Validation of Network Analysis Methods Applied in the Context of Dynamic Analysis of Software Systems

Anjan Pakhira and Peter Andras
Validation of Network Analysis Methods Applied in the Context of Dynamic Analysis of Software Systems

A. Pakhira and P. Andras

Abstract

Evolution of large-scale software systems generates very complex systems. The combination of network analysis with dynamic analysis provides a promising approach to understand such systems and support their maintenance and evolution. However, an important issue is the validity of network analysis based predictions about the functional importance of system components. Here we analyse dynamic analysis data generated for the JHotDraw 6.01b software system using network analysis methods. We show that network analysis based metrics can identify functionally important components (methods of classes) of the software system. However, we also show that some network metrics perform better than others. We show that combinations of network metrics may lead to improved performance in predicting functionally important software components, but this is again not always the case. Our results confirm the usefulness of network analysis methods in the context of dynamic analysis of software, and also underline the importance of proper validation of these methods.
Evolution of large-scale software systems generates very complex systems. The combination of network analysis with dynamic analysis provides a promising approach to understand such systems and support their maintenance and evolution. However, an important issue is the validity of network analysis based predictions about the functional importance of system components. Here we analyse dynamic analysis data generated for the JHotDraw 6.01b software system using network analysis methods. We show that network analysis based metrics can identify functionally important components (methods of classes) of the software system. However, we also show that some network metrics perform better than others. We show that combinations of network metrics may lead to improved performance in predicting functionally important software components, but this is again not always the case. Our results confirm the usefulness of network analysis methods in the context of dynamic analysis of software, and also underline the importance of proper validation of these methods.

About the authors

Mr Anjan Pakhira is PhD student in the School of Computing Science of Newcastle University.

Dr Peter Andras is Reader in the School of Computing Science of Newcastle University.

Suggested keywords

DYNAMIC ANALYSIS
NETWORK ANALYSIS
COMPLEX NETWORK
EXPERIMENTAL VALIDATION
SOFTWARE MAINTENANCE
Validation of Network Analysis Methods Applied in the Context of Dynamic Analysis of Software Systems

Anjan Pakhira
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
+44-191-2228593
ANJAN.PAKHIRA@NCL.AC.UK

Peter Andras
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
+44-191-2227946
PETER.ANDRAS@NCL.AC.UK

ABSTRACT
Evolution of large-scale software systems generates very complex systems. The combination of network analysis with dynamic analysis provides a promising approach to understand such systems and support their maintenance and evolution. However, an important issue is the validity of network analysis based predictions about the functional importance of system components. Here we analyse dynamic analysis data generated for the JHotDraw 6.01b software system using network analysis methods. We show that network analysis based metrics can identify functionally important components (methods of classes) of the software system. However, we also show that some network metrics perform better than others. We show that combinations of network metrics may lead to improved performance in predicting functionally important software components, but this is again not always the case. Our results confirm the usefulness of network analysis methods in the context of dynamic analysis of software, and also underline the importance of proper validation of these methods.

Categories and Subject Descriptors
D.2.8. [Metrics]: Software science; D.2.7. [Distribution, Maintenance and Enhancement]: Restructuring, reverse engineering, and reengineering.

General Terms
Algorithms, Measurement, Experimentation.

Keywords
dynamic analysis, network analysis, complex network, experimental validation, software maintenance.

1. INTRODUCTION
The evolution of large scale software, and our ever growing dependence on software in daily life, has led to understandable challenges in software engineering [1]. Evolution of software systems may imply superficial changes that do not have impact on the functional integrity, but often the implied changes go deeper in terms of structural integration of the system with significant impact on functional integrity of the system. Object oriented software viewed in a language agnostic manner, i.e. conceptually, can be represented as a complex network of message interactions between constituent classes/objects through their methods [2]. Conventional assumption is that when a system grows in size, it also grows in complexity. Software systems are similar to other real life systems in this respect.

To manage change in software systems, and to quantify the likely impact of changes imposed by system evolution, it is necessary to have methods that can measure and predict the likely impact of such changes. The techniques used to measure and analyse software can be broadly classified into two strands describing the process of measurement employed. These are static and dynamic analyses. Both approaches to software analysis can be challenging, and these challenges are compounded by factors like size of software and language environment. Static analysis captures a partial picture of the system, emphasising the measurement of the parts of system’s behavioural repertoire that are part of the actual usage pattern of the system [3],[4]. Network analysis [2, 17, 25] of the data generated by static or dynamic analysis of the software system promises an insight into the links between structural and functional features of the system [25]. However, most network analysis methods are based on theoretical studies of networks and have relatively little predictive experimental validation that support the claims about their ability to find system components with high functional importance and predict effectively the functional importance of such components.

In this work we focus on combination of dynamic analysis and network analysis techniques in order to evaluate the validity of the latter in terms of predicting functionally important components of software systems. We chose as our test bed the JHotDraw 6.01b software (www.jhotdraw.org), developed as an experiment in software design [14]. This software has been used earlier for demonstration of software analysis methods and as such provides some level of comparability for our results with results of previous works. While we did our work in the context of Java, the presented analysis techniques are language environment agnostic, and are applicable to any dynamic software analysis data represented as a network.

The paper is structured as follows. First we review briefly the relevant background. Next we present our analysis methods, the data that we used and our results. Finally we close the paper with a brief discussion and conclusion section.
2. STATIC, DYNAMIC, AND NETWORK ANALYSIS OF SOFTWARE

The basic concepts of structured design together with the definition of software metrics, including the application of such metrics is discussed in [6], in the context of non object oriented programming environments (FORTRAN, C). The two main concepts, coupling and cohesion of software components play a major role in analysis of object oriented software [6],[8].

Coupling is a measure of strength of modular interconnection, and cohesion is a measure of intermodular functional relatedness. Let us consider a set of classes, composing a software system, each containing its own set of distinct methods. The classes in this set interact with each other by exchange of messages, i.e. C1 invokes a method in class C2. These interactions define the interdependence between classes, and can be considered as a form of coupling. Let us consider a class as a functional entity, having as constituents the variables and methods of the class. Cohesion is a conceptual measure of functional relatedness of such constituent elements composing the class.

Static analysis is a way to measure these structural properties of a software system by inspection of source or binary code, against a model, [4,7], (designed to encapsulate desirable structural properties) without executing the software being analysed. In case of object oriented software, a set of six metrics (CBO, RFC, LCoM, DIT, NOC, and WMC) were defined in [8]. These metrics form a de-facto standard, and various derivatives exist which can be found in [9]. Static analysis is widely used, however provides only a summative view of the software system.

In case of object oriented software, that makes use of concepts like, polymorphism, inheritance, and dynamic binding, runtime behaviour cannot be easily predicted by the use of static metrics alone. Dynamic analysis investigates the software at runtime. This involves executing the software system, for a predetermined scenario, to collect execution trace data on properties of interest. Profiling is a common way of performing runtime analysis; however it is possible to instrument software, statically or at runtime, to collect customised data. This can used to study object level interactions, measure dynamic coupling [10], study object centric dynamic data and control flow [11], or predict temporal execution paths [12] amongst other things. This method of collecting runtime data is commonly referred to as ‘tracing’ or ‘execution tracing’, and can produce large amount of information. The resulting data is often noisy, which makes interpretation of the data challenging. Noise reduction strategies are used to remove irrelevant data [13]. A related challenge is the visualisation of such complex data aimed to support meaningful analysis [14].

Software systems are complex systems, with complex inter and intra component interaction relationship. These relationships can be considered as a complex graph or network, represented as a set of nodes and edges. Such representations facilitate quantification of some aspect of complexity, which can support reasoning about a system. Such reasoning can be further supported by intuitive visualization of the system [15-17].

Networks are defined by characteristic properties, and few major types of networks are: exponential random network, scale-free network and small–world network [18-21]. The premise of network analysis is that structural integrity of the graph representation of a system is closely related to the functional integrity of the system. This assumption if true, may lead to deriving the functional importance of system components, by analysis of structural properties of the graph or network.

Network analysis methods include methods that establish the type of networks, methods that quantitatively measure the structural integrity of the network [22], and methods for the analysis of structural properties of networks aimed to determine key components that contribute mostly to the structural integrity of the network. The type of network can be established by analysing the distribution of node connectedness, and structural integrity can be measured for example by calculating the average shortest path length, or average clustering co-efficient of the network [23, 24].

Recent years has seen active development in network analysis, leading to development of new methods, which includes, calculation of connectedness of nodes, and implied connectedness of edges, determination of frequent non-trivial network motifs, and many others.

Network analysis methods have been applied to data resulting from software analysis [15]. For example, such analysis established that the static class interaction network of various software systems is similar to scale-free networks [2]. A recent application of network analysis to software evolution uses this kind of analysis to find components that change in an unusual manner that may indicate their functional importance [2].

3. RESULTS

3.1 Assumptions and expectations

We consider software as a network of interactions between classes, where the interactions are method calls originating from one class and invoking a method of another class. Considering earlier results [16, 17, 25] we expect that this kind of network representation of large scale software has features that imply that the application of network analysis methods is meaningful and can lead to the determination of functionally important components of the software system. The key requirement for this is that the network is such that the likelihood of finding highly connected nodes in it is much larger than the likelihood of finding such nodes in random networks with exponential node connectedness distribution. Note that in networks of the latter kind randomly chosen nodes are likely to have similar contribution to the structural integrity of the network, which makes pointless the search for network components with significantly higher than average contribution to the structural integrity of the network.

Our key assumption is that software components that have high importance for the structural integrity of the network representation of the system have also high functional importance within the software system. Thus network analysis methods should be able to help the identification of software components that contribute critically to the system during its runtime execution. This should help the design and management of the evolution of the software system by identification of potential vulnerabilities that require extra effort and attention.
3.2 The data

To generate our dynamic analysis data we used the TPTP Proebkit agent (www.eclipse.org/tptp). The agent tracks the entry and exit of methods and analyses the stack trace following the entry into the method. At the time points of entry and exit checking the Proebkit agent logs the execution of the program. Following the entry phase the agent investigates the stack trace in order to determine the current class, the caller class, and the current class method that has been called by caller class. The data that we analysed included around 900,000 (caller class, called class, called method) entries for each run. We generated this reproducing each time the same operation sequence as the one used in [14] – i.e. we generated three drawing panels, placing on each after being generated five drawing objects. We also used the Java NetBeans profiler (netbeans.org) to generate call frequency data for each method that were used during the set operation sequence using the JHotDraw 6.01b.

The code of the JHotDraw 6.01b has over 66K lines of code and includes 344 classes with a few thousand methods that can be called. We found 195 classes that were active during our sequence of operation. There were 817 methods of these classes that were called during the runs of the software. To generate a network representation of the software we used the above described dynamic analysis data and we ignored the direction of method calls i.e. the method calls are represented as edges and not arcs in the graph that has as nodes the classes that were active during the execution of the program.

We calculated the frequency weighted connectedness values for each nodes of the network (i.e. the number of edges that are linked to the node considering the frequency of usage of the methods corresponding to these edges) We note that an edge represents a method call originating from a given class, the invocation of the same method by another class is represented by another edge. Considering the connectedness values we calculated the distribution of these values in order to check that the network representation of the analysed software system satisfies the assumptions of network analysis methods (i.e. it does not follow an exponential distribution and the likelihood of highly connected nodes is much higher than this likelihood in random network with exponential node connectedness distribution). Figure 1 shows a network representation of the analysed software and the estimation of the connectedness distribution. We found that the log(connectedness) values follow a linear distribution, implying that the connectedness distribution of the network is log-linear – see equation (1) below.

\[ p(x = a) = -\frac{4.9 \ln(a) + 45.167}{a} \]  

The log-linear distribution of node connectedness values means that the likelihood of highly connected nodes is much higher than the corresponding likelihood in random networks with exponential node connectedness distribution. This implies that it makes sense to use network analysis of this software representing network in order to find component with high structural integrity contribution. These components are then expected to have also high contribution to the functional of the system according to our assumptions.

3.3 Network analysis

We applied network analysis methods to determine the structurally most important edges of the network. According to our assumptions we expect that these edges represent the most important method calls in the software system.

To find important edges we used three network metrics: a) hub connection score (HCS), which is the product of the connectedness values of the nodes connected by the edge – equation (2) where \(e\) is an edge, and \(v(n)\) and \(v(m)\) are connectedness values of the nodes \(n\) and \(m\) connected by the edge \(e\):

\[ HCS(e) = v(n) \cdot v(m) \]
b) the edge betweenness score (BWS), which is the number of shortest paths between any two nodes of the network that contain the considered edge – here the length of an edge is calculated as the inverse of the usage frequency of the edge (i.e. if the method call represented by the edge is used $f$ times, the length of the edge is considered to be $1/f$) – equation (3) where $e$, $e_k$ are edges and $f(e_k)$ is the usage frequency of the edge $e_k$:

$$BWS(e) = \sum_{i \neq k} \frac{1}{\sum_{j=1}^{k} f(e_j)} \quad \forall \{e', ..., e'_k\} : \begin{cases} \text{nodes}(e_i) \cap \text{nodes}(e'_{j}) \geq 1, \\ \text{nodes}(e_i) \cap \text{nodes}(e'_{j'}) \geq 1, \\ \text{nodes}(e_i) \cap \text{nodes}(e_{j'=i}) = 1, \\ \text{nodes}(e'_{j'}) \cap \text{nodes}(e_{j'=i}) = 1, \\ \end{cases}$$

(3)

For each network metric we ranked the edges. We also considered combinations of these rankings of edges. In order to combine rankings we used two methods. First, we calculated the sums of rankings – i.e. if a method is ranked $r_1$-th according to metric $M_1$, and $r_2$-th according to metric $M_2$, then the combined score of the method for the combination of metrics $M_1$ and $M_2$ is calculated as $s(M_1, M_2) = r_1 + r_2$. Then we re-ranked the method according to this ranking-based combined score. Second, we calculated the combined score as the product of rank values, i.e. $s(M_1, M_2) = r_1 \times r_2$. Then again, we re-ranked the method according to the combined ranking-based scores. In addition we also calculated combined rankings for all three scoring methods using both approaches (i.e. sums and products of rank values).

The reason for choosing this way of combination of rankings is that the scores calculated according to the different network analysis methods are not necessarily comparable (e.g. one may be a magnitude larger than the other one for all highly ranked edges). Since the distribution of the score values is likely to be not normal, normalisation and calculation of a z-score (i.e. normalised value = (value – mean)/(standard deviation)) is not meaningful, and consequently normalised scores cannot be used.

$$CFS(p) = \sum_1 f(e_j) \quad e_j \text{ represent calling of } p \quad (4)$$

**Table 1. The top-2 ranked methods according to each individual and combined metric**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Rank</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>standard.StandardDrawingView.tool</td>
<td>1</td>
<td>standard.StandardDrawingView.tool</td>
</tr>
<tr>
<td>2</td>
<td>standard.StandardDrawingView.paintComponent</td>
<td>2</td>
<td>framework.FigureAttributeConstant.getName</td>
</tr>
</tbody>
</table>

**Table 2. Combined Rankings**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Rank</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>framework.FigureAttributeConstant.hashCode</td>
<td>1</td>
<td>contrib.AutoscrollHelper.Constructor</td>
</tr>
<tr>
<td>2</td>
<td>figures.FigureAttributes.get</td>
<td>2</td>
<td>standard.StandardDrawingView.tool</td>
</tr>
</tbody>
</table>

**Table 3. Combined Rankings**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Rank</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>framework.FigureAttributeConstant.hashCode</td>
<td>1</td>
<td>contrib.zoom.ZoomDrawingView$2.mouseMoved</td>
</tr>
<tr>
<td>2</td>
<td>standard.StandardDrawingView.tool</td>
<td>2</td>
<td>figures.AttributeFigure.getAttribute</td>
</tr>
</tbody>
</table>

**Table 4. Combined Rankings**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Method</th>
<th>Rank</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>standard.StandardDrawingView.tool</td>
<td>1</td>
<td>contrib.AutoscrollHelper.Constructor</td>
</tr>
</tbody>
</table>
necessarily for the calculation of a valid combined score. This leaves us the rank-based calculation of combined scores as the method that is sufficiently justifiable in sense of combining comparable values in order to avoid domination of the combined ranking by only one of the considered rankings.

The two highest ranked methods for each network analysis metric based rankings and for all combined score rankings are presented in Table 1.

To establish how good the network analysis metrics are in predicting functionally important methods, we evaluated the functional importance of each of the top-50 identified methods (according to all considered rankings). To do this we disabled the methods one-by-one and tried to run the same operation sequence with the modified software. We considered a method functionally important if the disabling of the method implied the crashing of the software or it caused significant dysfunction of the software (e.g. the software does not crash but does not allow to open drawing panels). We made sure that we applied minimally invasive disabling of methods, which were expected to not cause trivial errors – e.g. if a pointer to an object of a certain type is expected as output of a method call, not providing any output causes a trivial error – in such case we generated a default object of the right class, which was provided as the output of the method without executing the original contents of the method.

The performance of a network analysis metric or of a metric combination is calculated as the percentage of methods that are identified as important on the basis of the metric(s) and which turn out to be important according to the functional evaluation of methods out of the top-n ranked methods turn out to be functionally important then we have 100% performance for the ranking up to the n, if only half of the top-n methods turn out to be functionally important then the performance of the ranking for n is 50%. The performance results are presented in Figure 2.

The results show that most of the top-15 ranked methods according to the HCS and CFS analysis metrics are highly functionally important, while this is not the case to the same extent for ranking of methods based on the BWS metric. In case of combined metrics the results show that the additive combination or rank orders did not produce better combined metrics than the individual metrics. However, in the case of rank product based combination of metrics the performance of combined metrics improved for the top-50 methods, except for the metric combination HCS – BWS.

The results show that the HCS – BWS combination in both cases of rank combinations (sum and product) led to a combined ranking of methods that identified a smaller percentage of functionally important methods than the rankings based on the individual metrics. This indicates that these two metrics are likely to identify different kinds of methods as important and combining the separate rankings in a sense cancels correct identification of functionally important methods. Notably the triple combination of rankings does not perform better than paired combinations of rankings, which is again is likely to be due to the cancellation effect of the combination of HCS and BWS rankings.

Figure 2. The performance of the considered ranking methods in terms of the proportion of correctly predicted functionally highly important methods: A) individual metric-based rankings; B) combined metrics using the sums of ranks; C) combined metrics using the products of ranks.
4. DISCUSSION AND CONCLUSIONS
In this paper we analyse the effectiveness of network analysis methods to identify functionally highly important components (methods) of software systems in the context of dynamic analysis of these software. Since network analysis is a key method to untangle the complexity of large-scale systems it is important to establish the validity of such methods for the determination of important component of complex software systems. We found that the considered network analysis methods can find many functionally important methods, however they also rank high methods that are functionally not so important. The combination of ranking methods led in some cases to improved ranking performance in sense of high proportion of highly ranked method being functionally highly important. However the combination of HCS and BWS metric based rankings did not improve this kind of performance of the ranking.

The results indicate that network analysis methods may help in making sense of complex data generated by dynamic analysis of large-scale software systems. However, the results also show that proper validation of these methods is required in order to make sure that system components determined by them as highly important are indeed functionally highly important within the software system.

The potential of network analysis combined with dynamic analysis of large-scale complex software systems is important. Such methods applied to dynamic analysis data may help to direct software evolution [2,17,25] by predicting patching needs and may also help identifying software components that require preservation and slow evolution in order to guarantee the expected levels of dependability of the software system [5].

5. REFERENCES