356237533379
706;1465356237533380711;middletier;4;4
$1;1465356237533485057;public com.netflix.appinfo.InstanceInfo$InstanceStatus
com.netflix.appinfo.InstanceInfo.getStatus();<no-session-id>;7852729637762237389;14653562375334842
85;1465356237533484957;middletier;5;4
$1;1465356237533539542;public java.lang.Long
com.netflix.appinfo.InstanceInfo.getLastDirtyTimestamp();<no-session-id>;7852729637762237389;14653
56237533538565;1465356237533539450;middletier;6;4
$1;1465356237533674977;private org.apache.http.HttpEntity
com.sun.jersey.client.apache4.ApacheHttpClient4Handler.getHttpEntity(com.sun.jersey.api.client.Cli
entRequest);<no-session-id>;7852729637762237389;1465356237533674020;1465356237533674862;middletier
;10;7
```

Fig. 2 Partial records when monitoring *RSS Reader Recipes* application by Kieker

statistically estimated by SCITools Understand.[4] It contains three components, including Middletier, Edge, and Eureka. In our experiment, these components are deployed on three different VMs hosted on Ubuntu15.10 server. The whole framework is shown in Fig. 3.

4.2.2 *iBATIS JPetStore application*

We use the distributed version of *iBATIS JPetStore* referring to Marwede et al. (2009). The *iBATIS JPetStore* Demo is an online pet store. Like most e-stores, users can browse and search the product catalog, choose items to add into a shopping cart, amend the shopping cart, order the items in the shopping cart, and modify user account information. The lines of code is about 2K. The application consists of four components including Account, Presentation, Catalog, and Order. In our experiment, these components are deployed on four different VMs hosted on Ubuntu15.10 server, and access database located at the fifth VM. The whole framework is shown in Fig. 4.

4.3 Measurement results

4.3.1 *Measurement results for RSS reader recipes application*

After this distributed application starts successfully, it always remains in running status. Users can do different operations through its web GUI. Driven by representative test scenarios, 250,000 traces are generated when the system lasts running for five hours. Here, we set $N_{Const} = 100$ to generate growing CN, and CN grows from CN-100 to CN-250,000. Figure 5 illustrates three Growing CNs including CN-100, CN-500, and CN-1000. Vertex is rendered with different color, and it means that a method belongs to a different component. Methods located on Edge, Middletier, and Eureka are separately rendered in blue, red, and yellow color. Each edge (it is not Edge component) means a unique method-method interaction. An intra-component interaction is rendered in green while inter-component one in black. The width of one edge is directly proportional to the weight of this edge. Here, the weight is the method calling frequency. The larger the weight is, the wider the edge is. There are two parameters in Metrics *CC* formulation, and we set $\alpha = 0.7$, $\beta = 0.3$. Metrics *CC* and *CP* are computed in each CN, and their changing charts of Growing CNs are shown in Figs. 6 and 7.

[4]https://scitools.com/

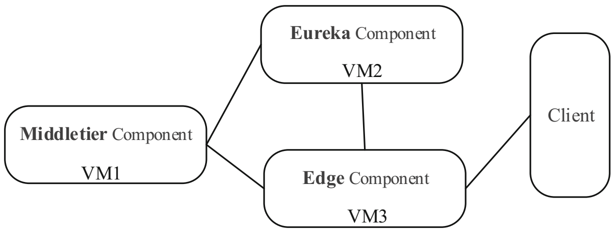


Fig. 3 The framework of *RSS Reader Recipes* application

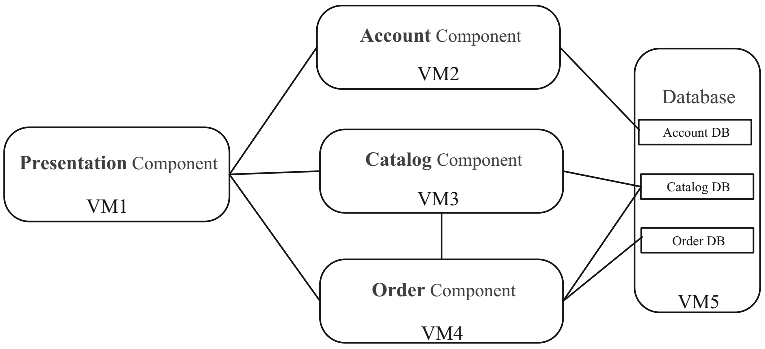


Fig. 4 The framework of distributed version of *iBATIS JPetStore* application

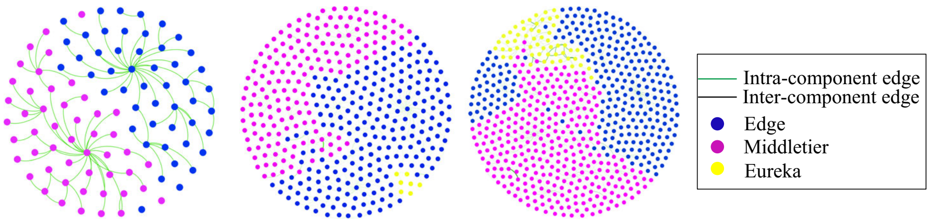


Fig. 5 An illustration of Growing CNs including CN-100, CN-500, and CN-1000

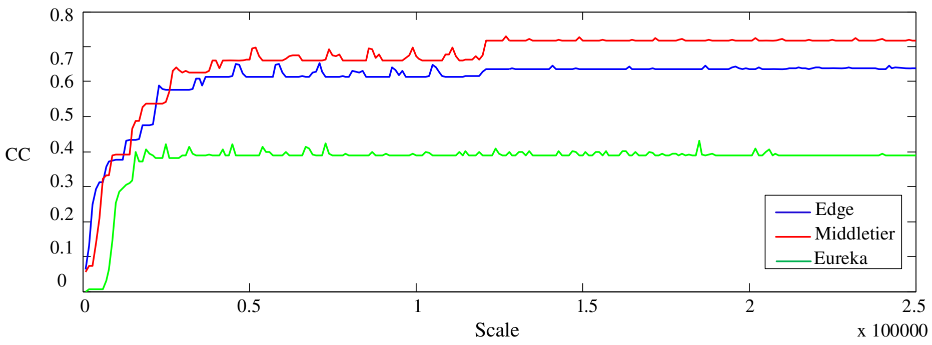


Fig. 6 *CC* value changing chart of Growing CN

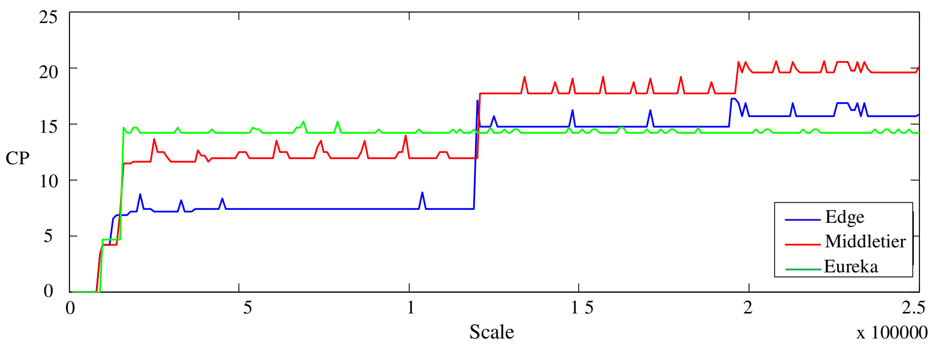


Fig. 7 *CP* value changing chart of Growing CN

In Figs. 6 and 7, it shows that CHC and CPC of each component tend to be in a steady state with the scale increase of growing CNs over time. Although there is a little fluctuation, the general trend is to be steady. The reason is that when a new software feature is invoked, the dynamic structure of the system is changed. After the major features are all covered, the dynamic structure tends to be stable. CHC and CPC tend to be stable after the number of traces reaches 1.25×10^5. We call this point as *stable point*. For each component, the mean and standard deviation of *CC* and *CP* after this *stable point* in Growing CNs are statistically computed with Wilcoxon signed-rank test for observation. *CC* results are shown in Table 3. $CP_{i,j}$ and CP_i results are shown in Table 4.

Tables 3 and 4 show *CC* and *CP* in Growing CNs present low standard deviation, which indicates that the data points tend to be close to the mean. Using the mean value, we observe CHC and CPC feature of all components in *RSS Reader Recipes* application. Figure 8 shows the overall results. It is discovered that the CHC of Eureka component is relatively lower than ones of Middletier and Edge component. Also, CPC between Middletier and Eureka is higher than that of other component pairs. Taking into consideration both CHC and CPC, Eureka is the most risky component who has lower structure quality (lower CHC and higher CPC) among all components in *RSS Reader Recipes application*. It implies that the subsequent structure refactoring works should focus more on Eureka component to improve its structure quality.

4.3.2 Measurement results for iBATIS JPetStore application

The *iBATIS JPetStore* Demo is an online pet store. Users can browse the product catalog, operating the shopping cart, order the items, and modify account information. After JPetStore application starts successfully, it remains in running status. Uses can do operations on it through web GUI. By implementing representative user scenarios, 7300 traces are generated when the system is in operation for 2 h. Because the scale of this application is smaller

Table 3 Statistical results of *CC*

CC	Edge	Middletier	Eureka
Mean	0.6302	0.6980	0.3910
Standard deviation	0.0110	0.0257	0.0066
P value	<0.02	<0.02	<0.02

Table 4 Statistical results of *CP*

CP	Mean	Standard deviation	*P* value
Edge-Middletier	7.3778	0.6978	<0.02
Edge-Eureka	8.1473	0.3480	<0.02
Middletier-Eureka	12.6764	0.7849	<0.02
Middletier-Edge	6.1605	0.4738	<0.02
Eureka-Edge	2.3373	0.0445	<0.02
Eureka-Middletier	11.8875	0.1364	<0.02
Edge	15.2848	0.7148	<0.02
Middletier	18.6834	1.0442	<0.02
Eureka	14.2380	0.1467	<0.02

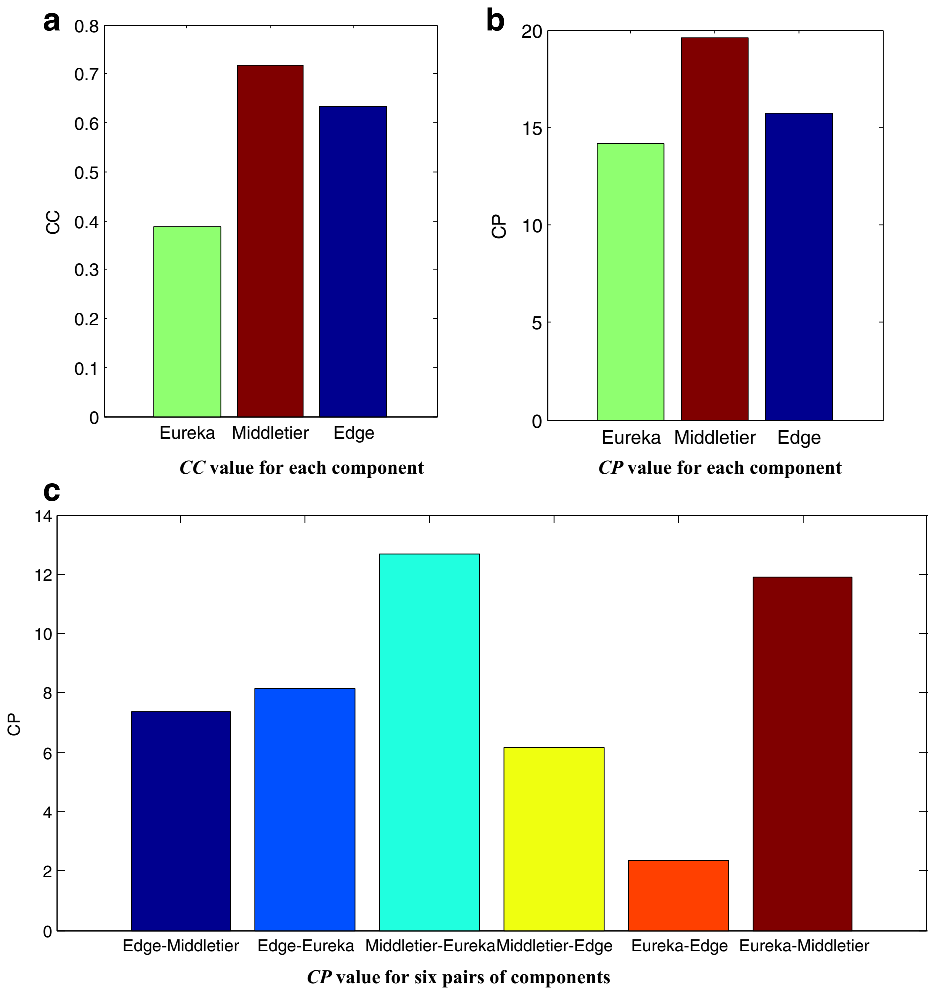


Fig. 8 *CC* and *CP* value of components in *RSS Reader Recipes* application

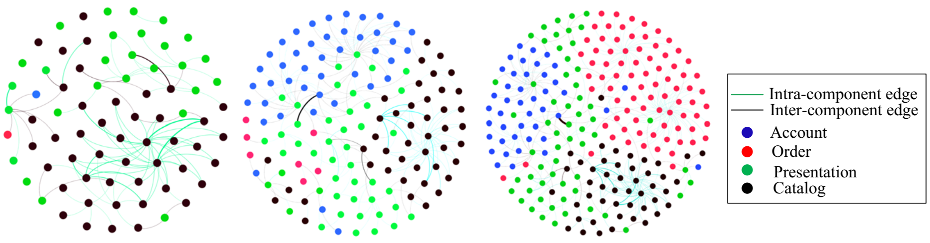


Fig. 9 An illustration of Growing CNs including CN-300, CN-1500, and CN-6000

than that of *RSS Reader Recipes*, the number of dynamic traces is also smaller. Here, we set $N_{Const} = 50$ generating growing CN. Figure 9 illustrates three Growing CNs. Methods located on Account, Order, Presentation, and Catalog component are separately rendered in blue, red, green, and black color. Each edge means method-method interaction. An intra-component interaction is rendered in blue while inter-component one in black. *CC* and *CP* are computed for each CN. There are two parameters in metrics *CC* formulation, and we set $\alpha = 0.7$, $\beta = 0.3$. Their changing charts of the Growing CNs are shown in Fig. 10.

It can be observed that CHC and CPC of each component tend to be in a stable state with the scale increase of growing CNs despite of a little fluctuation. The reason is that when a new software feature is invoked, the dynamic structure of the system is changed. After the major features are all covered, the dynamic structure tends to be stable. The *stable point* is the point where the number of traces reaches 3500. So for each component, the mean and

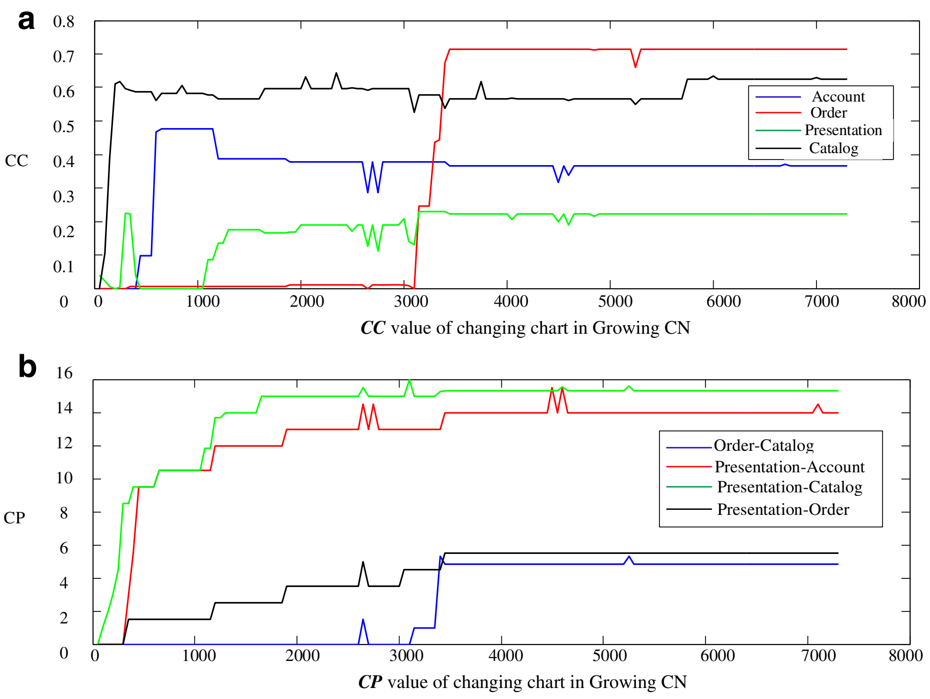


Fig. 10 The *CC* and *CP* changing chart of Growing CN

Table 5 *CC* statistical results

CC	Account	Order	Presentation	Catalog
Mean	0.3657	0.7144	0.2218	0.5918
Standard Deviation	0.0067	0.0063	0.0050	0.0295
P value	<0.02	<0.02	<0.02	<0.02

standard deviation of *CC* and *CP* after this point are statistically computed with Wilcoxon signed-rank test for observation. CC_i results are shown in Table 5 and $CP_{i,j}$ results in Table 6.

Tables 5 and 6 show *CC* and *CP* in Growing CNs present low standard deviation, which indicates that the data points tend to be close to the mean. Therefore, by using the mean value, we try to observe the CHC and CPC feature of all components in this distributed application. Figure 11 shows the overall results. It can been noticed that Presentation component has relatively lower CHC than other components. Also, CPC between Presentation and Account (or Catalog) is higher than that of other component pairs. By measuring both CHC and CPC, it is indicated that Presentation is the most risky one who has lower structure quality among all components in the distributed version of *iBATIS JPetStore* application. It suggests the structure refactoring works should focus more on Presentation.

4.3.3 Comparison and discussion of measurement results

Coupling is closely related to the dependency between software entities, but the existing coupling measurements are unable to capture the inter-component dependency, so we cannot do comparison with the existing coupling metrics. For cohesion, it is just dependent on the dependency inside one software entity, so we simply extend the existing class-level cohesion metrics into component level by using the average of class cohesion inside one component. Table 7 lists three existing cohesion metrics in detail for our comparison experiments. The measurement results are shown in Table 8. For CC, ICH, and Coh, the larger the value is, the more cohesive the software entity is. LCOM5 is opposite, and it measures the lack of cohesion. The value cells in italic highlight the components, which are identified as the relatively lowest cohesiveness by using different measurements.

Table 8 shows that Eureka and Presentation are in lower cohesiveness when measured by the proposed CHC and CPC, but other measurements are not. In *RSS Reader Recipes*, component Eureka is responsible for locating service component and load balancing,[5] so it performs more interactions across components than other components. In *iBATIS JPetStore*, component Presentation is responsible for directly processing users requests, then sends the data to other components including Catalog, Account, and Order for further process. So it performs more interactions across components than other components. Therefore, Eureka and Presentation should be in lower cohesion. It can be seen that the proposed measurements are consistent with function feature of components when compared with the existing ones. So, our methods are able to better measure the structure feature of components for distributed software. Through observing CHC and CPC attributes of all components, we can try to point out the potential components deviating from the general level in the same distributed software. Therefore, it can provide a view that helps developers focus on the relatively low-quality components in one distributed system.

[5]https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance

Table 6 *CP* statistical results

CP	Order-Catalog	Presentation-Account	Presentation-Catalog	Presentation-Order
Mean	4.8400	14.0461	15.3392	5.5
Standard deviation	0.0574	0.2474	0.0365	0
P value	<0.02	<0.02	<0.02	<0.02

Of course, this is just an initial experiment to apply our methods, and the number of components is limited. In order to observe the normal level of CHC and CPC attributes of components, we will do deeper analysis for more distributed software systems and help identifying the potential needed refactoring components. On the other hand, different kinds of software have different structure features and metric value distributions even for the same metric. Therefore, it is not simple to define a uniform threshold to generally assess whether a distributed system is good or bad-structure quality. This is also a common difficulty faced by the existing quality measurement works for single-server software.

4.4 Other results

In addition, CHC and CPC attributes also are measured based on Partitioned CN, and the characteristics are observed.

4.4.1 Measurement for RSS reader recipes application

The *RSS Reader Recipes* application provides users with three business functionalities, including getting, adding, deleting feeds. We use JMeter[6] to separately send Get, Add, and Delete requests concurrently in five times. So CN-Delete, CN-Add, CN-Get can be generated. Besides, CN-Start is generated during the period before the whole service is launched successfully, and CN-Idle is generated by lasting one minute during which the application does not have requests to process. Figure 12 illustrates the partitioned CNs. The color meaning of a vertex or an edge is same with that in Growing CN shown in Fig. 5. We measure CHC attribute by computing metric CC_i for each component. And we also measure CPC attribute by computing metric $CP_{i,j}$ for each pair of components. The results are listed in Tables 9 and 10.

In Fig. 12, it can be seen that the Eureka component does not participate in business functionality in most cases. So *CC* for Eureka in Table 9 is null (shown as " /"). They also show that CN-Delete, CN-Add, and CN-Get present similar CHC and CPC attributes. The reason is that these three CNs are generated from runtime traces representing similar business logic of the software system. Also, it is shown that *CP* value related with Eureka is relatively higher in CN-Start and CN-Idle than that in other CNs. This feature reflects that when software system is in starting and idle state, Middletier and Edge need to inform Eureka that they are alive at intervals.

4.4.2 Measurement for iBATIS JPetStore application

We design and implement four test functions, and four partitioned CNs can be obtained. CN-Account corresponds to the function that users can check and modify personal information.

[6]http://jmeter.apache.org

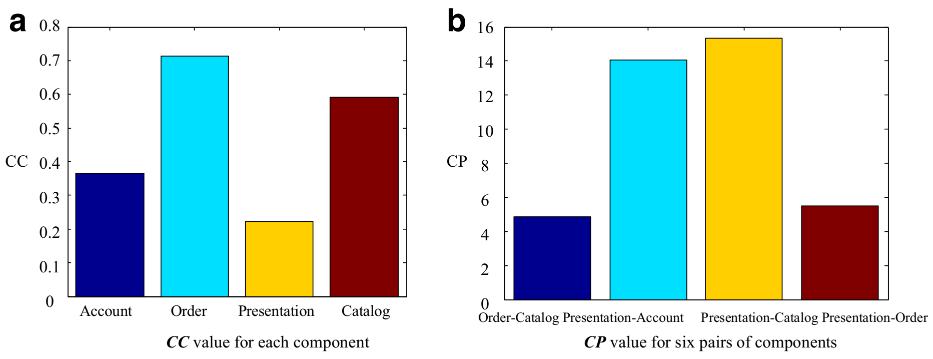


Fig. 11 *CC* and *CP* value for components in *iBATIS JPetStore* application

CN-Browse corresponds to that users can browse catalog and product items. CN-Cart corresponds to operations about shopping cart. CN-Order corresponds to that users can modify or submit the order. Figure 13 illustrates these partitioned CNs. The color of a vertex or an edge is same with that in Growing CN shown in Fig. 9. For each partitioned CN, CHC and CPC attributes are measured by computing CC_i and $CP_{i,j}$ metrics. The results are listed in Table 11.

In Fig. 13, it can be seen that different components dominate different Partitioned CNs. For example, Catalog component plays a major role in CN-Browse, while Account component plays a main role in CN-Account. Also, Table 11 shows that different partitioned CNs present different CHC and CPC feature. The reason is that the four CNs corresponds to different business logics.

5 Discussion

5.1 The limitation of dynamic measurement

The existing measurement methods include static and dynamic methods. Static metrics measure software systems based on information extracted from design document or source code.

Table 7 Cohesion metrics for comparison

Metric	Reference	Original form	Introduction
ICH	(Lee and Liang 1995)	$ICH^c(m) = \sum_{m' \in M}(1 + \lvert par(m')\rvert) \cdot NPI(m, m')$, $ICH(c) = \sum_{m \in M_I(c)} ICH^c(m)$, $ICH(SS) = \sum_{c \in SS} ICH(c)$	$NPI(m, m')$ is the calling times, $\lvert par(m')\rvert$ is the number of parameters of the calling method
Coh	(Briand et al. 1998)	$Coh = \frac{a}{kl}$	l is the number of class variable member, k is the number of class method member, a is the sum of the number of variable member referenced by one method member in this class
LCOM5	(Sellers 1995)	$LCOM5 = \frac{(kl-a)}{kl-l}$	a, k, l are same with those in Coh

Table 8 Meaurement results in comparison experiments

Metrics	CC	ICH	ICH-Avg	Coh-Avg	LCOM5-Avg
RSS reader recipes					
Edge	0.6302	45	1.9565	0.1670	1.0490
Middletier	0.6980	104	*1.8909*	0.2110	*1.1090*
Eureka	*0.3910*	4939	3.0582	*0.1540*	0.9440
iBATIS JPetStore					
Order	0.7144	24	1.0435	*0.0725*	0.7350
Catalog	0.5918	6	0.25	0.2920	0.6820
Presentation	*0.2218*	66	1.1786	0.0885	0.8450
Account	0.3657	5	*0.2174*	0.1900	*1.1546*

The value cells in italic highlight the components, which are identified as the relatively lowest cohesiveness by using different measurements

Dynamic metrics measure software quality using the information from executing traces. Compared with static ones, dynamic methods have expensive cost of dynamic analysis and complexity to perform (Arisholm et al. 2002; Geetika and Singh 2014). On the one hand, the software system need to be instrumented in a manual way or in an automatically monitoring way. Since it is impossible to instrument all the methods in a large-scale distributed systems, the instrumented scope need be carefully chosen to make sure that the major method calling behaviors can be obtained. In our case studies, the line of code of *RSS Reader Recipes* application is about 20K, so the methods of this application are partially instrumented by covering the major functions. *iBATIS JPetStore* application is about 2K, so we cover all the methods. On the other hand, test cases or test scenarios must be designed to drive the target software systems to generate the representative data. In our case studies, the two distributed systems expose their function features through GUI. It is convenient to design test scenarios. In general, when measuring other various distributed systems, the instrumented scope and test cases (or test scenarios) need to be selected according to the specific features of target applications.

5.2 The limitation of collecting dynamic data

Due to the distributed nature of distributed software, the inter-component interactions or dependencies cannot be captured by source code analysis. In this paper, we use a monitoring tool-Kieker to obtain the dynamic execution traces of an instrumented distributed software system. The intra-component and inter-component interactions can be extracted accordingly. This tool provides probes for collecting distributed traces in REST-based

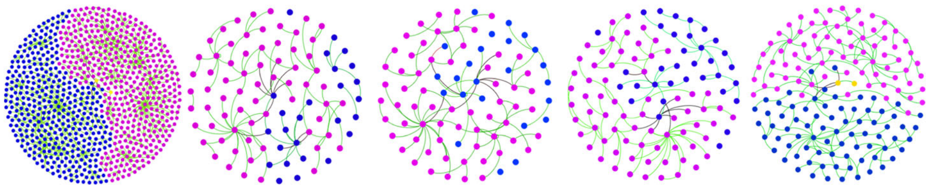


Fig. 12 An illustration of Partitioned CNs including CN-Start, CN-Add, CN-Get, CN-Delete, and CN-Idle

Table 9 *CC* results of partitioned CN

Partitioned CN	Middletier	Edge	Eureka
CN-Start	0.6847	0.6211	/
CN-Delete	0.5324	0.3517	/
CN-Add	0.5523	0.3252	/
CN-Get	0.5916	0.3530	/
CN-Idle	0.6726	0.6738	/

environments with Jersey. So we just collect data on the distributed systems which employ Jersey communication framework. In real world applications, there are all kinds of communication paradigms including inter-process communication, remote invocation, and indirect communication (Coulouris et al. 2012). Each paradigm can be implemented by employing different communication techniques. Different communication techniques use different data type and communication protocol. In consequence, the tool used for collecting data in this paper maybe is limited for monitoring various distributed systems. Except this constraint, our model and measurement methods are general and can be extended to other distributed systems.

6 Related works

6.1 Cohesion measurement for single-server software

The cohesion measures mainly include static and dynamic (or runtime) cohesion metrics. Static cohesion is measured based on information extracted from design document or source code. Dynamic cohesion is measured based on runtime information extracted from execution traces of software. A group of test cases or user scenarios are designed in order to drive the software to generate execution traces for dynamic measurement.

For static cohesion measurement, Chidamber and Kemerer (1994) proposed the lack of cohesion in methods metric (LCOM). Briand et al. (1999) and Counsell et al. (2006) proposed cohesion metrics based on information available in high-level design phase. Dallal and Briand (2012) proposed a class cohesion metric (LSCC) based on the degree of interaction between each pair of methods from source code. It also verified LSCC usefulness in improving class cohesion. Qu et al. (2015a) proposed cohesion metric MCC and MCEC abstracted from software source code. MCC and MCEC are measured based on community structure of software system.

Table 10 *CP* results of Partitioned CN

Partitioned CN	Middletier-Edge	Edge-Middletier	Eureka-Middletier	Middletier-Eureka	Eureka-Edge	Edge-Eureka
CN-Start	/	/	/	5.0363	1.5	4.3461
CN-Delete	2.0833	4	/	/	/	/
CN-Add	2.0833	4	/	/	/	/
CN-Get	2.0833	3.5	/	/	/	/
CN-Idle	/	/	/	3.4	1.5	2.9600

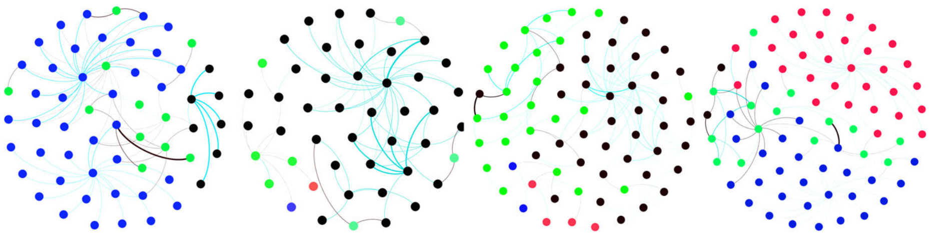


Fig. 13 An illustration of Partitioned CNs including CN-Account, CN-Browse, CN-Cart, and CN-Order

Yacoub et al. (1999), Tahir and Macdonell (2012), and Arisholm et al. (2002) distinguished static and dynamic metrics. For dynamic cohesion, Mitchell and Power (2004) proposed two run-time cohesion metrics RLCOM and RWLCOM. Both are direct extensions of LCOM. RLCOM measures the count of instance variables which are actually accessed at run-time, while RWLCOM measures cohesion by weighting each instance variable by the number of times it is accessed at run-time. Gupta and Chhabra (2011) proposed dynamic cohesion metrics at object level and experiments suggested the proposed ones can better capture the dynamic information. Mathur et al. (2011) defined cohesion RuCIVA similar to RLCOM (Mitchell and Power 2004). Desouky and Etzkorn (2014) proposed metric RLCOM-DESOUKY, an extension to RuCIVA (Mathur et al. 2011). It presented an empirical comparison with the existing runtime cohesion metrics suggested by Mitchell and Power (2004) and Mathur et al. (2011) in one case.

These existing structure measurement using static and dynamic metrics are almost only validated on single-server software at class level or object level. They could not observe and assess unique characteristics of distributed software at component level, which is our focus point.

6.2 Coupling measurement for single-server software

Different type of information relevant to software can capture and reflect different view of software attributes (Bavota et al. 2013). Similar to cohesion metrics, the existing coupling metrics also mainly include static and dynamic (or runtime) ones.

For static coupling, the most influential ones are coupling between object classes (CBO) and response for a class (RFC), proposed by Chidamber (1991) and Chidamber and Kemerer (1994). CBO is defined as the count of classes that a class referenced plus the count of classes that referenced the class. RFC is the size of the Response set of a class. The Response

Table 11 Measurement results of Partitioned CN

Partitioned CN	CC				CP		
	Account	Order	Presentation	Catalog	Presentation-Account	Presentation-Order	Presentation-Catalog
CN-Account	0.3919	/	/	0.6	10.5	null	2
CN-Browse	/	/	/	0.6421	/	/	3
CN-Cart	/	/	0.5043	0.6263	1	1.5	9.8333
CN-Order	0.4316	0.6586	0.1461	/	7.5	3.5	2.8333

set for a class is a set of methods that can potentially be executed in response to a message received by an object of that class. Fan In, Fan Out, Efferent Coupling (Ce) and Afferent Coupling (Ca) are similar metrics to the typical CBO (Elish 2010). Briand et al. (1999) designed a unified framework for static coupling measurement. Allen et al. (2001) defined coupling metrics by applying entropy and information measurement in information theory.

For dynamic coupling, Yacoub et al. (1999) defined two dynamic object-level coupling metrics export object coupling (EOC) and import object coupling (IOC), which were computed based on the count of exchanged messages during execution scenarios. Mitchell and Power (2004) extended CBO and validated the proposed dynamic coupling using SPEC JVM98 benchmark. Arisholm et al. (2002) conducted empirical evaluation of the proposed dynamic coupling measures and showed they complement existing static coupling metrics. Most of proposed dynamic coupling metrics have not yet been empirically validated due to the expensive cost of dynamic analysis and complexity to perform (Geetika and Singh 2014; Arisholm et al. 2002).

In addition to static and dynamic coupling above, there are also works measuring coupling based on other information relative to other software artifacts. Some works compute the coupling based on the semantic information obtained from the source code, encoded in identifiers and comments, such as Poshyvanyk and Marcus (2006), Poshyvanyk et al. (2009), and Gethers and Poshyvanyk (2010). Other works (Ying et al. 2004) measure the coupling of relation of software elements by analyzing the commit log history.

All these coupling metrics are almost only validated on single-server software at class level or object level, while our works investigate CHC and CPC attributes specific for distributed software.

6.3 Cohesion and coupling for SOA

Recently, there are some relevant works focusing on service-oriented architecture (SOA), which is one special type of distributed architecture from service view. These works research about cohesion and coupling of service for SOA software, and also can be categorized into static and dynamic ones.

For static metrics, Perepletchikov and Ryan (2011) investigated the impact of static coupling (Perepletchikov et al. 2007), and the results indicated a statistically significant causal relationship between the coupling metrics and the maintainability of software in service level for SOA software. Athanasopoulos et al. (2014) proposed a suite of cohesion metrics which only used interface specification. The proposed cohesion metrics were used to decompose a given interface into more cohesive interface to improve the service interface design quality. It can be seen these static metrics are based on design document, particularly web service design language (WSDL)-based document. Their work can be done in early design stage of SOA software development, and aims at improving service interface design quality, which are different with our works.

For dynamic metrics, Nayrolles et al. (2013) measured cohesion and coupling based on mining execution traces. For SOA software, they hypothesized that "If the number of different methods of a service A is equal or superior to the number of different services invoking A, the service is not externally cohesive; If a service appears in the consequent (antecedent) parts for a high number of associations, then it has high incoming (outgoing) coupling." These dynamic ones are based on causal relation of services by mining execution traces, while our work use the method calling behaviors instead of using service relation by data mining.

7 Conclusion and future work

In this paper, we aim at measuring structure quality specific for distributed software. To model distributed software, we have extended the existing CN model, generating Growing CN and Partitioned CN by using different schemas. Two structure quality attributes CHC and CPC are defined for distributed software. Then in order to quantitatively measure CHC and CPC attributes, two dynamic metrics *CC* and *CP* are formulized based on our models. These metrics have been proven to meet the mathematical properties. Finally, two case studies are conducted on real-world open-source distributed software systems. The studies show that dynamic metrics *CC* and *CP* can present structure attributes for distributed software. The measuring results are consistent with the expected features of distributed software. It also has been observed that CHC and CPC attributes of Growing CNs tend to be stable with a scale increase of Growing CNs over time. In addition, it has been discovered that the partitioned CNs with similar business functionality present similar CHC and CPC features.

Our work is just the first step toward future structure optimization for distributed software, and it aims at providing a quantitative and objective approach for guiding and assessing the subsequent structure refactoring or optimizing process. Of course, there is still a lot of practical work to do in the future. On the one hand, there are fewer publicly available distributed systems than single-server software systems for research purposes. We will try to collect more open source systems for supporting studies on distributed software. On the other hand, we will conduct comprehensive experiments and try to apply the proposed methods to structure re-factoring work for distributed software. In particular, since CHC and CPC attributes are relevant to the function role of a component, we will investigate how to guide the good-quality transformation from legacy single-server software to distributed software by considering the role of components. Also, structure quality is just one view of measuring good-quality distributed systems, and performance is a more objective perspective. Therefore, in the future structure optimization work, both structure quality and performance can be considered during the process.

Acknowledgements This work was supported by the National Natural Science Foundation of China (91218301, U1301254, 91418205, 61472318, 61428206, 61532015, 61602369), Fok Ying-Tong Education Foundation (151067), Key Project of the National Research Program of China (2013BAK09B01), Ministry of Education Innovation Research Team (IRT13035), and the Fundamental Research Funds for the Central Universities.

References

Al Dallal, J. (2010). Mathematical validation of object-oriented class cohesion metrics. *International Journal of Computers*, *4*(2), 45–52.

Allen, E.B., Khoshgoftaar, T.M., & Chen, Y. (2001). Measuring coupling and cohesion of software modules: an information-theory approach. In *International symposium on software metrics* (p. 124).

Arisholm, E., Briand, L.C., & Foyen, A. (2002). Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, *30*(8), 33–42.

Athanasopoulos, D., Zarras, A., Miskos, G., Issarny, V., & Vassiliadis, P. (2014). Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Transactions on Services Computing*, *8*(4), 1–1.

Bavota, G., Dit, B., Oliveto, R., Penta, M.D., Poshyvanyk, D., & Lucia, A.D. (2013). An empirical study on the developers' perception of software coupling. In *International conference on software engineering* (pp. 692–701).

Bieman, J.M., & Ott, L.M. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering*, *20*(8), 644–657.

Briand, L.C., Daly, J.W., & Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, *3*(1), 43–53.

Briand, L.C., Morasca, S., & Basili, V.R. (1999). Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, *25*(5), 722–743.

Cai, H., & Thain, D. (2016). Distia: a cost-effective dynamic impact analysis for distributed programs. In *IEEE/ACM international conference on automated software engineering* (pp. 344–355).

Chidamber, S.R. (1991). Towards a metrics suite for object oriented design. In *Proceedings of the conference on OOPSLA'91, Sigplan Notices*.

Chidamber, S.R., & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

Coulouris, G., Dollimore, G., Kindberg, J., & Blair, T. (2012). Distributed systems: concepts and design (5th edition).

Counsell, S., Swift, S., & Crampton, J. (2006). The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology*, *15*(2), 123–149.

Dallal, J.A., & Briand, L.C. (2012). A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology*, *21*(2), 1–34.

Desouky, A.F., & Etzkorn, L.H. (2014). Object oriented cohesion metrics: a qualitative empirical analysis of runtime behavior. In *ACM southeast regional conference* (pp. 1–6).

Elish, M.O. (2010). Exploring the relationships between design metrics and package understandability: a case study. In *The 18th IEEE international conference on program comprehension, ICPC 2010, Braga, Minho, Portugal, June 30–July 2, 2010* (pp. 144–147).

Geetika, R., & Singh, P. (2014). Empirical investigation into static and dynamic coupling metrics. *ACM SIGSOFT Software Engineering Notes*, *39*(1), 1–8.

Gethers, M., & Poshyvanyk, D. (2010). Using relational topic models to capture coupling among classes in object-oriented software systems. In *26th IEEE international conference on software maintenance (ICSM2010)* (pp. 1–10).

Gupta, V., & Chhabra, J.K. (2011). Dynamic cohesion measures for object-oriented software. *Journal of Systems Architecture - Embedded Systems Design*, *57*(4), 452–462.

Hoorn, A.V., Waller, J., & Hasselbring, W. (2012). Kieker: a framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC international conference on performance engineering* (pp. 247–248).

Indrajit Wijegunaratnec, M., & Fernandez, G. (1998). *Distributed applications engineering*. London: Springer.

Jin, W., Liu, T., Qu, Y., Chi, J., Cui, D., & Zheng, Q. (2016). Dynamic cohesion measurement for distributed system. In *The international workshop on specification, comprehension, testing, and debugging of concurrent programs* (pp. 20–26).

Lee, Y.S., & Liang, B.S. (1995). Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proceedings of international conference on software quality*.

Lin, Y., Peng, X., Cai, Y., Dig, D., Zheng, D., & Zhao, W. (2016). Interactive and guided architectural refactoring with search-based recommendation. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering (FSE)*.

Marwede, N., Rohr, M., Hoorn, A., & Hasselbring, W. (2009). Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In *European conference on software maintenance and reengineering* (pp. 47–58).

Mathur, R., Keen, K.J., & Etzkorn, L.H. (2011). Towards a measure of object oriented runtime cohesion based on number of instance variable accesses. In *Southeast regional conference 2011, Kennesaw, GA, USA* (pp. 255–257).

Mitchell, Á., & Power, J.F. (2004). An empirical investigation into the dimensions of run-time coupling in Java programs. In *Proceedings of the 3rd international symposium on principles and practice of programming in Java* (pp. 9–14). Trinity College Dublin.

Nayrolles, M., Moha, N., & Valtchev, P. (2013). Improving soa antipatterns detection in service based systems by mining execution traces. In *Working conference on reverse engineering (WCRE)* (pp. 321–330).

Nguyen, H., Shen, Z., Tan, Y., & Gu, X. (2013). Fchain: toward black-box online fault localization for cloud systems. In *2013 IEEE 33rd international conference on distributed computing systems (ICDCS)* (Vol. 7973, pp. 21–30).

Perepletchikov, M., & Ryan, C. (2011). A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *IEEE Transactions on Software Engineering*, *37*(4), 449–465.

Perepletchikov, M., Ryan, C., Frampton, K., & Tari, Z. (2007). Coupling metrics for predicting maintainability in service-oriented designs. In *Australian software engineering conference* (pp. 329–340).

Poshyvanyk, D., & Marcus, A. (2006). The conceptual coupling metrics for object-oriented systems. In *IEEE international conference on software maintenance* (pp. 469–478).

Poshyvanyk, D., Marcus, A., Ferenc, R., & Gyimóthy, T. (2009). Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, *14*(1), 5–32.

Qu, Y., Guan, X., Zheng, Q., Liu, T., Wang, L., Hou, Y., & Yang, Z. (2015a). Exploring community structure of software call graph and its applications in class cohesion measurement. *Journal of Systems and Software*, *108*, 193–210.

Qu, Y., Guan, X., Zheng, Q., Liu, T., Zhou, J., & Li, J. (2015b). Calling network: a new method for modeling software runtime behaviors. *ACM SIGSOFT Software Engineering Notes*, *40*(1), 1–8.

Sellers, B.H. (1995). *Object-oriented metrics measures of complexity*. Prentice-Hall Inc.

Stevens, W.P., Myers, G.J., & Constantive, L.L. (1974). Structured design. *Ibm Systems Journal*, *13*(2), 115–139.

Tahir, A., & Macdonell, S.G. (2012). A systematic mapping study on dynamic metrics and software quality. In *28th IEEE international conference on software maintenance, 2012* Vol. 9, no. 3, pp. 326–335.

Tanenbaum, A.S., & Steen, M.V. (2002). *Distributed systems: principles and paradigms* (pp. 279–283). Tsinghua University Press.

Thones, J. (2015). Microservices. *IEEE Software*, *1*, 116–116.

Tian, Z., Liu, T., Zheng, Q., Zhuang, E., Fan, M., & Yang, Z. (2017). Reviving sequential program birthmarking for multithreaded software plagiarism detection. *IEEE Transactions on Software Engineering*. http://ieeexplore.ieee.org/abstract/document/7888597/.

Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., & Yang, Z. (2017). Dependence guided symbolic execution. *IEEE Transactions on Software Engineering*, *43*(3), 252–271.

Yacoub, S.M., Ammar, H.H., & Robinson, T. (1999). Dynamic metrics for object oriented designs. In *Proceedings of the 6th international software metrics symposium, 1999* (pp. 50–50).

Ying, A.T.T., Murphy, G.C., Ng, R., & Chu-Carroll, M.C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, *30*(9), 574–586.

Zhou, Y., Lu, J., & Xu, H.L.B. (2004). A comparative study of graph theory-based class cohesion measures. *ACM SIGSOFT Software Engineering Notes*, *29*(2), 13–13.

Wuxia Jin is a Ph.D. candidate studying in Xi'an Jiaotong University, Xi'an, China. Her major is Computer Science and technology. Her research interests include trustworthy software and software analysis especially for distributed software system.

Ting Liu received his B.S. degree in information engineering and Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, Xi'an, China, in 2003 and 2010, respectively. Currently, he is an associate professor of the Systems Engineering Institute, Xi'an Jiaotong University. His research interests include Smart Grid, network security and trustworthy software.

Yu Qu received the B.S. and Ph.D. degrees from Xi'an Jiaotong University, Xi'an, China in 2006 and 2015 respectively. He is a post-doctoral researcher at the Department of Computer Science and Technology, Xi'an Jiaotong University. His research interests include trustworthy software and applying complex network and data mining theories to analyzing software systems.

Qinghua Zheng received the B.S. degree in computer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xi'an Jiaotong University, China. He was a postdoctoral researcher at Harvard University in 2002. He is currently a professor in Xi'an Jiaotong University, and the dean of the Department of Computer Science. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.

Di Cui is a Ph.D. candidate studying in Xi'an Jiaotong University, Xi'an, China. His major is Computer Science and technology. Her research interests include trustworthy software, architecture recovery of software system.

Jianlei Chi is currently a Ph.D. student in Software Engineering at Xi'an Jiaotong University. His research is focused on software analyzing, especially on test case evolving and prioritization.

View publication stats