Software Systems through Complex Networks Science: Review, Analysis and Applications

Lovro Šubelj
University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
lovro.subelj@fri.uni-lj.si

Marko Bajec
University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, SI-1000 Ljubljana, Slovenia
marko.bajec@fri.uni-lj.si

ABSTRACT

Complex software systems are among most sophisticated human-made systems, yet only little is known about the actual structure of ‘good’ software. We here study different software systems developed in Java from the perspective of network science. The study reveals that network theory can provide a prominent set of techniques for the exploratory analysis of large complex software system. We further identify several applications in software engineering, and propose different network-based quality indicators that address software design, efficiency, reusability, vulnerability, controllability and other. We also highlight various interesting findings, e.g., software systems are highly vulnerable to processes like bug propagation, however, they are not easily controllable.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures, software science

General Terms
Theory, algorithms, experimentation.

Keywords
Software systems, Software engineering, Software networks, Network analysis.

1. INTRODUCTION

Complex software systems are among most sophisticated systems ever created by human. Nevertheless, only little is known about the actual structure and quantitative properties of large software systems. For instance, in the context of software engineering, one is interested in how ‘good’ software looks like. Commonly adopted approaches and techniques fail to give a comprehensive answer, moreover, there is also a lack of a simple but yet rigorous framework for software analysis (to our knowledge). The above dilemma was denoted software law problem, which urges towards identifying (physical) laws obeyed by software systems that could be used in practical applications.

Networks possibly provide the most adequate framework for the analysis of the structure of complex systems like software projects. Also, due to their simple and intelligible form, analysis of different networks has already provided several significant discoveries in the last decade (46, 5, 16, 23). Note that the adoption of software networks is not novel (35, 27, 19, 29), however, network analysis is still only rarely used in software engineering. The main purpose of this study is thus to highlight different techniques developed in the field of network analysis, and to expose their use in software comprehension, development and engineering. We review most of the past work on different types of software networks, whereas we also include network analysis techniques proposed just recently (23, 14). (Note that the main focus of the paper is merely a review, rather than a detailed comparison of network analysis techniques with other approaches.)

The study in the paper analyses software networks on different levels of granularity. First, we address the macroscopic properties of software networks like scale-free and small-world phenomena (46, 5) that are related to the structure and design of the entire project, or projects, represented by the network. Second, we analyze the microscopic properties of individual nodes, with special emphasis on different dynamical processes occurring on software networks like bug propagation (2, 30). The above can be related to software quality, complexity, reusability, robustness, vulnerability and controllability. Third, we also identify mesoscopic structural modules within software networks (16, 44) and show their applicability in the context of software abstraction and refactoring. The paper thus exposes network analysis as a prominent set of techniques for software engineering.

The rest of the paper is structured as follows. Section 2 introduces software networks used in the study. Section 3 analyzes different characteristics of adopted networks and discusses their use in software engineering. Some applications of the presented techniques are given in Section 4 while Section 5 concludes the paper.

2. SOFTWARE NETWORKS

Various types of networks have been proposed for the analysis of the structure of complex software systems. For
Figure 1: (left) A simple Java class and the corresponding part of class dependency network. Direction of links is (mostly) just the opposite to the flow of information. (right) Class dependency network of java (circles) and java (triangles) namespaces of Java language.

### Table 1: Properties of class dependency networks used in the study.

| Network | Project                        | $n$  | $m$  | $k$ | LCC | $|A|$ | $|P|$ |
|---------|--------------------------------|------|------|-----|-----|------|------|
| flmng   | Flamingo 4.1 (GUI components) | 141  | 269  | 3.82| 0.88| 153  | 18  |
| colt    | Colt 1.2.0 (scientific computation) | 243  | 720  | 5.93| 0.94| 267  | 21  |
| jung    | JUNG 2.0.1 (network analysis)  | 317  | 719  | 4.54| 0.96| 357  | 41  |
| org     | Java 1.6.0.7 (org namespace)  | 709  | 3571 | 10.07| 0.69| 778  | 50  |
| weka    | Weka 3.6.6 (data mining framework) | 953  | 4097 | 8.60| 0.98| 1054 | 84  |
| java    | Java 1.6.0.7 (java namespace) | 1595 | 5287 | 6.63| 0.44| 1889 | 118 |

instance, software architecture maps [39], software mirror graphs [6], class, method and package collaboration graphs [17], subroutine call graphs [27], inter-package dependency networks [21], software class diagrams [37] and class dependency networks [39], to name just a few. Networks mainly divide whether they are constructed from source code, byte code or program execution traces, and due to the level of software architecture represented by the nodes, and the set of interdependencies represented by the links.

For consistency with some previous work [4, 17, 47, 39] (Table 1). Due to the object-oriented view of Java language, nodes in the network can represent either project packages, software classes, methods and functions or individual lines of code. We here adopt class dependency networks [39], where nodes represent classes and links correspond to different dependencies among them (Figure 1).

Formally, let a project consist of classes $A = \{A_1, A_2, \ldots \}$ and let $P$ be the set of software packages (bottom-most level of the package hierarchy). Corresponding class dependency network is then a directed graph $G(N, L)$, with nodes $N = \{1, \ldots, n\}$ and links $L (m = |L|)$. Node $i$ corresponds to a class $A_i$, however, since isolated nodes are discarded in the analysis, $n \leq |A|$. A directed link $(i, j) \in L$ represent some dependency between classes $A_i$ and $A_j$; inheritance ($A_i$ inherits or implements $A_j$), parameter ($A_i$ contains a method, function or constructor that takes $A_j$ as parameter), return ($A_i$ contains a method or function that returns $A_j$) and field ($A_i$ contains a field of type $A_j$). Denote $k$ to be the average degree in the network (i.e., average number of links incident to a node). Furthermore, let $k^{in}$ and $k^{out}$ be the average in-degree and out-degree of the nodes, $k = k^{in} + k^{out}$. Hence, $k^{out}$ corresponds to a number of other classes required to implement the functionality of a respective class $A_i$, while $k^{in}$ corresponds to the number of classes that use (depend on) $A_i$. Last, denote LCC to be the fraction of nodes in the largest connected component $3$.

Table 1 shows properties of class dependency networks used in the study. Networks were selected thus to represent a diverse set of software systems including utility libraries

3Networks are available from [http://lovro.lpt.fri.uni-lj.si/](http://lovro.lpt.fri.uni-lj.si/)

3All networks in figures are reduced to LCC-s.
(e.g., fltnq and colt networks), complete frameworks (e.g., jung and weka networks) and also the core of Java language itself (i.e., java network).

Software networks are compared against Erdős-Rényi random graphs [12], where a link are placed between each pair of n nodes with probability k/(n-1), where k = 2m/n for some n and m.

3. ANALYSIS AND DISCUSSION

3.1 Scale-free networks – software complexity and reusability

Simple random graphs experience a Poisson degree distribution \( p_k \). On the contrary, \( p_k \) of most real-world networks including software follows a power-law form \( p_k \sim k^{-\gamma} \), where \( \gamma \) is a scale-free exponent, \( \gamma > 1 \). The latter can be clearly observed by a straight line with slope \(-\gamma\) in a log-log plot (Figure 2). Networks with power-law degree distributions are denoted scale-free, while \( \gamma \) can be directly related to the spreading processes occurring on networks [32] (e.g., bug propagation). For \( \gamma \in (2, 3) \), even a very small fraction of faulty nodes can already render the entire system inapplicable [30]. Unfortunately, the latter applies for all software networks used here (Table 2).

Scale-free networks are usually considered an artifact of Yule’s process or rich get richer phenomena [3]. For class dependency networks, this refers to the fact that highly used classes are, obviously, well known among developers, and would thus also be more commonly adopted in the future. However, power-laws should thus emerge merely in the indegree distribution \( p_i^{\text{in}} \) that refers to the number of times each class is used [35] (Figure 3). More precisely, scale-free nature of \( p_i^{\text{in}} \) is a result of high code reusability. On the other hand, out-degree distribution \( p_k^{\text{out}} \) is related to software complexity, since classes with high \( k^{\text{out}} \) encompass most complex functionality. Here, complexity refers to the number of other classes needed to implement the functionality of the respective class. For example, most commonly reused class in java network is String, whereas FileDialog is the most complex one (Table 3).

Well developed software project should thus exhibit scale-free \( p_i^{\text{in}} \) and highly truncated \( p_k^{\text{out}} \). Next, lower \( \gamma \) indicates higher code reuse, which also decreases the probability of fault propagation throughout the system. Last, classes with very high \( k^{\text{out}} \), and also \( k^{\text{in}} \), should be implemented with extra care (see Section 3.3).

3.2 Small-world networks – software structure and design

Software networks exhibit small-world phenomena [46] (see Figures 2 and Table 3), which refers to high clustering \( C \) and very short average distance between the nodes \( l \) (also known as six degrees of separation [29]). \( C \) measures transitivity in the network, and is defined as the probability that two neighbors of a node are also linked, \( C \in [0, 1] \), \( l = \frac{1}{n(n-1)} \sum_{i \neq j} d_{ij} \), where \( d_{ij} \) is the distance between \( i \) and \( j \) in the respective undirected network (i.e., number of links in the shortest path). Small-world networks most commonly refer to \( C \gg C_{\text{ER}} \) and \( l \approx l_{\text{ER}} \) [46], where \( C_{\text{ER}} \) and \( l_{\text{ER}} \) are the values for a corresponding random graph.

Clustering of software networks can be related to intrinsic characteristics of the underlying systems [43]. For instance, visualization classes usually experience very high clustering, while clustering is almost zero for I/O classes [43].

Average distance \( l \) is an important indicator of the structural design of the project, or projects, represented by the network. More precisely, since \( l \approx l_{\text{ER}}, l \gg l_{\text{ER}} \) indicates that the underlying software system has divided into several independent parts with rather different functionality (Fig-

![Figure 2: Degree distributions of weka, javax and java networks.](image-url)
For instance, high-level Java class `String` implies a cyclic flow of information within the software project.

It ought to be mentioned that software networks are small-world only in the undirected case [19]. The contrary would completely obscure its structure and dynamics.

![Figure 3: A random graph, `jung` network, `jung & colt` network and `jung & java` network. Average distance between the nodes $l$ equals 3.88, 4.19, 5.37 and 2.18. Node symbols correspond to clustering $D$ [33] that ranges between 0 (triangles) and 1 (circles).]

![Figure 4: `weka`, `javax` and `java` networks with highlighted seed nodes.]

Well designed software project should thus experience $C \gg C_{ER}$, $l \approx l_{ER}$ and $E \approx 0$. Also, one should be wary of $l \gg l_{ER}$ throughout the project development.

### 3.3 Network nodes – software vulnerability and control

In the context of spreading processes on software networks [28] [12] (e.g., bug propagation) and network robustness [2] [24] (i.e., software vulnerability), one is interested into so called seed nodes that could originate the propagation of faults through the entire systems [4]. Centrality metrics that measure nodes influence are commonly regarded as a prominent indicator of seed nodes [13] [14]. Denote $DC_i$ to be the degree centrality defined as $DC_i = k_i/(n - 1)$, where $k_i$ is the degree of node $i$, $DC_i \in [0, 1]$. Next, denote $CC_i$ to be the harmonic closeness centrality defined as the average inverse of distance from $i$ to the rest of the nodes, $CC_i = 1/n \sum_{j \neq i} 1/d_{ij}$, $CC_i \in [0, 1]$. Last, denote $BC_i$ to be the betweenness centrality defined as the fraction of shortest paths between the nodes that go through $i$, $BC_i \in [0, 1]$.

As $k_i \approx k_i^{in}$ for software networks, $DC_i$ actually identifies classes with the highest code reuse or, equivalently, high in-degree $k_i^{in}$ (Table 3). Similar set of influential classes is reported by $BC_i$ (Table 3). On the other hand, $CC_i$ identifies classes that somewhat coincide with high complexity classes identified in Section 3.1. $BC_i$ (and $DC_i$) thus reveals classes whose faulty implementation could influence the entire system, whereas $CC_i$ exposes classes that are most prone to an arbitrary fault within the system. The former commonly reside in the core of the respective software network, whereas $CC_i$ exposures classes that are most prone to an arbitrary fault within the system. The latter are found in the periphery (Figure 4).

Extra care should be put in the development of classes with high $BC_i$, while high $CC_i$ classes can be adopted for an effective, and also efficient, software testing.

Network controllability has just recently been proposed for the analysis of directed real-world networks [24] [23]. Here, one is particularly interested in the number of driver nodes $n_d$ that one has to govern in order guide the entire system [23] (i.e., gain control over the output of the system under the assumption of simple linear transformations). For scale-free networks with $p_i^{out}$ equal to $p_i^{out} = n_d/n \approx e^{k(\gamma - 2)/(2-2\gamma)}$, $\gamma > 2$ [23]. Note that, contrary to seed nodes (Table 1) and general belief, driver nodes tend to avoid high degree nodes [23] [11].

---

4 Although a poor implementation of any software class already makes the system vulnerable, the problem is even amplified in the case of, e.g., highly reused classes.
Table 3: Hubs (i.e., nodes with very high degree) within weka, javax and java networks.

<table>
<thead>
<tr>
<th></th>
<th>weka</th>
<th></th>
<th>java</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>(k_i)</td>
<td>(k_{out})</td>
<td>Node</td>
<td>(k_i)</td>
</tr>
<tr>
<td>Instances</td>
<td>541</td>
<td>5</td>
<td>JComponent</td>
<td>235</td>
</tr>
<tr>
<td>Instance</td>
<td>391</td>
<td>4</td>
<td>Accessible</td>
<td>222</td>
</tr>
<tr>
<td>Capabilities</td>
<td>304</td>
<td>4</td>
<td>ComponentUI</td>
<td>175</td>
</tr>
<tr>
<td>ClassAssigner</td>
<td>0</td>
<td>19</td>
<td>JTable</td>
<td>6</td>
</tr>
<tr>
<td>Filter</td>
<td>0</td>
<td>19</td>
<td>JTextPane</td>
<td>0</td>
</tr>
<tr>
<td>Classifier</td>
<td>0</td>
<td>18</td>
<td>JMenu</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Seed nodes (i.e., very influential nodes) within weka, javax and java networks.

<table>
<thead>
<tr>
<th></th>
<th>weka</th>
<th></th>
<th>java</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>CC_i</td>
<td>BC_i</td>
<td>Node</td>
<td>CC_i</td>
</tr>
<tr>
<td>PredictionAppender</td>
<td>0.03</td>
<td>0.00</td>
<td>DefaultCellEditor</td>
<td>0.10</td>
</tr>
<tr>
<td>Classifier</td>
<td>0.03</td>
<td>0.01</td>
<td>JTable</td>
<td>0.10</td>
</tr>
<tr>
<td>Filter</td>
<td>0.03</td>
<td>0.00</td>
<td>JTextPane</td>
<td>0.09</td>
</tr>
<tr>
<td>Instances</td>
<td>0.01</td>
<td>0.51</td>
<td>JComponent</td>
<td>0.04</td>
</tr>
<tr>
<td>RevisionHandler</td>
<td>0.00</td>
<td>0.26</td>
<td>Accessible</td>
<td>0.01</td>
</tr>
<tr>
<td>Instance</td>
<td>0.01</td>
<td>0.13</td>
<td>PrintService</td>
<td>0.02</td>
</tr>
</tbody>
</table>

Most software network are not highly controllable, since one would have to manage 30-50% of classes in order to control the entire project (Table 2). Nevertheless, due to high density, the core of Java language can be controlled through merely 17% of classes in javax namespace. For comparison, \(n_d/n\) equals \(\approx 80\%\) for regulatory networks, \(\approx 50\%\) for the Internet, \(\approx 30\%\) for power grids and on-line social networks, while, interestingly, it is below 3% for corporate ownership networks.

Controllability of a software system can be limited by decreasing \(k\) or \(\gamma\), which is achieved by decreasing code complexity and increasing code reuse (Section 3.4).

3.4 Network modules – software aggregation and modularity

Packages of the software system reflect in different structural modules within class dependency networks. For instance, visualization classes commonly aggregate into communities of densely connected nodes, whereas different parsers, transformers or plugins often arrange into functional modules that correspond to (disconnected) groups of nodes with common linkage patterns. Otherwise, clear community structure signifies highly modular structure of the respective software system, while well supported functional modules are related to clear functional roles of the classes within the project.

Table 5 compares software packages against network modules with high degree centrality. The nodes are weighted according to Jaccard similarity, which is defined as \(|\Gamma_i \cap \Gamma_j| / |\Gamma_i \cup \Gamma_j|\), where \(j\) is a similar node and \(\Gamma_i\) is the neighborhood of node \(i\). Structural modules are identified with the algorithm in [44].

Table 6 shows classification accuracies for the prediction of dependencies between the classes of a project. One can adopt a community detection algorithm to reveal highly modular structure (Figure 6 (left)) or a functional module detection algorithm to identify the underlying functional structure (Figure 6 (middle)). General structural module detection algorithms partition software classes according to both modular and functional links that are present among the dependencies of the project (Figure 6 (right)).

4. APPLICATIONS

Due to space limitations, the following section only briefly describes different applications of network analysis techniques presented in Section 3. Future work will focus on a more detailed examination and development of supporting implementations that could be easily applied in practice.

4.1 Software project abstraction

Figure 5 shows an application of network structural module detection to software project abstraction. One can identify an entire hierarchy of modules that is consistent with the package hierarchy, while also enclosing class dependencies that go beyond packages decided by the developers. Besides better comprehension, revealed hierarchy enables the prediction of dependencies between the classes of a project.

4.2 Software packages refactoring

Network module detection algorithms can also be applied for refactoring of software packages. One can adopt a community detection algorithm to reveal highly modular structure (Figure 6 (left)) or a functional module detection algorithm to identify the underlying functional structure (Figure 6 (middle)). General structural module detection algorithms partition software classes according to both modular and functional links that are present among the dependencies of the project (Figure 6 (right)).

4.3 Software packages prediction

Table 5 shows classification accuracies for the prediction of software packages for the classes of different systems. Let \(i\) be a node corresponding to class \(A_i\). Package of \(A_i\) is then predicted to be the most likely package considering nodes within the same structural module as \(i\). The nodes are weighted according to Jaccard similarity, which is defined as \(|\Gamma_i \cap \Gamma_j| / |\Gamma_i \cup \Gamma_j|\), where \(j\) is a similar node and \(\Gamma_i\) is the neighborhood of node \(i\). Structural modules are identified with the algorithm in [44].

On average, one can predict software packages with \(\approx 80\%\) probability for most classes of the systems considered, whereas complete package hierarchy can be precisely identified for over 60% of the software classes (Table 6).

4.4 Software quality indicators

Table 7 and Table 8 show software project and class qual-
Figure 5: (left) *jung* network where node symbols represent high-level packages of JUNG framework: visualization (circles), io (triangles), graph (squares) and algorithms (diamonds). (right) Hierarchy of structural modules revealed with the algorithm in [43].

Figure 6: (left) Communities representing highly modular structure of the software system [42, 41]. (middle) Functional modules that represent highly functional partitioning of the system [44, 43]. (right) General structural modules conveying modular and functional links (bottom-most level of the hierarchy in Figure 5).

Table 5: Normalized mutual information [10] (NMI) between software packages and identified network modules, NMI ∈ [0, 1]. Number of modules is shown in small font.

<table>
<thead>
<tr>
<th>Network</th>
<th>MO</th>
<th>CP</th>
<th>MM</th>
<th>GP</th>
</tr>
</thead>
<tbody>
<tr>
<td>flmng</td>
<td>16</td>
<td>0.580</td>
<td>14</td>
<td>0.609</td>
</tr>
<tr>
<td>colt</td>
<td>19</td>
<td>0.519</td>
<td>10</td>
<td>0.473</td>
</tr>
<tr>
<td>jung</td>
<td>39</td>
<td>0.614</td>
<td>13</td>
<td>0.650</td>
</tr>
<tr>
<td>org</td>
<td>47</td>
<td>0.503</td>
<td>11</td>
<td>0.537</td>
</tr>
<tr>
<td>weka</td>
<td>81</td>
<td>0.558</td>
<td>26</td>
<td>0.410</td>
</tr>
<tr>
<td>javax</td>
<td>107</td>
<td>0.704</td>
<td>59</td>
<td>0.761</td>
</tr>
</tbody>
</table>

Table 6: Classification accuracy (CA) for software package prediction, CA ∈ [0, 1]. ($l_\infty$ is the number of levels of the package hierarchy, whereas $l$ is the average level for a software class. Value under $P_i$ corresponds to CA for the $i$-th level of the hierarchy.)

<table>
<thead>
<tr>
<th>Network</th>
<th>$l$</th>
<th>$l_\infty$</th>
<th>P</th>
<th>$P_1$</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_4$</th>
<th>$P_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>flmng</td>
<td>2.65</td>
<td>4</td>
<td>0.566</td>
<td>0.572</td>
<td>0.793</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>colt</td>
<td>3.35</td>
<td>4</td>
<td>0.654</td>
<td>0.756</td>
<td>0.942</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jung</td>
<td>2.97</td>
<td>4</td>
<td>0.617</td>
<td>0.663</td>
<td>0.857</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>org</td>
<td>3.50</td>
<td>7</td>
<td>0.616</td>
<td>0.616</td>
<td>0.714</td>
<td>0.989</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>weka</td>
<td>3.02</td>
<td>6</td>
<td>0.684</td>
<td>0.692</td>
<td>0.736</td>
<td>0.871</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>javax</td>
<td>3.11</td>
<td>5</td>
<td>0.626</td>
<td>0.631</td>
<td>0.816</td>
<td>0.982</td>
<td>1.000</td>
<td></td>
</tr>
</tbody>
</table>
ity indicators identified in the study. Indicators can be employed to assess project structure and design, code complexity, and vulnerability and controllability properties. Due to space limitations, comparison with other approaches for measuring software quality is omitted (e.g., metrics of coupling and cohesion [34]).

5. CONCLUSIONS

The paper conducts a comprehensive study of software networks constructed from Java source code. First, we address macroscopic network properties that are related to structural design of the corresponding software project. Next, we analyze the networks on a microscopic level of nodes, to highlight most influential and vulnerable software classes. Last, we analyze mesoscopic network structural modules and expose their applicability in project refactoring. Among other, we show that software systems are highly vulnerable to processes like bug propagation, however, they are not easily controllable. On the other hand, Java language can be controlled through merely 17% of java namespace. We also identify several network-based quality indicators that can be employed to assess software project design, reusability, robustness, controllability and other. The study thus exposes network analysis as a prominent set of tools for software systems engineering.

6. ACKNOWLEDGMENTS

This work has been supported by the Slovene Research Agency ARRS within Research Program No. P2-0359.

7. REFERENCES


