Inter-Package Dependency Networks in Open-Source Software

Nathan LaBelle and Eugene Wallingford
Department of Computer Science
University of Northern Iowa
Cedar Falls, Iowa 50613
labelle@uni.edu

Abstract

The new subject of network science examines large graphs that model real-world, growing, and dynamic systems of interacting components. These networks form outside the realm of centralized control, but exhibit global organization and share common non-trivial properties. This research analyzes networks in Open-Source Software (OSS) at the package level, where dependencies link diverse software systems constructed by otherwise disjoint development groups.

Data was mined from two OSS repositories and interaction graphs were constructed. We show that this module coupling creates a clustered network with low separation between packages (a “Small-World”). We show that the distribution of edges in the graph is self-similar at all scales and skewed, yielding a power-law distribution. These properties indicate that package networks share natural organization patterns with previously studied systems in natural and social sciences. We argue that software networks are governed by similar dynamics. Finally, we discuss the implications of network structure on software engineering.

Introduction

Since Euler's 1735 solution to the Königsberg Bridges problem, a huge amount of knowledge has been built on the topic of mathematical graphs. In the last 5 years alone, at least 21,000 papers have been published on the topic of complex networks (National Academies, 2005). A network is a typically unweighted and simple large graph $G = (V, E)$ where V denotes a vertex set and E an edge set. Vertices represent discrete objects in a system, such as social actors, economic agents, computer programs, or biological producers and consumers. Edges represent interactions among these "interactons". For example, if software objects are represented as vertices, edges can be assembled between them by defining some meaningful interaction between the objects, such as inheritance or procedure calls (depending on the nature of the programming language used). In this paper, the term "network" generalizes large dynamic graphs where each agent has the ability to change its own set of links based on limited information, goals, and capacity.

In Open-Source Software (OSS) systems, applications are often distributed in the form of packages. A package is a bundle of related components necessary to compile or run an application. Because resource reuse is naturally a pillar of OSS, a package is often dependent on resources in some other packages to function properly. These packages may be third-party libraries, bundles of resources such as images, or UNIX utilities such as *grep* and *sed*. Package dependencies often span across project development teams, and since there is no central control over which resources from other packages are needed, the software system self-organizes in to a collection of discrete, interconnected components.

The collection of OSS interactions was chosen because it is large (about 22,000 packages with 84,000 interactions), and contains a diverse set of software developed by many people. Additionally, links between packages can be defined in discrete, definite ways; and the data set is free of extraneous data. OSS is also particularly relevant to modern computing, given the growth in popularity of Linux and UNIX-like operating systems in recent years. Finally, OSS repositories provide software engineering researchers with a view of a large collection of software that is used "in the wild", and developed using many methodologies, providing a unique opportunity to identify development practices with the ultimate goal of producing better software.

Previous Research

Real-world networks tend to share a common set of non-trivial properties: they have scale-free degree distributions following a power-law, exhibit the Small-World effect, and have a large connected component. These properties are non-trivial in the sense that they do no appear in randomly constructed graphs (Watts, 1999). Real-world networks such as the Internet (Faloutsos, Faloutsos, & Faloutsos, 1999), the World-Wide Web (Albert et al., 1999), software objects (Potanin et al., 2004), and networks of scientific citations (Lehmann, Lautrup, & Jackson, 2003; Redner, 1998) all have these properties.

The goal of this research is to show that these properties are also found in networks of package dependencies in OSS repositories.

The degree of a vertex v, denoted k, is the number of vertices adjacent to v, or in the case of a directed graph either the number of incoming edges or outgoing edges, denoted k_{in} and k_{out}, respectively. The distribution of edges in real-networks roughly follows a power law: $P(k) \propto k^{-\alpha}$. That is, the probability of a vertex having k edges decays with respect to some constant $\alpha \in \mathbb{R}^+$ which is typically between 2 and 3. This is significant because it shows deviation from randomly constructed graphs, first studied by Erdös and Rényi and proven to take on a Poisson distribution in the limit of large n, where $n = |V|$ (Watts, 1999). Also, it has been shown that dynamical systems undergoing phase transitions near the "edge of chaos" often display power-law behavior (Bak, 1996). The power-law distribution of edges implies that many vertices will not be highly connected and will have one or two edges, while some vertices will be "hubs" of the network and contain thousands of edges.

Real-world networks have two specific length-scaling features not found together in random networks: (1) a low characteristic path length and (2) a high degree of clustering (Watts & Strogatz, 1998). Together, these these properties are known as the "Small-World" (SW) effect, popularly known as "Six Degrees of Separation". Formally, a network is Small-World if $k \gg \ln(n) \gg 1$, $L \approx L_{random}$ and $C \gg C_{random}$. $k \gg \ln(n)$ places a lower limit on the sparseness, preventing the graph from becoming disconnected. $L \approx L_{random}$ indicates that the geodesic path length (unweighted number of hops) between vertices is, on average, of the same order as that of random networks. In real-world networks, the number of hops between vertices grows on the order of $\log(n)$. The clustering coefficient C is the propensity for local cliques to form, and is much higher in real networks than in randomly constructed networks. In more formal terms, if a vertex *v* is adjacent to vertices *u* and *w*, then *u* and *w* are more likely to be adjacent to each other than in a random network. The clustering coefficient measures this value, which is the likelihood of the neighbors of a vertex to also be neighbors.

The size of the connected component is the number of vertices such that there exists an edge traversal path between each vertex in the set. For real-world networks, the size of this set is approximately 90% or more, which is much larger than statistically expected in randomly-constructed networks. This fact makes Internet routing possible: at any given time, a large subset of the entire Internet is connected by some path. Those nodes that are not connected to the main component are disjoint and unreachable from most users.

Previous research in networks of software has focused on software at low levels of abstraction relative to the current research. Clark and Green (1977) found Zipf distributions (a ranking distribution similar to the power-law, which is also found in word frequencies in natural language (Zipf, 1965) in the structure of CDR and CAR lists in large Lisp programs during run-time. In object-oriented programming languages, several studies (Valverde & Solé, 2004; Wheeldon & Counsell, 2003) have identified the Small-World effect and power-law edge distribution in networks of objects or procedures where

edges represent meaningful interconnection between objects, such as inheritance or in the case in procedural languages, procedures are represented as vertices and edges between vertices symbolize function calls. These networks may be constructed either from source code by using syntax analysis; or from run-time analysis of objects stored on the program's heap. Similar statistical features have also been identified in networks where the vertices represent source code files on a disk and edges represent a dependency between files rather than objects or procedures (de Moura, Lai, & Motter, 2003). For example, in C and C++ one source file may *#include* another, thus creating a network of meaningful interaction inside a program. The Small-World property and a power-law distribution of edges has also been found in documentation systems such as JavaDoc (Wheeldon & Counsell, 2003).

Methodology

Data was gathered from the Debian GNU/Linux and FreeBSD systems, primary using the Advanced Package Manager (APT) tool for database querying. This data was gathered as "snapshots" of different system maturity levels (Unstable, Testing, and Stable) and different hardware architectures. The raw statistical data was processed using Minitab 14.

Dependency and Repository Anatomy

Several reasons for package dependencies have been identified:

1. Resource sharing. Multimedia resources such as graphics and sounds may be part of a package and required by another package.
2. Library dependency. Programs make calls to various libraries, such as libPNG or libc6.
3. Interpretation dependency. Programs may rely on language interpreters such as Perl or Python to parse their code and execute them.
4. Executable dependency. A program may call some other system executable, such as *grep* or *sed* for processing textual data.
5. Dummy references. Some packages are "umbrella" packages that exist to aid the user in installing other packages. This is the case where many packages are part of a large project, such as the X Window System.
6. "Provides". These "pseudo-packages" provide some function that is package independent. An example would be the "x-window-manager" package, where specific windowing management packages provide this pseudo-package, and are assumed to supply the same functionality, transparent to dependent packages.
7. "Conflicts". Sometimes, two packages which provide similar functionality will conflict with each other when installed in a system. An example is the "abiword" package, which conflicts with the "abiword-gtk" and "abiword-gnome" packages.

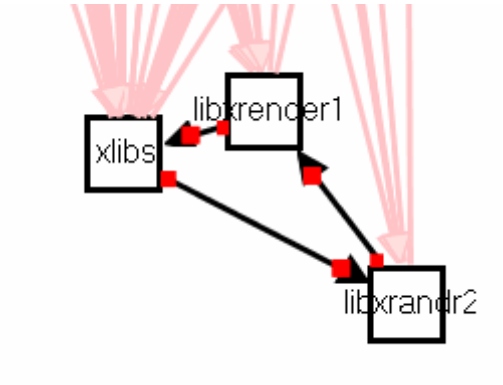

Figure 1: A cyclical dependency

Statistical Software

Customized software was constructed in Java to interact with APT, create the networks, and calculate the statistics. The JUNG (Java Universal Network/Graph Framework) libraries provided calculation of the clustering coefficient, component size, and diameter. In order to obtain the power-law fitting, the following steps were taken:

1. Cumulative tallies of and k_{out} and k_{in} were calculated. That is, the sum of the number of nodes having exactly k_{out} and k_{in} edges was calculated.
2. $k = k + 1$. Each package is dependent on itself.
3. $\log(N(k))$ was plotted against $\log(k)$ and fitted with linear regression.
4. The slope $m = \frac{\Delta y}{\Delta x}$ of the fitted line was calculated. The power-law exponent is $\alpha = \mid m \mid$.

Results

Results for the i386 distributions are given in Table 1. The i386/Unstable network is the largest architecture in Debian with $n = 22,264$ packages and $k = 84,437$ dependencies, giving each package an average coupling to 3.79 packages. For each network snapshot, L and C were in the SW range, since they have path lengths shorter than those of SW networks but have higher cliquishness. There are about 2,500 components, but the largest component contains 88% of the vertices. The rest of the vertices are disjoint from each other, resulting in a large number of components with only 1 vertex. The diameter of the largest component is 19. The distribution of outgoing edges, which is a measure of dependency to other packages, follows a power-law with $k_{out} = 2.33$ (Figure 2). The distribution of incoming edges, which measures how many packages are dependent on a package, follows a power-law with $k_{in} \approx 0.90$ (Figure 2). About 73% of packages depend on some other package to function correctly. The "Pure" dependencies in Table 1 are dependencies that are not categorized as Provides, Conflicts, or Dummy. Correlation

between k_{in}, k_{out}, and package size is not calculated because the normality assumption is violated.

Another way of categorizing the edge distribution is by ranking event sizes and frequencies, instead of comparing event sizes and frequencies directly. Each node is assigned a rank $1 \le r \le n$ based on the degree of the node, where a lower number implies a higher rank. $\log(r)$ is then plotted against $\log(k)$. If linear regression yields a close fit over several orders of magnitude (which is implied by log scale), then a Zipf distribution is present. Figure 3 shows the Zipf distributions for Debian i386/Unstable where $n = 22,264$ vertices. In Figure 3, k_{out} diminishes for large ranks, implying finite size effects in the number of outgoing dependencies. This is because no project is dependent on thousands of other projects for functionality, since a project encapsulating that much utility would be infeasible due to programmer and management limits.

	n	k	$\bar{k}$	Components	Largest Component Size
Unstable	22264	84437	3.792	2524	87.9%
Testing	21310	75699	3.552	2865	85.7%
Stable	19677	74642	3.793	2014	88.8%
Unstable Pure	17183	72501	4.219	1407	91.2%
	C	C_{random}	L	L_{random}	
Unstable	N/A	0.0017	N/A	7.509	
Testing	0.498	0.0016	3.441	7.862	
Stable	0.533	0.0019	3.352	7.415	
Unstable Pure	0.528	0.0024	3.056	6.774	

Table 1: Debian i386 Dependency Network Statistics.

In the Debian network, the 20 most highly depended-upon packages are libc6 (7861), xlibs (2236), libgcc1 (1760), zlib1g (1701), libx11-6 (1446), perl (1356), libxext6 (1110), debconf (1013), libice6 (922), libsm6 (919), libglib2.0-0 (859), libpng12-0 (622), libncurses5 (616), libgtk2.0-0 (615), libpango1.0-0 (610), libatk1.0-0 (602), libglib1.2 (545), libxml2 (538), libart-2.0-2 (524), and libgtk1.2 (474). The number in parentheses represents the number of incoming edges. The list is composed mainly of libraries that provide some functionality to programs such as XML parsing or that provide some reusable components such as graphical interface widgets. Because the most highly-connected package (libc6) is required for execution of C and C++ programs, we can infer that these are the most widely used programming languages. These “authority” packages form the backbone of the Debian system, just as large routers on the Internet (Faloutsos et

al., 1999) or informational hubs on the Web (Albert et al., 1999) are major contributors to connectivity.

Several non-i386 architectures were analyzed with similar results. A summary of the results are given in Table 2. In each case, the networks created fit the SW criteria and the distributions were skewed towards a power-law. The AMD64 architecture is significantly smaller than the others because software for AMD64 is in the process of being ported from other architectures. If AMD64 develops like the other architectures, more dependencies will be created as more packages are ported.

	n	k	$\bar{k}$	C	L	Largest Component Size
AMD64 Unstable	11061	15980	1.444	0.509	5.516	63.8%
AMD64 Testing	10504	14926	1.420	-	-	63.5%
Alpha Unstable	21537	83722	3.887	0.536	3.411	87.5%
Alpha Testing	20814	78198	3.756	0.490	3.543	86.5%
Alpha Stable	18687	70696	3.783	0.535	3.355	88.1%
HP-PA Unstable	21550	83323	3.866	-	-	87.1%
HP-PA Testing	20466	77918	3.807	0.544	3.402	86.1%
HP-PA Stable	18670	70599	3.781	0.536	3.357	88.1%

Table 2: Networks for AMD64, Alpha, and HP-PA Architectures.

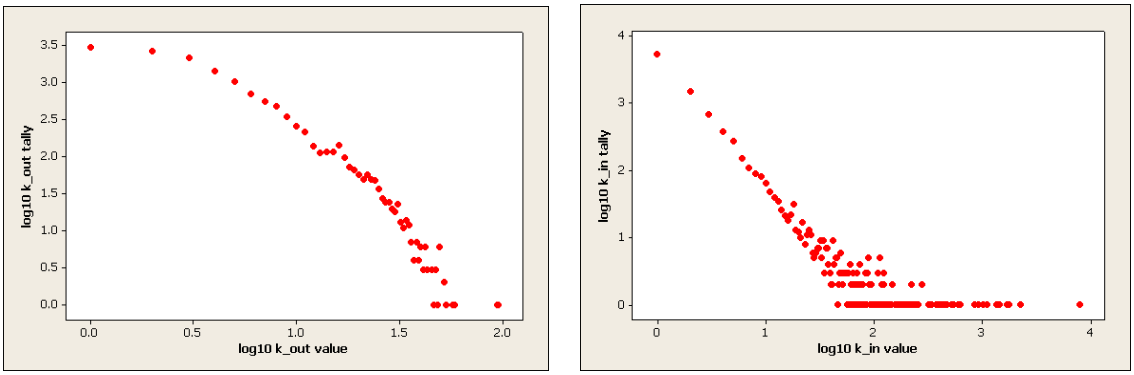

Figure 2: Log-log scatterplots of k_{out} and k_{in}, respectively, of Debian i386/Unstable.

The BSD network yielded similar results. This network was constructed from a single snapshot taken in October, 2004. In the case of both source code and binary dependencies, the networks were similar. For the source-code network, n =10,222 packages and

k =74,318 dependencies, giving an average of 7.27 dependencies per package. The SW statistics are $C = 0.56 \gg C_{random} = 0.00071$ and $L = 2.865 \approx L_{random} = 4.653$. The network has a diameter of 5 and component size of 72% of all packages. The remaining packages are disjoint from each other. The BSD source-code network has a power-law degree distribution with $a_{in} = 0.62$ and $a_{out} = 1.28$.

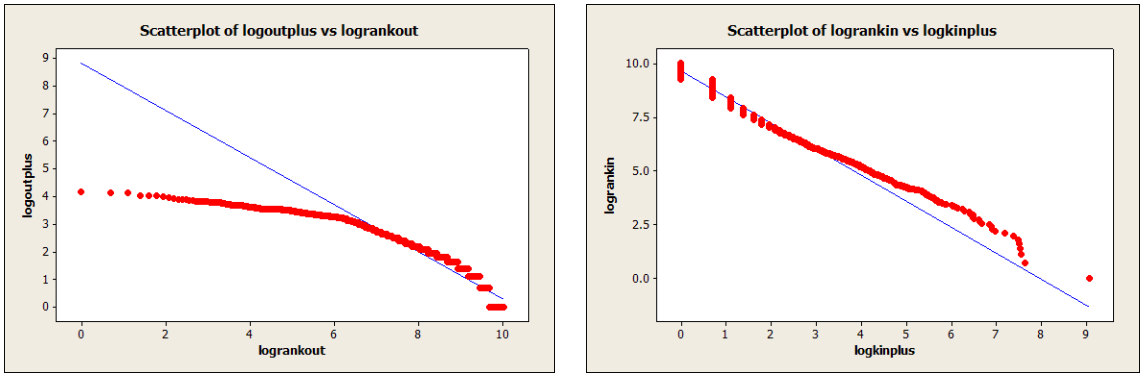

Figure 3: Zipf distribution of k_{out} and k_{in}, respectively, of Debian i386/Unstable.

Figure 4: A rendering of n=100 random packages.

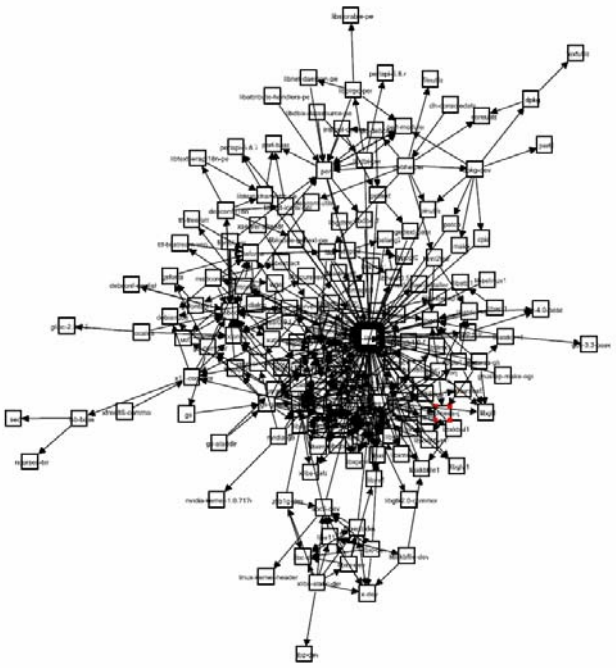

Figure 5: A rendering of $n = 163$ random packages.

Conclusion and Future Research

The customized software used in this research queried the Debian and FreeBSD package repositories, created networks from the results, and calculated graph statistics which gave the following results:

1. Short geodesic path lengths with high clustering (the Small-World effect).
2. A near power-law distribution of edges.
3. A giant component connecting about 88% of the packages.

This research has contributed to the body of knowledge about complex networks in Computing by showing that a previously unstudied group of components have naturally emerging structures similar to other networks. Using statistical analysis on large graphs representing package interactions, we have shown the existence of the power-law distribution and the Small-World effect in networks mined from two large software repositories. We have also identified statistically significant motifs, and shown the existence of a large connected component. Each of these results are in concordance with results from other studies of complex networks. Our research indicates that despite the absence of a central governing body to engineer global structure in the package interaction network, software structures follow patterns that are found in many natural and artificial systems. These results not only extend the number of classified complex networks, but suggest that there are organizational patterns in software that transcend explicit design.

The network can also have an effect on software engineering and robustness, that should be investigated in future research. For example, many OSS projects implement a "changelog" standard, where changes to a project are ad-libbed into a plain text file. For highly-dependent projects, a machine-readable documentation standard could be implemented using existing XML standards and used to communicate possibly important changes. This may prove to be more efficient than the current "Victorian Novel" style of documenting changes in software.

Also, highly-dependent projects should engage in high quality testing for bugs and security problems. As an example, consider libPNG, a package that provides Portable Network Graphics format usage and manipulation to other packages. In 2004, a list of security vulnerabilities in libPNG were made public (U.S. CERT, 2004). These security vulnerabilities could allow a malformed PNG image to crash applications and allow arbitrary code execution. At the time this vulnerability was made public, libPNG had over 650 dependent packages. Because many different web-browsers, file managers, email clients, and graphics editing programs depend on libPNG for rendering and creating PNG files, they were all subject to potential security vulnerabilities inherent in libPNG (depending on which functions in libPNG being used).

The dynamics of software development is an interesting area for study. Many feedback loops exist in the package system, which ultimately provide the basis for "increasing returns" in the development of robust software. A simple model of these feedback loops are shown in Figure 6. As illustrated, changes in one package may inspire changes in another. Creating and using software in a connected system is ultimately self-reinforcing behavior and provides nonlinear feedback. Additionally, because an ecosystem of dependent packages adapt to each other, the effects of evolution on network dynamics should be investigated. Future research should identify how feedback loops work in software, their effect on software robustness and software size, and on network creation.

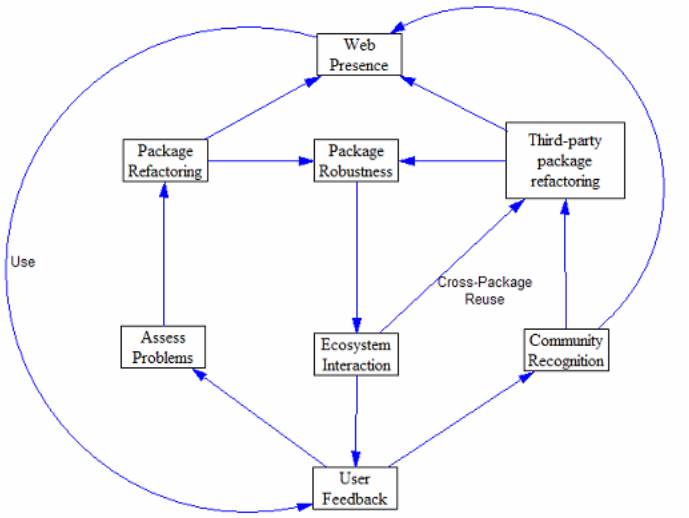

Figure 6: Feedback loops influencing network dynamics and software robustness.

It is known that general power-law networks have a high resistance to random failure, but particular vulnerability to direct attacks on "hub" packages (Albert et al., 2000). Since software networks have no natural homeostasis against failure, the highly-connected nodes should strive to avoid malfunction. Potential avalanches of cascading failures should be identified and isolated before propagated through hub packages.

There are many directions for future research in the study of software networks. Currently, there is no model of network formation that takes software dynamics (reuse, refactoring, and addition of new packages) in to account. Also, the impact of the network structure on software dynamics should be investigated, for example, security exploit propagation through a network of dependent software. Future research should identify other networks in software and move towards formulating a theory of networks and their value to software engineering. Additional dependency networks can be constructed on Windows computers using memory profiling tools, and determining interactions based on shared .DLL (Dynamic Library Link) files and Active-X controls.

References

Albert, R., Jeong, H., & Barabási, A. (1999). Diameter of the World-Wide Web. *Nature, 401*, 130-131.

Albert, R., Jeong, H., & Barabási, A. (2000). Error attack and tolerance of complex networks. *Nature, 406*, 378-381.

Bak, P. (1996). *How Nature Works: The Science of Self-Organized Criticality*. New York: Springer-Verlag.

Clark, D., & Green, C. (1977). An empirical study of list structure in Lisp. *Communications of the ACM, 20,* 78-87.

Faloutsos, M., Faloutsos, P., & Faloutsos, C. (1999). On power-law relationships of the internet topology. *Proc. of ACM SIGCOMM*, 251-262.

Lehmann, S., Lautrup, B., & Jackson, A.D. (2003). Citation networks in high energy physics. *Phys. Rev. E, 68,* 026113.

Lopez-Fernandez, L., Robles, G., & Gonzalez-Barahona, J. (2004). Applying social network analysis to information in CVS repositories. *Proc. of the First International Workshop on Mining Software Repositories*, 101-105.

de Moura, A.P.S., Lai, Y.-C., & Motter, A. (2003). Signatures of small-world and scale-free properties in large computer programs. *Phys. Rev. E, 68,* 017102.

Myers, C.R. (2003). Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. *Phys. Rev. E., 68,* 046116.

National Academies (2005). *Network Science*. Washington: National Academies Press.

Noble, J., Vitek, J., Potter, J. (1998). Flexible alias protection. *Proc. of the 12th European Conference on Object-Oriented Programming*, 158-185.

Potanin, A., Noble, J., Frean, M., & Biddle, R. (2004). Scale-free geometry in object-oriented programs. To appear in *Communications of the ACM*.

Redner, S. (1998). How popular is your paper? An empirical study of the citation distribution. *European Physical Journal B, 4*, 131-134.

Valverde, S., Solé, R. V. (2004). Hierarchical small-worlds in software architecture. Santa Fe Institute working paper SFI/03-07-044.

U.S. CERT: United States Computer Emergency Readiness Team (2004). Technical Cyber Security Alert TA04-217A. Accessed October 25, 2004 from http://www.us-cert.gov/cas/techalerts/TA04-217A.html

Watts, D., & Strogatz, S. (1998). Collective dynamics of 'small-world' networks. *Nature, 393*, 440-442.

Watts, D. (1999). *Small Worlds: The Dynamics of Networks Between Order and Randomness*. Princeton, New Jersey: Princeton University Press.

Wheeldon, R., & Counsell, S. (2003). Power law distributions in class relationships. *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*. Arxiv: cs.SE/0305037.

Zimmerman, T., & Zeller, A. (2001). Visualizing memory graphs. *Lecture Notes in Computer Science, 2269*. 191-204. New York: Springer.

Zipf, G. (1965). *The Psycho-Biology of Language: An Introduction to Dynamic Philology*. Cambridge: MIT Press.