Calling Network: A New Method for Modeling Software Runtime Behaviors

Yu Qu 
yqu@sei.xjtu.edu.cn

Xiaohong Guan 
hxguan@sei.xjtu.edu.cn

Qinghua Zheng 
qhzheng@mail.xjtu.edu.cn

Ting Liu* 
tliu@sei.xjtu.edu.cn

Jianliang Zhou 
jlzhu@sei.xjtu.edu.cn

Jian Li 
jianlee@stu.xjtu.edu.cn

MOE Key Lab for Intelligent Networks and Network Security, Xi’an Jiaotong University
Xi’an, China

ABSTRACT

Modern software systems’ structures and behaviors are becoming very complicated. Existing models either lack systematic considerations on the whole system’s behavior patterns or are inefficient in runtime monitoring. In this paper, the Calling Network (CN) model is proposed to provide new perspectives to analyze the dynamic execution process of a software system. CN is consisted of one or a series of Calling Graph (CG), which is a dynamic version of Call Graph and encodes method call frequencies. Some new perspectives such as Growing Network and Network (Graph) Sequence are also embodied in CN model. Based on a data set of 10 real-world Java programs, we show that CN presents several interesting features, such as Power-law degree distribution, Densification Power Law, and the stability of an entropy value – Local Entropy. Experiments have been conducted to show the applications of CN in software significant module identification and runtime failure diagnosis.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, software science

General Terms

Theory, Measurement

Keywords

Software Model, Software Behavior, Method Call

1. INTRODUCTION

Software systems might be one of the most complex systems made by human being. Many models have been proposed to understand and describe the structures and behaviors of software systems. These models can be summarized from the following two perspectives.

On one hand, software models in the form of Finite State Automata [19] or Program Invariants [9] have been intensively used in software engineering domain. But these models’ practical applications are often hindered by the enormously large state space of software. For example, the Model Checking techniques [6] often have to face the state space explosion problem, and Dynamic Symbolic Execution techniques [12] are suffered from the path explosion problem. In a word, these models and techniques are very useful in software engineering, but they need the help provided by systematic considerations on the behavior patterns and dynamics of the whole system.

On the other hand, with the progress of Complex Network (also referred to Complex System) theory and graph mining algorithms in recent years, these theories have been intensively applied to the quantitative analysis of software systems. This is an interdisciplinary research area. Existing methods in this area have successfully applied the Complex System theory and graph algorithms to software evolution prediction [4], software community, controllability, vulnerability analysis [21, 25, 26], software structure interpretation and evaluation [3, 20, 31], etc. However, existing methods suffer from the following problem:

Most of the networks [3, 4, 20, 21, 25, 26, 27, 30] are constructed by statically analyzing the source code of a software system. Thus, these networks are not applicable in testing, monitoring and maintenance processes, which take important roles in a software system’s life cycle. Moreover, modern software design patterns and practices make it sometimes impossible to capture all the prerequisite information to construct a complete network only by static analysis. Take the Inversion of Control (IoC) [13] mechanism of the popular J2EE framework – Spring, for example. Figure 1 is an excerpt taken from a J2EE benchmark web application – JPetStore’s “applicationContext.xml” file, which is a standard configuration file of the Spring framework.

This excerpt defines that in the source files of JPetStore, all the class transactionManager is actually the real implementation class DataSourceTransactionManager. It can also be linked to another implementation class just by changing this figuration file. By using this flexible mechanism, the Spring framework decouples the interconnections between classes, making it impossible to construct the network by only analyzing the system’s source code.

Network and graph models are inherently suitable for software systems. Different methods/functions, classes, modules and components are the basic ingredients which are combined together to form a complete system.

As shown in Figure 2, during the execution process of a software system, its method call behaviors are recorded using instrumentation frameworks. Calling Network (CN) is constructed to model these method call behaviors and relationships. CN is consisted of

http://code.google.com/p/mybatis/

1In this paper, we use software interchangeably with program, and method interchangeably with function.
one or a series of Calling Graph (CG). Three computing schemes are proposed to generate CN. CN has the following characteristics and advantages:

Firstly, CN has Power-law degree distribution, and is a typical Small-world and Scale-free network (Scheme 1 in Figure 2). These findings have laid a good foundation for further applying Complex Network theory to evaluating software dynamic structure [31] and behavior. Moreover, as CN is constructed dynamically, it is directly related to a certain functionality of a system. In experiments, we show that node centrality metrics in Complex Network theory are useful in identifying significant modules for a web forum system’s posting functionality.

Secondly, this paper propose to model the dynamic method calls from a Growing Network perspective. As shown in Scheme 2 in Figure 2. A very interesting phenomenon is observed: it is discovered that all the software’s CNs obey the same growing pattern – Densification Power Law. It is the first time to report such phenomenon in software domain.

Thirdly, we propose to partition the original CN into a sequence of CGs, as shown in Scheme 3 in Figure 2. It is discovered that the Local Entropy, which is a quantification method in Complex Network theory, exhibits very stable nature in such Calling Graph sequence. Local Entropy is helpful in software runtime fault diagnosis and localization.

Finally, all the aforementioned methods and perspectives are included in a single and integrated model, which is complementary to existing models.

In summary, this paper makes the following contributions: 1) a new and systematic model, CN, of software runtime method calls is proposed. 2) three schemas are proposed to generate CN, then several interesting features of CN are observed, such as Power-law degree distribution, Densification Power Law and the stability of Local Entropy in CN. 3) Experiments have been conducted to show the application of CN in performance optimization and runtime failure diagnosis.

The rest of this paper is organized as follows. In Section 2, the definition of CG is introduced. A formal definition of CN is given, then three CN generation schemes are discussed. In Section 3, based on 10 real-world Java programs, the measurements, discussions and applications of CN model and its generation schemes are discussed. Section 4 gives the conclusion and future work.

2. CALLING NETWORK MODEL

2.1 Calling Graphs and Other Networks

CN is consisted of one or a series of CG. In this section, an illustrative example is given to describe the differences between CG and other networks proposed in previous related works. Figure 3 (a) is a simple example code snippet in Java syntax.

Most of the networks previously proposed are constructed by statically analyzing the source code of the target system. For the given code snippet, the networks in Figure 3 (b) and Figure 3 (c) are constructed statically. Network in Figure 3 (b) is the Class Dependency Network [25, 26]. Although other literatures [3, 20, 30] usually only gave methods to construct the networks, the intrinsic nature of these networks is same to the Class Dependency Network’s. In these networks, nodes are classes and edges represent relationships between classes. These relationships include aggregation (A→B, A→C, and B→D), inheritance (D→C) and interface implementation, and return / parameter types. The Class Collaboration Graph [21] is similar to Class Dependency Network except that the return / parameter type relationships are not considered. Network in Figure 3 (c) is the Class Graph [27], which only considers the inheritance relationships and is a simplified version of the Class Dependency Network. Figure 3 (d) shows the Object Graph [24], which is constructed dynamically. Each node in Object Graph represents an object created during program execution, and the edges are using relationships between objects (b:B→d:D). The nodes c:C and c2:C are isolated because they are used by the static main() method rather than an object.
has a weighting function $E$.

**Definition 2.** Descriptive, Param $\rightarrow$ 5.

CB $\rightarrow$ Definition 4. Weight mainly represents method call frequency. It can represent different computation schemas. In this paper, edge weights in Figure 3 (e) represent method call frequencies during CG generation function $f_{CG-Gen}$: CB $\rightarrow$ CG.

**Definition 5.** Calling Network is an ordered set of CG: $CN = \{CG_i | i \in N\}$, where $CG_i = f_{CG-Gen}(CB_i)$, $CB_i \subseteq CB$.

$CB$ is partitioned into $CB_i$ using some strategies. The 2 following strategies are discussed in this paper. Other strategies can also be included in the model.

**Strategy 1.** uses fixed interval and quantity of $cb$s to generate CG. Two parameters are set in this strategy: $N_{Itv}$ and $N_{CG}$, represent interval between two consecutive $CB$s, and the number of $cb$s in each $CB_i$, respectively. Then $CB_i$ is expressed as: $CB_i = \{cb_k | (i-1) \cdot N_{Itv} \leq k \leq (i-1) \cdot N_{Itv} + N_{CG}\}$.

**Strategy 2.** uses time step to partition $CB$. Two parameters are set in this strategy: $T_i$ and $\Delta t$. $T_i$ is the $i$-th time point, $\Delta t$ is the time window to select $cb$s to construct $CB_i$. Then $CB_i$ is expressed as: $CB_i = \{cb_k | T_{i-1} - \Delta t \leq k \leq T_i\}$.

In summary, the Calling Network model is formalized as (Strategy 1 is included):

\[
\begin{align*}
CN & = \{CG_i | i \in N\}, \\
CG_i & = f_{CG-Gen}(CB_i), CB_i \subseteq CB \text{ and } \\
CB_i & = \{cb_k | (i-1) \cdot N_{Itv} \leq k \leq (i-1) \cdot N_{Itv} + N_{CG}\}, \\
CG & = (V,E), w : E \rightarrow N, \\
CB & = \{cb_k | k \in N\}, \\
\end{align*}
\]

\[\text{(1)}\]

**2.3 CN Generation Schemes**

Based on Equation 1, three CN generation schemes are proposed by setting different values of $N_{Itv}$ and $N_{CG}$:

**2.3.1 Raw Calling Network (Raw CN)**

Setting $N_{Itv} = 0$ and $N_{CG} = |CB|$ in CN model. Then there is only one $CG$ in $CN$, denoted as $CG_\infty$. Scheme 1 in Figure 2 depicts this Raw CN of Makagiga\(^3\), a personal information management system. $CG_\infty$ is constructed based on a $CB$ whose cardinality is 324928 ($|CB|$ = 324928), which contains 2358 nodes and 5540 edges.

**2.3.2 Growing Calling Network (Growing CN)**

Assuming $N_{Itv} = 0$ and $N_{CG} = i \cdot N_{Const}$, where $N_{Const}$ is a constant value, which means $CB_i = \{cb_k | 0 \leq k \leq i \cdot N_{Const}\}$. It can be noticed that this sequence of $CG$s represents the growing growth.

\[^3\text{http://sourceforge.net/projects/makagiga/}\]
process of Calling Graph over time. Scheme 2 in Figure 2 shows Growing CN of Makagiga.

2.3.3 Partitioned Calling Network (Partitioned CN)
Letting \( N_{1tv} \) and \( N_{CG} \) be some non-zero values, then Partitioned CN is derived. Scheme 3 in Figure 2 illustrates the Partitioned CN of Makagiga, in which \( N_{1tv} = N_{CG} = 2000 \), means that \( CB_i = \{ cb_k | (i-1) \cdot 2000 \leq k \leq i \cdot 2000 \} \). Considering \(|CB| = 324928\), then a CN contains 163 CGs is derived (\(|CN| = 163\)). The 1st, 40th, 80th and the 160th CGs in this CN are shown.

3. EXPERIMENTS, MEASUREMENTS, ANALYSIS AND APPLICATIONS OF CN

3.1 Data Set and Experiment Implementation
A data set including 10 real-world open-source Java programs are collected, as shown in Table 1. Corendal Wiki is a intranet wiki application. DLOG4J is a web blog system based on JSP and Servlet. Endeavour is a software project management system. FreeMind is a mind-mapping software. JForum is a discussion board system. JPetStore is a J2EE demo system introduced in Section 1. Kunagi is a software project management system based on the agile framework Scrum. LogicalDOC is a document management system. Makagiga is a personal information management system introduced in Section 2. OpenKM is a multi-platform application for document management.

These programs exhibit a strong heterogeneity in their sizes, design principles and functionalities. Most of these programs are popular and widely used in practice (except for JPetStore, which is a standard demo system developed by MyBatis and it also has been widely used in researches as a benchmark system [15]). For example, Endeavour has been downloaded for 73,637 times in SourceForge since April 2nd, 2009.

Table 1 shows basic information of these programs. For each program, the second column shows the program’s architecture (Arch) (Web or desktop), the third column provides the version number. The 4th to 6th columns give the value of Static Lines of Code (SLoC) and the number of classes, packages, and methods respectively. The last column gives the cardinality of CB, which will be explained later.

To construct CGs and CN, dynamic execution traces of the target programs have to be collected. The Kieker framework [28], which is an open-source software dynamic behavior monitoring framework based on AspectJ [16], was used and re-developed as the instrumentation framework. The open-source Python software NetworkX, which is a package for the computation of the structure, dynamics, and functions of complex networks, was re-developed as the main tool to conduct network data analysis.

4http://sourceforge.net/projects/corendalwiki
5http://sourceforge.net/projects/dlog4j/
6http://sourceforge.net/projects/endeavour-mgmt/
7http://sourceforge.net/projects/freemind/
8http://jforum.net/
9http://kunagi.org/
10http://www.scrumalliance.org/
11http://www.logicaldoc.com/
12http://www.openkm.com/
13http://blog.mybatis.org/
14http://sourceforge.net/
15http://kieker-monitoring.net/
16http://networkx.github.io/

To generate CBs of these programs, different usage scenarios, user inputs and test cases were designed according to these programs’ functionalities and documents. Then these test cases were used to drive the programs to generate CBs. The last column of Table 1 lists the cardinality of CB of these programs. In the following part of this section, unless specifically noted, all the measurements and analysis are based on these Calling Behavior sets. Experiments were done on a 8-core Intel Xeon server with 16 GB of RAM.

3.2 Raw CN: Statistics and Measurement of \( CG_{\infty} \)
Table 2 shows the statistical data of \( CG_{\infty} \) of the 10 programs. The second and third columns in Table 2 give the number of nodes and number of edges of the corresponding \( CG_{\infty} \). Statistics in the 4th to 6th columns correspond to the undirected version of \( CG_{\infty} \).

Real world complex networks often exhibit Power-law degree distribution, expressed as: \( p(k) \sim k^{-\gamma} \), where \( k \) is a node’s degree, \( p(k) \) is the probability distribution of \( k \), and \( \gamma \) is the scaling exponent. We have found that all the programs’ \( CG_{\infty} \) has an approximate Power-law degree distribution. Furthermore, all the scaling exponents (\( \gamma \)) are in the interval (2, 3), which means that \( CG_{\infty} \) is Scale-free [2]. Figure 4 shows the Cumulative Degree Distribution function of Corendal Wiki’s \( CG_{\infty} \).

The 5th column shows the average path length of \( CG_{\infty} \), which is defined as:

\[ l = \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij} \]

where \( d_{ij} \) is the shortest path length between node \( i \) and \( j \). The 6th column gives the average cluster coefficient of \( CG_{\infty} \). A node \( i \)'s cluster coefficient is defined as:

\[ C(i) = \frac{e_i}{k_i(k_i-1)/2} \]

where \( k_i \) is node \( i \)'s degree, \( e_i \) is the number of edges between node \( i \)'s neighbors. The last 2 columns in Table 2 give the average clustering coefficient and the average path length of the corresponding Erdős-Rényi (ER) random graph [8] which has the same quantity of nodes and connection probability with \( CG_{\infty} \). ER random graph is one of the most commonly used random graph models, it is generated by connecting pairs of nodes randomly with a given connection probability \( p \) [8]. Based on Table

![Figure 4: Degree distribution of Corendal Wiki's \( CG_{\infty} \).](image-url)
Table 1: Experiment subject programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Arch</th>
<th>Version</th>
<th>SLoC</th>
<th># Package</th>
<th># Class</th>
<th># Method</th>
<th>CB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corendal Wiki</td>
<td>Web</td>
<td>3.0.2</td>
<td>49,083</td>
<td>21</td>
<td>513</td>
<td>3,147</td>
<td>6,584</td>
</tr>
<tr>
<td>DLOG4J</td>
<td>Web</td>
<td>1.4.2</td>
<td>85,112</td>
<td>38</td>
<td>540</td>
<td>5,384</td>
<td>8,327</td>
</tr>
<tr>
<td>Endeavour</td>
<td>Web</td>
<td>1.21</td>
<td>18,312</td>
<td>6</td>
<td>216</td>
<td>1,706</td>
<td>39,322</td>
</tr>
<tr>
<td>FreeMind</td>
<td>Desktop</td>
<td>0.9.0</td>
<td>53,669</td>
<td>34</td>
<td>960</td>
<td>5,974</td>
<td>192,694</td>
</tr>
<tr>
<td>JForum</td>
<td>Web</td>
<td>2.1.9</td>
<td>65,040</td>
<td>42</td>
<td>397</td>
<td>2,991</td>
<td>42,516</td>
</tr>
<tr>
<td>JPetStore</td>
<td>Web</td>
<td>6.0</td>
<td>1,893</td>
<td>4</td>
<td>24</td>
<td>289</td>
<td>2,099</td>
</tr>
<tr>
<td>Kunagi</td>
<td>Web</td>
<td>0.23</td>
<td>176,486</td>
<td>130</td>
<td>2,792</td>
<td>18,021</td>
<td>198,259</td>
</tr>
<tr>
<td>LogicalDOC</td>
<td>Web</td>
<td>6.7.0</td>
<td>131,888</td>
<td>103</td>
<td>2,058</td>
<td>8,692</td>
<td>160,685</td>
</tr>
<tr>
<td>Makagiga</td>
<td>Desktop</td>
<td>3.8.2</td>
<td>156,906</td>
<td>65</td>
<td>2,020</td>
<td>10,356</td>
<td>324,928</td>
</tr>
<tr>
<td>OpenKM</td>
<td>Web</td>
<td>6.2.2</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>249,990</td>
</tr>
</tbody>
</table>

Table 2: Basic statistics of the 10 programs’ $CG_\infty$s in Raw CNs

<table>
<thead>
<tr>
<th>Programs</th>
<th>$n$</th>
<th>$m$</th>
<th>$\gamma$</th>
<th>$l$</th>
<th>$C$</th>
<th>$C_{ER}$</th>
<th>$l_{ER}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corendal Wiki</td>
<td>475</td>
<td>1,030</td>
<td>2.61</td>
<td>4.27</td>
<td>0.13</td>
<td>0.007</td>
<td>4.30</td>
</tr>
<tr>
<td>DLOG4J</td>
<td>517</td>
<td>748</td>
<td>2.49</td>
<td>4.46</td>
<td>0.02</td>
<td>0.002</td>
<td>5.74</td>
</tr>
<tr>
<td>Endeavour</td>
<td>630</td>
<td>1,799</td>
<td>2.43</td>
<td>3.97</td>
<td>0.07</td>
<td>0.011</td>
<td>3.95</td>
</tr>
<tr>
<td>FreeMind</td>
<td>299</td>
<td>865</td>
<td>2.34</td>
<td>3.02</td>
<td>0.25</td>
<td>0.014</td>
<td>3.54</td>
</tr>
<tr>
<td>JForum</td>
<td>716</td>
<td>1,506</td>
<td>2.63</td>
<td>4.44</td>
<td>0.07</td>
<td>0.008</td>
<td>4.58</td>
</tr>
<tr>
<td>JPetStore</td>
<td>222</td>
<td>328</td>
<td>1.98</td>
<td>3.40</td>
<td>0.04</td>
<td>0.003</td>
<td>5.19</td>
</tr>
<tr>
<td>Kunagi</td>
<td>781</td>
<td>1,352</td>
<td>2.54</td>
<td>5.31</td>
<td>0.10</td>
<td>0.004</td>
<td>5.32</td>
</tr>
<tr>
<td>LogicalDOC</td>
<td>892</td>
<td>3,684</td>
<td>2.51</td>
<td>3.87</td>
<td>0.06</td>
<td>0.010</td>
<td>3.44</td>
</tr>
<tr>
<td>Makagiga</td>
<td>2,358</td>
<td>5,340</td>
<td>2.81</td>
<td>4.75</td>
<td>0.06</td>
<td>0.001</td>
<td>5.27</td>
</tr>
<tr>
<td>OpenKM</td>
<td>1,390</td>
<td>2,929</td>
<td>2.03</td>
<td>4.08</td>
<td>0.09</td>
<td>0.002</td>
<td>5.72</td>
</tr>
</tbody>
</table>

2, it can be concluded that $C \gg C_{ER}$ and $l \approx l_{ER}$, which means that $CG_\infty$ is a typical Small-world network [29].

To sum up, $CG_\infty$ usually has Power-law degree distribution, and is a typical Scale-free and Small-world complex network.

The preceding results have laid a good foundation for further applying Complex Network theories to evaluating software dynamic structure and behavior. To show the application of $CG_\infty$, an experiment was designed and conducted. $CG_\infty$ of JForum was constructed based on JForum’s posting operation. Then 5 faulty versions of JForum were constructed. For the 1st version, we randomly selected 5 methods and injected 50 ms delay into each method. Then based on the $CG_\infty$, each node’s values of Betweenness Centrality [5], Closeness Centrality [11], Communicability Centrality [10] and Load Centrality [22] were computed. The 2nd to 5th fault versions of JForum were constructed by injecting 50 ms delay into 5 methods which had the largest these centrality values respectively. Then we used Apache JMeter\(^{17}\) to simulate concurrent 50 users who perform the same operations with the operations been used to generate $CG_\infty$. For each version, the experiment has been conducted for 10 times. The average bandwidths of these experiments are illustrated in Figure 5. “Normal” in Figure 5 represents the normal version of JForum, while “Random” to “Load” represent the faulty versions respectively. The error bars in Figure 5 depict standard deviations.

This experiment shows that the metrics of nodes’ centrality values in $CG_\infty$ are useful in identifying a software system’s significant module related to the system’s performance. These centrality values are also helpful in a software system’s performance optimization process, as the methods with large centrality values should be optimized preferentially.

\(^{17}\)http://jmeter.apache.org/

![Figure 5: Fault injection experiment on JForum.](image)

The performance degradation introduced by nodes with large centrality values is much significant than the random situation. In Complex Network theory, importance of a node is quantified by its centrality values [2]. Results of this experiment are in accordance with the perceptions of Complex Network theory. This experiment shows the potential applications of $CG_\infty$. Many theoretical results of Complex Network theory and graph algorithms can be further applied to the analysis of $CG_\infty$.

### 3.3 Growing CN: Densification Power Law

In recent years, it has been discovered that the real world networks often exhibit a consistent tendency in their evolutions: these networks usually become denser over time, which is not in accordance with some widely used models, e.g., preferential attachment model [1, 23]. Furthermore, the densification processes exhibit a consistent relation, often expressed as [17, 18]:
where $c(t)$ and $n(t)$ are the number of edges and nodes of the network at time $t$, $a$ is an exponent between 1 and 2. This relation is named as Densification Power Law \cite{17, 18}, which means that network usually become denser and the number of edges and nodes grows obey a super linear relationship defined in equation 2 during the network’s evolution.

During the execution process of a software system, its CN also evolves. Software starts with an entry point, e.g., the main() method in Java program, more methods join in CN as the software executes and does various computation jobs (as shown in Scheme 2 in Figure 2). Growing CN can represent this evolution process. Figure 6 shows the results of Growing CNs of 4 programs.

In Figure 6, the number of edges versus the number of nodes are plotted in log-log scale. The straight line is the linear regression fit results, the slope of the straight line and the correlation coefficient are also shown. These results are quite surprising: it seems that all the experiment programs’ Growing CNs strictly obey the Densification Power Law. We believe that the Densification Power Law nature of Growing CNs is related to essential mechanisms of software systems’ dynamics. Whether the exponent is a quality indicator or a parameter which is related to the functionalities of a software system? This interesting phenomenon needs further research.

### 3.4 Partitioned CN: Stability of Entropy and Entropy-based Applications

#### 3.4.1 Entropy Based Method

In CN, the weights of edges can represent execution frequencies of method calls. Considering the frequencies encode the basic dynamics and method call patterns of a software system, a method in Complex Network theory is used to quantify the method call patterns. For a node $i$ in CN, the Local Entropy \cite{2} of $i$ is:

$$LE(i) = \frac{1}{\ln k_i} \sum_{j \in V(i)} \frac{w_{ij}}{s(i)} \ln \left( \frac{w_{ij}}{s(i)} \right)$$

where $k_i$ is node $i$’s degree, $s(i)$ is node $i$’s vertex strength, defined as: $s(i) = \sum_{j \in V(i)} w_{ij}$, where $V(i)$ is the adjacent node set of node $i$, $w_{ij}$ is the weight between node $i$ and node $j$.

Local Entropy is used to quantify the heterogeneity of node $i$’s edges’ weights. It goes from 0 if all the invocation of $i$ is fully concentrated on one link to the maximal value 1 for $i$’s strength is introduced by homogeneous method calls.

Local Entropy can quantify the heterogeneity of a method’s weight in CN. From software engineering practice perspective, a method is usually called in a relatively fixed pattern, like “in what program state, it should be called by what method for how many times”. This intuition inspired us that the Local Entropy of a method may not change drastically in Partitioned CN, makes it a stable metric to quantify a software method behavior pattern. The following experiment was conducted to verify such hypothesis.

Based on previously introduced data, we used different values of $N_{CG}$ in equation 1 (let $N_{ite} = N_{CG}$) and constructed Partitioned CNs of 7 programs (we haven’t analyzed Corendal Wiki, DLOG4J and JPetStore in this experiment because their CBs are too small). Basic statistics of Partitioned CNs are illustrated in Table 3.

> Table 3: The size of Partitioned CNs of 7 systems

| Programs | \(|CB|\) | \(N_{CG} \) | \(|CN|\) |
|---------|-------|--------|-------|
| Endeavour | 39322 | 2000 | 19 |
| FreeMind | 192694 | 3500 | 56 |
| JForum | 42516 | 1500 | 29 |
| Kunagi | 198259 | 2500 | 80 |
| LogicalDOC | 160685 | 2500 | 65 |
| Makagiga | 324928 | 2000 | 163 |
| OpenKM | 249990 | 4000 | 63 |

Table 3. The values of $N_{CG}$ were chosen with the purpose of generality.

Figure 8 shows values of several metrics in Partitioned CNs of 2 programs. It shows that most of the metrics vary significantly in Partitioned CN. On the other hand, the Local Entropy of each method doesn’t exhibit significant change. It can also be noticed that a certain method may not appear in every CG in Partitioned CN, but the Local Entropy values have little difference among CGs in which the method appears.

Table 4 shows the Sample Variance of methods’ Local Entropy values in these 7 programs’ Partitioned CNs. For each entry, the percentage is shown in the first line, followed by the quantity of the corresponding methods in the second line. It shows that for 69.33% methods of these 7 programs, their Local Entropy doesn’t change in Partitioned CN. For these methods, Local Entropy can be used in runtime failure diagnosis and localization, the following part of this section shows the details. Most of the other methods’ Local Entropy values only exhibit slightly changes. This experiment confirms the hypothesis that Local Entropy of a method usually does not change drastically in Partitioned CN.

#### 3.4.2 Runtime Fault Diagnosis and Localization

In this section, an experiment aiming to show the application of Local Entropy was conducted. In this experiment, we manually injected a runtime fault, which could cause an “Array Index Out of Bounds” exception with 40% probability if the current time of the server exceeds a pre-defined time, into a method of JPetStore. Then we used Apache JMeter to simulate 40 concurrent users: 50% of them were browsing users, and the others had purchasing behaviors.
The result of this experiment is shown in Figure 7. The “faulty method” in Figure 7 represents the method contained the injected fault; the “called method” is one of the methods called by the faulty method, while m1 and m2 are two methods which don’t have direct invocation relations with the faulty method.

As shown in Figure 7, before the injected fault was triggered, Local Entropies of all the 4 methods didn’t have drastic changes (the faulty method and m1’s Local Entropies didn’t change in Partitioned CN). After the 25th slice of Partitioned CN (when the fault was triggered), the Local Entropy of the faulty method exhibited drastic changes, while the Local Entropies of the other 3 methods, including the called method, remained in the same trend as before. By analyzing Local Entropies in Partitioned CN, one can immediately identify the abnormal deviation of the faulty method’s behavior pattern. The partitioned and dynamic nature of Partitioned CN makes it possible to dynamically update and re-evaluate the Local Entropy value, which is the key factor in this experiment.

4. CONCLUSIONS

In this paper, we have proposed a new model – Calling Network, to describe a software system’s runtime method call behaviors. Three CN generation schemes are proposed: Raw CN, Growing CN and Partitioned CN. It has been shown that like other previously proposed networks, Raw CN also has Power-Law degree distribution, and is a typical Small-world and Scale-free network. Experiment shows that centrality values in Raw CN are useful in significant module identification and performance optimization for a software system. It has been discovered that Growing CN evolves strictly obey the newly discovered Densification Power Law, which may reveal the underlying mechanism of a software system’s dynamics. We have proposed using the Local Entropy in Partitioned CN to quantify a method’s behavior pattern. Experiment has been conducted to prove the stability of Local Entropy. We also have conducted experiment to prove that Partitioned CN and the notion of Local Entropy are helpful in software runtime fault diagnosis and localization. In the future, we plan to use more advanced data mining algorithms in the analysis of CN.

5. ACKNOWLEDGMENTS

This paper was supported by the National Natural Science Foundation of China (91118005, 91218301, 61221063, 61203174 and 61202392), Key Projects in the National Science & Technology Pillar Program (2011BAK08B02), Doctoral Fund of Ministry of Education of China (2011020120010) and the Fundamental Research Funds for the Central Universities. We would also like to thank the anonymous reviewers for their valuable comments and suggestions for improving this paper.

6. REFERENCES

Figure 8: Different metrics in Calling Graph sequence of *Partitioned CN*. EN represents number of edges, AD represents average degree, ACC means the average clustering coefficient, and LE represents Local Entropy.


