In-Depth Measurement and Analysis on Densification Power Law of Software Execution

Yu Qu
yqu@sei.xjtu.edu.cn
Qinghua Zheng
qzheng@mail.xjtu.edu.cn
Ting Liu*
tliu@sei.xjtu.edu.cn
Jian Li
jianlee@stu.xjtu.edu.cn
Xiaohong Guan
xhguan@sei.xjtu.edu.cn
MOE Key Lab for Intelligent Networks and Network Security, Xi’an Jiaotong University
Xi’an, China

ABSTRACT
Measuring software execution is important for many software engineering tasks. In this paper, Densification Power Law (DPL) of software execution is measured and studied as a feature of growing software complexity. Densification means that during a networked system’s evolution, it usually becomes denser and the number of edges and nodes grows with a consistent super linear relation. This feature was discovered and reported in 2005. In this paper, based on a measurement of 15 open-source Java programs, we show that when software systems are modeled as a series of dynamic Call Graphs during their executions, they always obey DPL with very close correlation. Then a comparison between static Call Graph and DPL is presented, showing that DPL’s properties cannot be derived statically. An explanation for DPL of software execution is given and verified. We believe the universality of DPL makes it an appropriate metric for software execution process.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—Complexity measures, Process metrics

General Terms
Theory, Measurement

Keywords
Software metrics, software execution, Densification Power Law

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

1. INTRODUCTION

Complex Network theory and graph algorithms have been successfully applied to software measurement and modeling [2, 3, 9]. But only a few of them [3] concentrate on dynamic execution process of software. Although many network growing models [1, 7] have been proposed in Complex Network theory, none of the existing research has investigated whether or not the dynamic processes of software also obey such models. Is there any common law among different software systems’ execution processes? Can we discover new metrics for software execution from a growing network point of view? This paper aims to partially answer these questions and pave new research directions for software metrics.

It has been discovered in recent years that real world networks often exhibit a consistent tendency during their growing processes: they become denser over time. This phenomenon is not in accordance with traditional models, e.g., preferential attachment model [1, 7]. Moreover, the densifying processes exhibit a consistent relation, expressed as: e(t) \propto n(t)^a, where e(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 feature is named as Densification Power Law [5, 6] (DPL).

In our previous research [8], we have discovered DPL feature in software’s Calling Network (CN). CN is a software behavior model which is consisted of a series of Calling Graph (CG), a dynamic version of Call Graph [4].

As shown in Figure 1, during the execution process of a software, its CGs are constructed based on dynamic method/function call records. The sequence of these CGs (CG1, CG2, ..., CGn) presents typical DPL.

This paper makes the following contributions: 1) a much larger scale measurement is performed based on 15 widely-used open-source Java programs to study and show the universality of DPL; 2) A comparison between static Call Graph and the DPL equation is presented, showing that DPL’s properties cannot be derived statically; 3) An explanation for this finding is given. It is proved that the densification of software is mainly caused by the intensive reuse of software methods. We believe these results will provide new angles for software metrics research, and can be used in software dynamic execution and growth measurement and modeling.

2. CALLING NETWORK MODEL

The Calling Network (CN) model is formalized as:
Figure 1: Densification Power Law feature.

\[
CN = \{CG_i \mid i \in \mathbb{N} \}, \\
CG_i = f_{CG \rightarrow Gen}(CB_i), CB_i \subseteq CB \text{ and} \\
CB_i = \{cb_k \mid (i - 1) \cdot N_{Itv} \leq k \leq (i - 1) \cdot N_{Itv} + N_{CG} \}, \\
CG = (V,E), w : E \rightarrow \mathbb{N}, \\
CB = \{cb_k \mid k \in \mathbb{N} \}, \\
\{cb_k = \{t_k, Caller_k, Callee_k, Param_k\}\}.
\]

For more details of this model, please refer to our previous paper [8]. Briefly, setting \(N_{Itv} = 0\) and \(N_{CG} = i \cdot N_{Const}\), where \(N_{CG}\) is the step size to generate \(CG\), \(N_{Const}\) is a constant non-zero value, will generate a \(CN\) model contains a series of \(CGs\) which represent the growing process of dynamic Call Graph over time.

3. MEASUREMENTS & DISCUSSIONS

3.1 Data Set and Experiment Implementation

A data set containing 15 real-world open-source Java programs are collected, as shown in Table 1. Most of the programs are widely used in practice and have millions of downloads on software repository sites. The 2nd column shows the program’s architecture (Arch) (Web or desktop), the 3rd column provides the version number. The 4th to 5th columns give the value of Static Lines of Code, and the number of methods (“N/A”s in Table 1 and Table 2 represent that source codes of the corresponding programs are not publicly available). The 6th column gives the cardinality of \(CB\), which is the quantity of method call records.

The KiCker framework\(^1\), which is an open-source software dynamic monitoring framework based on AspectJ, is used as the instrumentation tool. Specific usage scenarios, user inputs and test cases were designed according to these programs’ functionalities and documents. Then method call behaviors data of these programs are collected.

3.2 Densification Power Law

In our previous research [8], DPL feature was reported for the first time in software domain. In this section, a much larger scale measurement study is performed, followed by the comparison between static Call Graph and the dynamically derived DPL equation.

For the 15 subject programs, their \(CNs\) have been investigated. Figure 2 shows the results of \(CNs\) of 8 programs. In each subfigure, the number of edges versus the number of nodes are plotted in log-log scale. The straight line is the linear regression fit results, and the correlation coefficient of the regression is also shown. The first four subfigures show the results of 4 web programs, followed by the results of 4 desktop programs. It can be noticed that no matter what the architecture, size and functionality is, the DPL feature exists for all the subjects.

Table 2 gives the details of DPL results of the 15 programs. The 2nd column gives the regression equation. The 3rd and 4th columns give the corresponding \(N_{Const}\) (see Section 2) and the regression correlation coefficient \(r\). It can be concluded that although the values of \(N_{Const}\) vary drastically, the real growing process of \(CN\) always fits to DPL with very high correlation coefficient (the smallest value is 0.98775 for LogicalDOC). The 5th column shows the number of nodes of the highest point in DPL figures \((N_{highest})\).

The 6th to 10th columns in Table 2 show the comparisons between the DPL equations and the static Call Graphs which are generated based on source codes of the corresponding programs. The 6th column emphasizes the total number of methods. The 7th and 8th columns show the number of nodes \((N_{static})\) and the number of edges \((E_{static})\) of the static Call Graphs. It should be noticed that for all the programs, \(N_{static}\) is much smaller than the total number of methods. This is mainly caused by the problem which has been discussed in [8]. The \(E_{equation}\) in the 9th column is derived by substituting \(N_{static}\) into the equation in the 2nd column. The last column gives the difference between \(E_{static}\) and \(E_{equation}\):

\[
\Delta E = \frac{E_{equation} - E_{static}}{E_{static}}
\]

The last column shows that the difference between static Call Graph and the equation of DPL is significant (ranging from -3.06% to 801.82%). Based on these results, it can be concluded that DPL’s properties (the exponent \(a\) and the adjustment parameter) can only be derived based on dynamic execution process and cannot be derived statically. The DPL’s properties reflect execution process of software from a certain viewpoint. What does the difference between dynamic and static \((\Delta E)\) imply? This question needs further investigation.

3.3 Microscopic Discussion

What makes the difference between DPL of software system and traditional Complex Network theory models? This section gives a microscopic discussion considering the characteristics of method calls in software.

In traditional Complex Network models [1, 7], when a new node arrives, it will connect to (or be connected by) old nodes following certain mechanisms. This process often leads to a constant average node degree over time. When we investigated the growing process of software’s \(CN\), the following differences have been observed:

Take the growing process of JForum’s \(CN\) for instance, Figure 3 (a) shows the growth details from \(CG_1\) to \(CG_2\) in JForum’s \(CN\). The method \texttt{JForumBaseServlet.init} was in \(CG_1\). When JForum’s functionality executed, more methods were called, adding more nodes into \(CG_2\). In \(CG_2\), the method \texttt{JForumBaseServlet.init} called the method \texttt{getApplicationPath}. Then, to accomplish its functionality, \texttt{getApplicationPath} called the method \texttt{getValue}, which was an old node in \(CG_1\). As the execution of \texttt{JForumBaseServlet.init} continued, it also called the \texttt{getValue} method.

\(^1\)http://kicker-monitoring.net/
### Table 1: Experiment subject programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Arch</th>
<th>Version</th>
<th>SLoC</th>
<th># Method</th>
<th>CB</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>Desktop</td>
<td>5.7.5</td>
<td>162,416</td>
<td>10,593</td>
<td>447,842</td>
<td><a href="http://www.drjava.org/">http://www.drjava.org/</a></td>
</tr>
<tr>
<td>Endeavour</td>
<td>Web</td>
<td>1.21</td>
<td>18,312</td>
<td>1,706</td>
<td>39,322</td>
<td><a href="http://sourceforge.net/projects/endeavour-mgmt/">http://sourceforge.net/projects/endeavour-mgmt/</a></td>
</tr>
<tr>
<td>FreeMind</td>
<td>Desktop</td>
<td>0.9.0</td>
<td>53,609</td>
<td>5,974</td>
<td>192,694</td>
<td><a href="http://sourceforge.net/projects/freemind/">http://sourceforge.net/projects/freemind/</a></td>
</tr>
<tr>
<td>JabRef</td>
<td>Desktop</td>
<td>2.9.2</td>
<td>144,406</td>
<td>6,449</td>
<td>185,144</td>
<td><a href="http://jabref.sourceforge.net/">http://jabref.sourceforge.net/</a></td>
</tr>
<tr>
<td>jEdit</td>
<td>Desktop</td>
<td>5.1.0</td>
<td>185,569</td>
<td>7,844</td>
<td>438,321</td>
<td><a href="http://www.jedit.org/">http://www.jedit.org/</a></td>
</tr>
<tr>
<td>JForum</td>
<td>Web</td>
<td>2.1.9</td>
<td>65,040</td>
<td>2,991</td>
<td>160,685</td>
<td><a href="http://jforum.net/">http://jforum.net/</a></td>
</tr>
<tr>
<td>JPetStore</td>
<td>Web</td>
<td>6.0</td>
<td>1,893</td>
<td>289</td>
<td>3,331</td>
<td><a href="http://code.google.com/p/mybatis/">http://code.google.com/p/mybatis/</a></td>
</tr>
<tr>
<td>Kunagi</td>
<td>Web</td>
<td>0.23</td>
<td>176,486</td>
<td>18,021</td>
<td>198,259</td>
<td><a href="http://kunagi.org/">http://kunagi.org/</a></td>
</tr>
<tr>
<td>LogicalDOC</td>
<td>Desktop</td>
<td>3.8.2</td>
<td>156,906</td>
<td>10,564</td>
<td>192,694</td>
<td><a href="http://www.logicaldoc.com/">http://www.logicaldoc.com/</a></td>
</tr>
<tr>
<td>OpenKM</td>
<td>Web</td>
<td>6.2.2</td>
<td>N/A</td>
<td>N/A</td>
<td>249,990</td>
<td><a href="http://www.openkm.com/">http://www.openkm.com/</a></td>
</tr>
<tr>
<td>OpenProj</td>
<td>Desktop</td>
<td>1.4</td>
<td>151,821</td>
<td>11,632</td>
<td>371,750</td>
<td><a href="http://sourceforge.net/projects/openproj/">http://sourceforge.net/projects/openproj/</a></td>
</tr>
<tr>
<td>OpenSyncro</td>
<td>Web</td>
<td>2.2</td>
<td>54,163</td>
<td>3,276</td>
<td>137,433</td>
<td><a href="http://www.opensyncro.org/">http://www.opensyncro.org/</a></td>
</tr>
<tr>
<td>Sweet Home 3D</td>
<td>Desktop</td>
<td>4.2</td>
<td>109,090</td>
<td>6,346</td>
<td>381,586</td>
<td><a href="http://www.sweethome3d.com/">http://www.sweethome3d.com/</a></td>
</tr>
<tr>
<td>Weka</td>
<td>Desktop</td>
<td>3.7.10</td>
<td>N/A</td>
<td>N/A</td>
<td>237,239</td>
<td><a href="http://www.cs.waikato.ac.nz/ml/weka/">http://www.cs.waikato.ac.nz/ml/weka/</a></td>
</tr>
</tbody>
</table>

### Table 2: Densification Power Law results and comparison with Static Call Graphs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Equation</th>
<th>$N_{\text{Const}}$</th>
<th>$r$</th>
<th>$N_{\text{highest}}$</th>
<th>$#\text{Methods}$</th>
<th>$N_{\text{stat}}$</th>
<th>$E_{\text{stat}}$</th>
<th>$E_{\text{equation}}$</th>
<th>$\Delta E%$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DrJava</td>
<td>0.08431x + 0.746</td>
<td>7,000</td>
<td>0.99956</td>
<td>1.701</td>
<td>10,593</td>
<td>6,204</td>
<td>13,546</td>
<td>12,938</td>
<td>-3.06%</td>
</tr>
<tr>
<td>Endeavour</td>
<td>0.10727x + 0.858</td>
<td>1,000</td>
<td>0.99544</td>
<td>723</td>
<td>1,706</td>
<td>1,468</td>
<td>5,749</td>
<td>5,274</td>
<td>9.91%</td>
</tr>
<tr>
<td>FreeMind</td>
<td>0.31904x + 1.055</td>
<td>2,500</td>
<td>0.99816</td>
<td>236</td>
<td>6,449</td>
<td>4,885</td>
<td>9,805</td>
<td>11,565</td>
<td>17.95%</td>
</tr>
<tr>
<td>JabRef</td>
<td>0.67521x + 1.187</td>
<td>4,000</td>
<td>0.99894</td>
<td>867</td>
<td>2,991</td>
<td>2,051</td>
<td>5,749</td>
<td>5,274</td>
<td>-3.03%</td>
</tr>
<tr>
<td>jEdit</td>
<td>0.26761x + 1.249</td>
<td>5,000</td>
<td>0.99702</td>
<td>2,221</td>
<td>7,844</td>
<td>5,606</td>
<td>13,845</td>
<td>16,094</td>
<td>16.24%</td>
</tr>
<tr>
<td>JForum</td>
<td>0.31046x + 2.646</td>
<td>1,000</td>
<td>0.99341</td>
<td>715</td>
<td>2,991</td>
<td>2,051</td>
<td>5,749</td>
<td>5,274</td>
<td>-3.03%</td>
</tr>
<tr>
<td>JPetStore</td>
<td>0.97557x + 1.178</td>
<td>50</td>
<td>0.99826</td>
<td>221</td>
<td>289</td>
<td>97</td>
<td>124</td>
<td>122</td>
<td>-1.2%</td>
</tr>
<tr>
<td>Kunagi</td>
<td>0.73925x + 1.287</td>
<td>4,500</td>
<td>0.99858</td>
<td>780</td>
<td>18,021</td>
<td>12,583</td>
<td>24,699</td>
<td>31,349</td>
<td>26.92%</td>
</tr>
<tr>
<td>LogicalDOC</td>
<td>0.0077485x + 0.921</td>
<td>2,000</td>
<td>0.98775</td>
<td>891</td>
<td>8,692</td>
<td>5,932</td>
<td>11,122</td>
<td>12,270</td>
<td>801.82%</td>
</tr>
<tr>
<td>Makagiga</td>
<td>0.47128x + 2.075</td>
<td>1,500</td>
<td>0.99887</td>
<td>1,776</td>
<td>10,356</td>
<td>7,078</td>
<td>17,811</td>
<td>20,992</td>
<td>18.06%</td>
</tr>
<tr>
<td>OpenKM</td>
<td>0.46685x + 1.842</td>
<td>6,000</td>
<td>0.99963</td>
<td>1,389</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>OpenProj</td>
<td>0.57761x + 1.437</td>
<td>2,000</td>
<td>0.99918</td>
<td>2,823</td>
<td>11,632</td>
<td>7,258</td>
<td>13,752</td>
<td>13,494</td>
<td>-1.87%</td>
</tr>
<tr>
<td>OpenSyncro</td>
<td>0.29226x + 2.085</td>
<td>3,500</td>
<td>0.99502</td>
<td>657</td>
<td>3,276</td>
<td>1,865</td>
<td>2,937</td>
<td>4,823</td>
<td>64.21%</td>
</tr>
<tr>
<td>Sweet Home 3D</td>
<td>0.3917x + 1.948</td>
<td>5,500</td>
<td>0.99956</td>
<td>1,117</td>
<td>6,346</td>
<td>4,547</td>
<td>11,219</td>
<td>11,744</td>
<td>4.68%</td>
</tr>
<tr>
<td>Weka</td>
<td>0.57267x + 1.177</td>
<td>4,000</td>
<td>0.99942</td>
<td>910</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 2: Densification Power Law phenomenon of 8 open-source programs.
The aforementioned process added 3 new edges to $CG_2$. Among them, one edge was between 2 old nodes in $CG_1$ (the purple edge), one edge (the green edge) was a “passive” method call to accomplish the functionality of a new node ($getApplicationPath$).

These 2 edges are different from the edge which links $JForumBaseServlet.init$ and $getApplicationPath$, the latter is the same with all the red edges in Figure 3 (a). These 2 situations (new edge between 2 old nodes and “passive” edge) have not been considered in existing models. What if these edges are excluded in $CN$? Whether we could derive a similar result with traditional models?

Figure 3 (b) shows the results of $JForum$, all the points are the number of nodes versus the number of edges after removing the 2 aforementioned kinds of edges with different $N_{Const}$. These results are consistent with the predictions made by existing Complex Network models [1, 7] (constant average node degree over time). Similar results have been obtained on other subject programs.

Thus, the densification process is mainly caused by the pervasive dynamic reuse of software methods. But why the densification process follows a consistent exponent? This question should be answered in future research.

4. CONCLUSIONS

Based on 15 widely-used software systems, we have shown the universality of Densification Power Law (DPL) during software’s execution. We believe this interesting finding will provide new angles for software runtime measurement. The difference between static Call Graph and DPL has been presented. An explanation for the densification process has been given. It is believed that the major cause of this finding is the reuse of software methods. After removing these reusing method calls, the growth of software’s $CN$ is in accordance with predictions made by traditional Complex Network models. These measurements and findings will pave new research directions for software metrics.

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 (2011201120010) and the Fundamental Research Funds for the Central Universities.

6. REFERENCES