Not All Networks are Equal:
Empirical Analysis of Flexibility and Controllability in Software Systems

Douglas Feitosa Tome & David A. Broniatowski
The George Washington University
Washington, DC

Abstract

Scholars have posited that the architecture of a system drives its lifecycle properties. For example, Moses [1] represents “generic architectures” as networks of resource flows, which he states are related to flexibility and controllability in systematic ways. This study aims to investigate these claims empirically in the context of software systems. We compare and contrast the architectures of six distinct software packages at different levels of abstraction by computing structural metrics of laterality and verticality derived from Moses’ work. Next, we investigated the relationship between these structural metrics and Moses’ proposed measures of flexibility and controllability, as well as new measures that we propose. We found a relationship between system architecture and flexibility that is opposite to what was expected in the case of dependency networks. This suggests that measures of system lifecycle properties depend strongly on the network representation of the system used (e.g., dependency vs. control flow), thus requiring different approaches to measuring system properties.

Keywords
Systems architecture, flexibility, controllability, software systems, network analysis.

1. Introduction

There has been a growing interest in the research community in investigating the relationship between the systems’ architectures and their life-cycle properties. In particular, scholars have focused on exploring the mechanisms through which the structure of a system drives its properties. The underlying assumption is that structural characteristics both at the micro- and macro-level determine system-level properties. Moses [1] developed a theoretical framework that laid a foundation for the study of the relationship between systems architecture and life-cycle properties, especially in the case of flexibility. Furthermore, Broniatowski and Moses [2] expanded the treatment of this topic by discussing the relationship between systems architecture and controllability. In these studies, the researchers posited that the flexibility and controllability of systems are systematically related to their architecture.

These claims have not yet been empirically tested. It is therefore relevant to conduct a formal study to investigate them since they have significant implications for the design of systems (e.g., one could design systems that have desired characteristics by carefully crafting their architectural layout). Thus, the purpose of this paper is to verify to what extent structural measures of software systems are related to two specific lifecycle properties, namely flexibility and controllability. We chose to study software systems due to data availability and ease of access (e.g., open source software packages freely available on the web). Thus, this study intends to compare and contrast the architecture of six software packages across two levels of analysis – source code directory, and function – and to determine how they relate to measures of flexibility and controllability.

2. Literature Review

There are many alternative definitions of systems architecture in the scientific literature. The one adopted in this paper is provided by de Weck and colleagues [3]: the structure of a system refers to the interconnection pattern of its elements whereas the architecture of a system is a larger concept that encompasses both the structure and function of its components. Furthermore, a number of scholars have discussed the relationship between systems architecture and lifecycle properties. Similarly, Moses [1] notes that “there are several historical examples where structure was analyzed early on in order to gain a better understanding of systems.”

Moses [1] posited that there are four generic architectures: tree hierarchy, layered hierarchy, decentralized network, and team. A graphical depiction of each generic architecture is presented in Fig. 1. In these graphs, nodes
represent elements of a system whereas links denote an association between nodes that usually indicates a flow of some resource such as information. The different architectural patterns reflect varying degrees of hierarchy and interconnectedness. In particular, hierarchical structures (i.e., tree and layered hierarchies) clearly present sets of nodes displayed in multiple levels whereas it is difficult to distinguish such structural pattern in nonhierarchical structures (i.e., decentralized network and team). Hierarchical levels arise due to the directionality of the network (i.e., flows from the “top” to the “bottom” of the network). Structures characterized by high levels of interconnectedness (i.e., layered hierarchy and team) have clusters of highly connected nodes as opposed to loosely connected structures that lack such cohesive patterns (i.e., tree hierarchy and decentralized network).

Moses [1] contends that “well-designed architectures of particular systems will tend to rely on a mix of these generic architectures”. In other words, the conceptualized generic architectures are neither mutually exclusive nor collectively exhaustive.

![Diagram of generic architectures](image)

**Figure 1:** Proposed distribution of the generic architectures posited by Moses [1] in the quadrants determined by crossing verticality vs. laterality

Many scholars have also attempted to define flexibility. De Weck et al. [3] noted that this property has become an umbrella term related to the ability of a system to respond to change. We agree with Moses [1], who considers a system flexible “if one can implement classes of changes in its specifications with relative ease”. By observing that systems that have many alternative paths tend to cope with changing environments reasonably well, Moses [1] defined “the flexibility of a system as the number of paths in it, counting loops just once”. Note that this measure of flexibility applies to the generic architectures defined above (i.e., network representations that rely on flows).

Controllability is another important property that has been defined in various ways in different contexts. In this study, we follow the approach of Broniatowski and Moses [2], where control is associated with hierarchy such that “higher elements in a hierarchy have the capacity to control lower elements”. Building on this intuitive interpretation of controllability, the authors [2] defined a measure of this property “as the total number of paths in the graph (i.e., its flexibility) divided by 1 + the total number of loops in the graph”. In the next section, this idea of associating control to hierarchy will be used to derive an alternative measure of controllability in the context of dependency representations of software systems. In sum, the flexibility of a system tends to increase when it presents higher levels of interconnectedness whereas its controllability tends to increases as the network becomes more hierarchical.

3. Methodology

We gathered data from software systems to empirically verify whether flexibility and controllability are related to generic architectures in the ways predicted [1, 2]. Here, a software system is defined as the source code of a computer program. Note that this definition of a software system refers to the state of a software package prior to any compilation of its source code. In other words, a software system denotes a collection of source code files as opposed to an executable program. Furthermore, source code files exist in a well-defined directory structure that is determined by the developer(s) of the software package. This structure is more or less arbitrary and MacCormack et al. [4] note that it reflects how the programmers view the software system.

The present research conducted an analysis of the source code of the following packages: Microsoft Word for Windows version 1.1a (MS Word v.1.1a), the core of the Replicant OS version 4.2 (Replicant v.4.2), the core of the Android OS version 5.1 (Android v.5.1), the JavaScript core of WebKit release 189384 (WebKit r.189384 – JavaScript), the web core of WebKit r.189384 (WebKit r.189384 – Web), and the web inspector user interface component of WebKit r.189384 (WebKit r.189384 – UI). We downloaded the source code for these software packages, which are freely available online. This study followed recommendations of Knoke and Yang [5] regarding the design
of a network analysis-based research. Specifically, the analysis of each system at the directory level used the directory structure as the sampling unit and the level of data analysis encompassed the complete network; the relational form was any dependency-creating relationship, and the relational content was the dependency between two directories. A node in the dependency representation of each software system represents a source code directory. For the analysis of each system at the function level, the sampling unit was the control flow of a function in a source code file; the level of data analysis was set as a complete network; the relational form was flow of control; and the relational content was control. A node in the control flow representation of each software system denotes a basic block. Hennessy and Patterson [6] defined a basic block as “a straight-line code sequence with no branches in except to the entry and no branches out except at the exit”. Note that these are different abstractions of the same system.

We used the source code of each software package to generate the network data. As noted previously, a network node may represent either a source code directory or a basic block, and a link from node A to node B may represent either a dependency-creating relationship (e.g., a function in source code file A calls another function in source code file B) or a flow of control in the execution of a function. The network data – nodes and links – were extracted from the raw source code files using Understand™. This software application is a static source code analyzer, meaning that it takes as input the pre-compiled source code files of a software package and creates a comprehensive database of the dependencies found in the analyzed package at the directory level as well as of the control flow graphs of the functions included in the code. Regarding the types of dependency-creating relationships effectively used in the proposed model, Understand™ captures the following kinds of dependencies: address uses, calls, includes/imports, inherits, initializes, implements, modifies, overrides, sets, throws, and uses. The resulting network data is binary at both levels of analysis.

For each software system, we randomly sampled a large subset of its functions to perform our analyses (sampling all functions was prohibitive due to computational costs). The total number of functions (i.e., number of control flow graphs) in each package and the computed sample size are presented in Table 1. Every metric described in the remaining of this section was computed for the directory structure and the set of randomly selected control flow graphs for each analyzed system. For control flow network representations, system metrics were derived as the weighted average of the metrics of the individual control flow graphs for a given software package, weighted by the number of nodes in each control flow graph.

<table>
<thead>
<tr>
<th>Package</th>
<th>Number of Control Flow Graphs</th>
<th>Sample Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS Word v. 1.1a</td>
<td>4135</td>
<td>572</td>
</tr>
<tr>
<td>Replicant v. 4.2</td>
<td>4702</td>
<td>582</td>
</tr>
<tr>
<td>Android v. 5.1</td>
<td>28367</td>
<td>649</td>
</tr>
<tr>
<td>WebKit r189384 – JavaScript</td>
<td>8828</td>
<td>618</td>
</tr>
<tr>
<td>WebKit r189384 – Web</td>
<td>43464</td>
<td>654</td>
</tr>
<tr>
<td>WebKit r189384 – UI</td>
<td>3405</td>
<td>556</td>
</tr>
</tbody>
</table>

3.1 Layer Extraction

The characterization of the architectures of the various collected networks started with the identification or extraction of their layers. The approach taken in this study is based on the algorithm used by Estrada et al. [7] to identify a multipartite structure in an arbitrary network. This algorithm yields a set of quasibipartite subgraphs of a given network. However, the resulting subgraphs may overlap (i.e., share the same node). Therefore, we developed a method to consistently select which subgraphs should compose the layers of the network. We selected layers according to the following algorithm: 1) select the largest subgraph (by number of nodes) in the set of quasibipartite subgraphs; 2) if there is a tie, select the subgraph whose number of intraset links is larger; 3) the selected subgraph becomes a layer of the underlying network; 4) discard all subgraphs that have any overlapping element with the last selected subgraph; 5) if there is any remaining subgraph in the set, go back to 1.

This algorithm yields a subset of the original set of quasibipartite subgraphs. Each subgraph in the resulting subset, then, corresponds to a layer. The rationale for the algorithm is based on two guiding principles: 1) the larger subgraph should be chosen to minimize the total number of layers and 2) the number of intraset links should be the criteria for breaking a tie between subgraphs of equal size because it maximizes the density within the resulting layer. We believe that this approach tends to capture the wider, most prominent patterns in the network by identifying large, dense clusters of nodes. In general, the application of the proposed algorithm may result in a number of nodes not being included in any layer. In such cases, each individual node is treated as a one-node layer, thus allowing for the identification of highly segregated nodes in the network.
3.2 Verticality

Once the layers of the network have been extracted, the extent to which they exhibit “verticality” and “laterality” was measured to characterize their architecture. One can think of these structural constructs as roughly analogous to the concepts of hierarchy and interconnectedness discussed by Moses [1]. Verticality is measured by computing the flow hierarchy, as defined by Luo and Magee [8], of the “layer-equivalent” representation of a given network. Specifically, Luo and Magee [8] defined flow hierarchy as the “the percentage of links that retain their overall direction in the network”. In other words, the ratio of the number of links that are not included in any cycle over the total number of links in a network. However, the direct application of this measure to a network fails to consider its (possible) underlying layers. In turn, this may inappropriately lead to intralayer links that are part of cycles detracting from the hierarchy of the network (i.e., the overall directionality of the network is not affected by intralayer links).

Here, we propose that the computation of the flow hierarchy of the “layer-equivalent” representation of a network more accurately captures its overall directionality. The “layer-equivalent” representation would be the graph generated by aggregating all nodes within a network layer as a single node and ignoring the resulting self-loops and parallel edges. The result is a graph that captures only the inter-layer flows and, hence, is suitable for computing the flow hierarchy, and thus the verticality, of the underlying network.

3.3 Laterality

We propose a measure for the laterality of a network as the weighted average of the density of its individual layers, weighted by the number of nodes in each layer. Thus, the presence or absence of interlayer links will not affect the resulting metric. Fig. 1 shows how the proposed measures of verticality and laterality relate to the four generic architectures posited by Moses [1].

3.4 Flexibility

When representing a software package at the directory level, the average nodal indegree of the resulting dependency network provides a measure of its rigidity. Rigidity is defined as the opposite of flexibility – the relative difficulty of changing a system in response to external changes. Consider, for instance, that one needs to modify the source code of a given software package in order to respond to a change in user requirements. If the modification requires changing a single source code file, then all the source code files which depend on the modified one may have to be altered accordingly. In case the interfaces are well designed and the changed source code file preserves the same interface as the original one (e.g., same arguments), no further changes may be necessary. Therefore, the average nodal indegree of a software dependency network is an upper-bound measure of how many elements of the source code, on average, need to be changed when one changes a single element in the source code – i.e., its rigidity. Note, as cited previously, that the function of the nodes of the network plays a critical role in the proposed definition of rigidity.

In order to control for network size, the normalized rigidity of a network that has \( g \) nodes is defined as its average nodal indegree divided by \( g - 1 \). Furthermore, one can define the normalized flexibility for ease of interpretation as \( 1 \) minus its normalized rigidity.

For the software representations that emphasize flow of control (i.e., function-level analysis), Moses’ [1] measure of flexibility is appropriate since it captures the number of alternative paths through which control may flow. As cited previously, Moses [1] defined flexibility as the total number of paths in a directed graph with loops being counted once. Here, the number of paths from the “start” node to the “end” node of a control flow graph was estimated by enumerating the simple paths in the network between these two nodes. A modified version of depth-first search proposed by Sedgewick [9] was used to provide an estimate of such paths.

3.5 Controllability

With respect to software dependency networks, controllability can be defined in terms of the node that exhibits an optimal configuration of its links for the purposes of acting as a central controller. Consider, for instance, that if all nodes depend on a single node, then the latter node controls the behavior of the entire network since in the absence of this node the network could not work properly. This controller node resembles the central node of a star graph in which all nodes point to the central one (i.e., centralized control). However, if this controller node at the same time also depends on other nodes, this will decrease its ability to control the network since the controller itself relies on other nodes. In this case, the controller node is similar to the central node of a star graph in which all links are bidirectional. Normalized controllability \( c_N \) can, then, be defined as:

\[
    c_N = \max \left( \frac{\text{nodal indegree} - \text{nodal outdegree}}{\text{nodal indegree} + \text{nodal outdegree}} \cdot \frac{\text{nodal indegree}}{g - 1} \right)
\]

(1)
Notice that the denominator of the first term in the product above was used for scaling the metric relative to the size of the network whereas the second term was introduced in order to account for nodes whose both indegree and outdegree are very low (e.g., in a network with a hundred nodes, a node whose outdegree is zero and indegree is one would have the maximum controllability of 1 had the second term not been introduced).

Regarding software representations that focus on flow of control, the measure of controllability defined by Broniatowski and Moses [2] accurately captures the degree of control. Specifically, they posited that controllability can be measured as the ratio of the total number of paths in a network over the total number of loops plus one. Here, we estimated the total number of cycles in a control flow network by enumerating its simple cycles using Johnson’s algorithm [10]. Although the number of simple cycles and total cycles in a network are not the same in general, closed walks can be decomposed into simple cycles in a directed network (i.e., directed cycles) and, hence, the number of simple cycles is as first-order proximate metric for the number of loops in a directed network.

We tested the following hypotheses at both the directory and function levels of analysis: 1) the flexibility of a system increases when its laterality increases, and 2) the controllability of a system increases when its verticality increases.

4. Results

Correlations between laterality and flexibility as well as between verticality and controllability are presented in Table 2 for both the directory and function network representations of the systems under analysis. The respective p-values are also displayed. Although we did not find a significant correlation in the case of control flow networks, sample size was small. Furthermore, we found a significant correlation between laterality and flexibility in the case of dependency networks, although in a manner opposite to Moses’ [1] predictions.

<table>
<thead>
<tr>
<th>Level of Analysis</th>
<th>Laterality vs. Flexibility</th>
<th>Verticality vs. Controllability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Correlation</td>
<td>p-value</td>
</tr>
<tr>
<td>Directory Structure</td>
<td>-0.97</td>
<td>0.0014</td>
</tr>
<tr>
<td>Control Flow Graphs</td>
<td>-0.62</td>
<td>0.18</td>
</tr>
</tbody>
</table>

Verticality and controllability are positively related for both dependency and control flow system representations, as predicted by Moses [1], although these results are not significant due to small sample size. This suggests that verticality and controllability may be related in a similar manner in different network representations.

It is important to note that the relatively small sample size (i.e., six software packages) and the consequent limited statistical power of the present analysis may be the reason why we could not observe statistically significant correlations in many cases. Hence, further analysis of larger samples is necessary to investigate the relationship between the proposed structural metrics and system properties in general. Furthermore, it was not possible to compute the discussed metrics for the two largest control flow graphs in the Replicant v. 4.2 sample as well as for the three largest ones in the WebKit r189384 – JavaScript sample. Future studies will address this limitation with alternative computing techniques and/or computational infrastructure.

Additionally, although useful as a preliminary analysis, the adopted sampling methodology, averaging method for metrics of control flow network representations, and the proposed layer extraction algorithm may have an impact on how the conceptualized structural metrics and system properties are related. Future work will examine the effect of each of these modeling decisions on the ability of accurately measuring the constructs discussed here.

Fig. 2 provides further insight into the relationship between structural metrics and system properties by examining their sample distributions in the case of the control flow graphs of Android v. 5.1. Note that the laterality and flexibility distributions both are asymmetric (skewness of 4.2 and 18.23, respectively) but the first is moderately peaked while the second is highly peaked (kurtosis of 21.2 and 358.4, respectively). While the verticality and controllability sample distributions are both positively skewed (skewness of 1.1 and 22.7, respectively) the first is slightly flat and the latter is highly peaked (kurtosis of -0.44 and 538.0, respectively). Similar observations can be made considering the remaining software packages. These differences suggest the presence of outliers in the flexibility and controllability distributions without counterparts to the same extent in the laterality and verticality distributions, respectively, compounds the complexity of correlating these metrics and may help to explain our current results.
5. Conclusions

In this paper, we present preliminary empirical measures of flexibility and controllability in complex systems, and use them to test hypotheses regarding aggregate properties of software systems architecture. Specifically, the present study explores the categorization of software systems in terms of laterality and verticality. Our findings suggest that the measures of system lifecycle properties may depend on the adopted network representation of the system. In other words, the particular abstraction used to model a given system as a network has an impact on how the structure of this specific representation relates to properties of the system. In sum, different system abstractions may lead to different conclusions regarding the relationship between system architecture and lifecycle properties. Thus, future work should examine how the choice of network representation interacts with measures of system lifecycle properties.

References


