Analysis of the Runtime Linux Operating System as a Complex Weighted Network

Haoqin Wang
School of Automation Science and Electrical Engineering
Beihang University
Beijing, China
wanghaoqin@buaa.edu.cn

Guanping Xiao
School of Automation Science and Electrical Engineering
Beihang University
Beijing, China
gpxiao@buaa.edu.cn

***Abstract*—The structures and behaviors of modern software systems are more and more complicated, thus the modeling of the runtime software systems is difficult. In this paper, we model the runtime Linux operating system (LOS) as a weighted network to investigate the execution process of LOS. The topologies of the weighted LOS network are analyzed and it is found that the weight distribution follows a power-law distribution. For better understanding the execution process of LOS, we explore the manifestations of LOS components. The result shows that the component of process management plays the key role in the execution process of LOS. Moreover, an assessment of the reliability of LOS is proposed by considering the execution status of LOS as a discrete time Markov chain. The reliabilities of 10 LOS versions ranging from versions 3.15 to 4.4 are compared and the result shows that the reliability declines. Our work may shed a light on the testing and monitoring processes of software systems.**

Keywords—Complex network, Linux operating system, Markov chain, Software reliability

I. Introduction

Many natural and artificial systems can be described as networks, where the entities are denoted by nodes and the relationships between entities are denoted by edges, such as the World Wide Web [1], scientific collaborations [2], transportation infrastructures [3][4] and biological systems [5]. In 1959, the presentation of ER random network model [6], which considered that nodes in a network are randomly connected, dominated the research on network for decades. After that, the breakthrough proposals of small-world network model [7] and scale-free network model [8] lead the research on complex networks to a flourishing state in varieties of research domains from many aspects, such as network modeling [9][10][11], epidemic spreading [12][13], cascading failures [14], traffic dynamics [15], evolutionary games [16][17][18], optimization process [19][20] and social dynamics [21][22][23]. One interesting direction worth mentioning is to investigate software systems from the view of complex networks [24][25][26].

Software systems might be one of the most intricate human inventions and take more and more important role in our daily life now. A better understanding of the structures and behaviors of software systems is helpful for designing them. However, it is a great challenge to model software systems due to their high complexity. Thanks to the complex networks theory as a powerful tool to investigate complex systems, researchers have studied software systems in the framework of networks. Valverde et al. [27] built a software system as a complex network and presented the first evidence for the emergence of scaling and the presence of small world behavior in software systems. Myers [28] examined software systems as complex networks and provided a model of software evolution based on refactoring processes.

However, most of the networks are constructed by statically analyzing the source code of software systems [26][27][28][29]. These networks are not applicable in testing and monitoring processes, which are important in the software system's life cycle. Different from the static source code, the dynamic execution process of software systems reflects what is really going on inner the software. Building the dynamic execution processes of software systems are interesting and helpful in the understanding of software behaviors. In this paper, we construct the dynamic execution process of Linux operating system as a weighted network.

Among various software systems, operating systems are the foundation of other software for providing basic execution environment. One of the most widely used operating systems is Linux operating system (LOS), which is released in 1991 by Linus Torvalds and famous for its source open characteristic. In our previous work [30][31], the static source code of LOS was modelled as an unweighted network and we had a number of interesting findings. In this paper, the runtime LOS is built as a weighted network and we analyze its topological properties. Furthermore, a reliability assessment of LOS by constructing the component transition status as a discrete time Markov chain (DTMC) is proposed.

The rest of this paper is organized as follows. Section II proposes the network modeling of the dynamic execution of LOS. Section analyzes the topological properties of the weighted LOS network. Section presents different manifestations of components in LOS. In Section , the reliability of LOS is estimated. Finally, the conclusion of the work is given in Section .

II. Modeling

LOS consists of thousands of functions which collaborate with each other by function calls. The static source code of

LOS shows us the overall design for different requirements of users and hardware. However, not all of the functions are called in a specific execution. Inspired by this, we wish to explore the status of the runtime LOS.

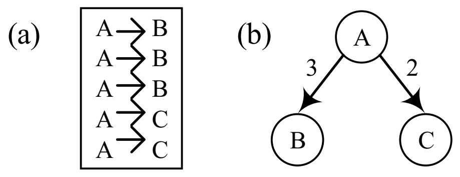

Fig. 1. Illustration of modelling the execution traces as a weighted network. (a) an example of the execution traces; (b) the weighted network represents the execution traces.

In the static analysis of LOS [30][31], we abstracted functions and function calls as nodes and edges in an unweighted network. Obviously, it is not suitable to adapt the unweighted network to represent the runtime LOS when considering the dynamic execution status. Therefore, in this paper, the execution traces of the runtime LOS are modelled as a weighted network, where the calling times of a function call are considered as the weight of an edge. An illustration of the network modelling is exhibited in Fig. 1.

III. Topological properties of weighted LOS network

In this section, Linux version 4.4, the latest version since we started this research, is modelled as a weighted network.

To get the execution information of LOS, an experiment was designed to trace the function calls and record their execution times in real-time when LOS is running. The experiment steps are clearly elaborated in the following.

- We use a benchmark called Unixbench-5.13 [32], which is a widely used Linux testing tool for the testing of kinds of LOS performances, to test LOS. Note that, we execute the benchmark one time and it costs about 30 minutes.
- To trace the function calls in real-time, we utilize a powerful and widely used tool called *Ftrace* to monitor and record the execution information. *Ftrace* is mainly developed by Steven Rostedt and merged into the Linux kernel mainline in kernel version 2.6.27. It is designed to assist developers to find the execution information inside the LOS. Note, the functions traced by *Ftrace* are typical for debugging and testing LOS.
- The trace results from *Ftrace* when executing the Unixbench-5.13 are then recorded and modelled as a weighted LOS network according to the way we illustrate in Section II.

TABLE I. Topological parameters

n	m	$<w>$	w_{min}	w_{max}	$s_{in,min}$	$s_{in,max}$	$s_{out,min}$	$s_{out,max}$
3098	6109	19661.27	1	5660006	0	11648579	0	6095733

Now we study the topological properties of the weighted LOS network. It can be observed from Table I that the weighted LOS network consists of 3098 nodes and 6109 edges. In Table I, $<w>$ is the average weight of edges, it is about twenty thousand and reveals that LOS frequently calls functions when executing. w_{min} is the minimum weight while w_{max} is the maximum weight which is more than 5 million, and its counterpart function call relates to spin lock (a self-protection mechanism for shared resource) in LOS. Moreover, the maximum in-strength $s_{in,max}$, is much larger than the maximum out-strength $s_{out,max}$, which can be interpreted as follows:

1) In general, programmers prefer to write a function with fewer calls for the purpose of making the program readable and reliable.

2) Some basic service functions are called by huge number of functions. This lead to a large value of in-strength.

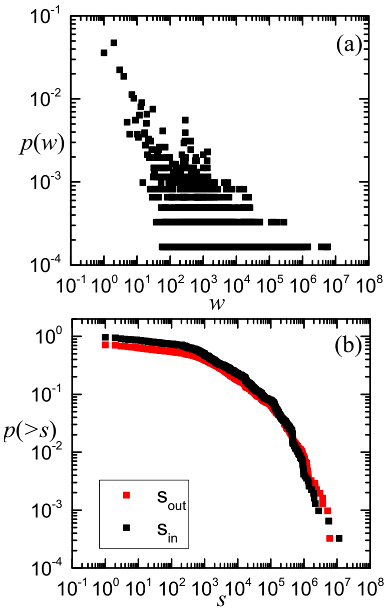

Fig. 2. (a) weight distribution of the weighted LOS network; (b) accumulative strength (both in-strength and out-strength) distribution of the weighted LOS network.

Fig. 2 (a) illustrates the weight distribution $P(w)$ of the weighted LOS network, which follows a power law distribution. It is clearly observed that few nodes have extremely large weights while most nodes have small weights. Inspired by this, we make a further investigation about the weights, which shows that the sum of the top 400 weights (6.5% of the whole edges) takes 82.9% percentage of the total sum of weights, indicating the high reuse rate of code in the execution of LOS. Moreover, the accumulative strength (both in-strength and out-strength) distribution of the weighted LOS network is shown in Fig. 2 (b). It is found that they both have power-law tails, indicating the high level of heterogeneity in the weighted LOS network.

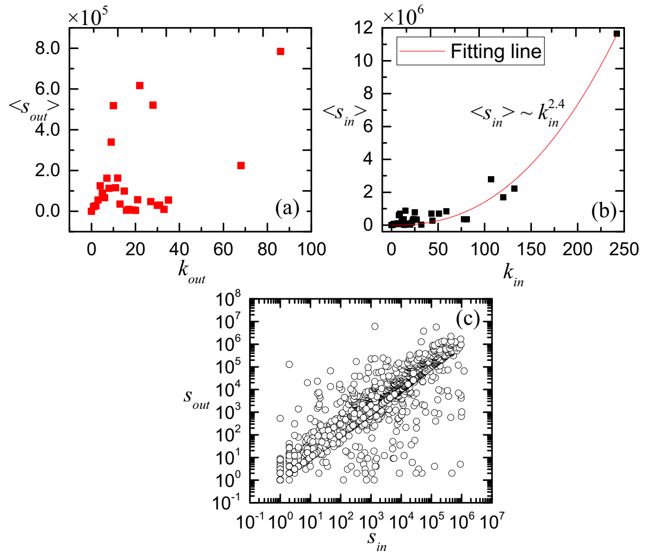

Fig. 3. (a) The relationship between out-strength and out-degree. (b) The correlation between in-strength and in-degree. (c) The relationship between s_{out} and s_{in}.

Fig 3 (a) and (b) illustrate the relationship between strength and degree, which show that the relationship between out-strength and out-degree is disordered while the relationship between in-strength and in-degree follows power-law distribution. Fig. 3 (c) displays a positive power-law relation between in-strength and out-strength. This is reasonable that with the increasing of the number of calling times of a function, the number of its called times also increases.

IV. Manifestations of LOS Components

LOS is mainly composed of 8 components (i.e. *arch, block, drivers, fs, kernel, mm, net, security*), which collaborate with each other to realize the functionality of LOS [33]. In the following, we make an analysis of execution counts of the 8 components to reveal their manifestations in the runtime LOS.

The strength of a node indicates the number of times it is called by LOS. Similarly, we define the strength of component I as $M(I)$, which is the sum of node strength in a specific component. Also, the strength of a component includes in-strength and out-strength, i.e., $M_{in}(I)=\sum_{i\subset I} s_{in}(i)$, $M_{out}(I)=\sum_{i\subset I} s_{out}(i)$, where $M_{in}(I)$ is the in-strength of component I, $s_{in}(i)$ is the in-strength of node i, $M_{out}(I)$ is the out-strength of component I, $s_{out}(i)$ is the out-strength of node i. The values of in-strength and out-strength reveal the reuse status of a component, indicating the importance of components to a certain extent.

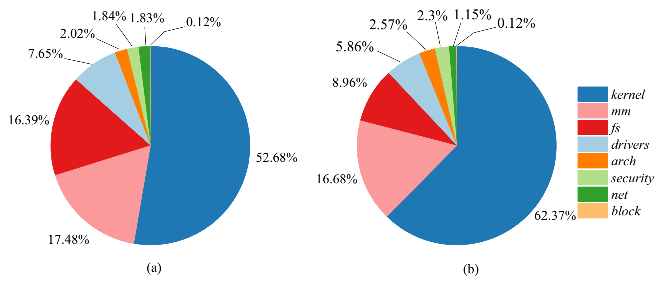

Fig. 4. (a) is the out-strength of the 8 components; (b) is the in-strength of the 8 components.

In summary, the total number of times that the 8 components are called occupy 98.98% percentage of the whole times of function calls in the execution of LOS, which means that the execution of LOS are closely related to the 8 components. Fig. 4 illustrates the percentage of in-strength and out-strength of the 8 components. It is found that *kernel* component occupies more than 50% percentage both in the in-strength and out-strength, indicating that the behavior of the runtime LOS mainly manifests in the operation of *kernel*. This illustrates that *kernel* takes the most important role in the execution of LOS to some extent. Besides, the top 3 components with high percentages of both in-strength and out-strength are *kernel*, *mm* and *fs*, which reveals that in the runtime LOS, the most 3 frequently executed events are process management, memory management and disk operation.

V. Reliability Assessment of LOS

The reliability of a software system is one of the most important properties of software quality. In this part, we propose an assessment of the reliability of LOS according to the real execution status by establishing a discrete time Markov chain (DTMC) [34].

We describe the modular structure as an absorbing DTMC, characterized by its one-step transition probability matrix $P=[p_{ij}]$. Note that, all the elements in a row of P add up to 1 and each of the p_{ij} values lies in the range [0, 1]. The one-step transition probability matrix with n states and m absorbing states can be partitioned as:

$$P=\begin{pmatrix} Q & C \\ 0 & I \end{pmatrix}$$

where Q is an $(n-m)\times(n-m)$ substochastic matrix, I is an $m\times m$ identity matrix, 0 is an $m\times(n-m)$ zero matrix and C is an $(n-m)\times m$ matrix. Let P^k be the k-step transition probability matrix, then the (i,j) entry of Q^k gives the probability of arriving in state *j* from state *i* after *k* steps and the so called fundamental matrix *M* is obtained as:

$$M=(I-Q)^{-1}=I+Q+Q^2+\cdots+Q^k=\sum_{k=0}^{\infty}Q^k$$

Let $X_{i,j}$ be the number of visits from state *i* to state *j* before absorption, it can be shown that the expected number of visits from *i* to *j* equals the value of (i,j)th element of *M*, i.e. $E(X_{i,j})=m_{i,j}$. Thus, we can obtain the expected number of visits from the initial state to state *j*, denoted as V_j. Therefore, the system reliability can be obtained by the following equation $R=\prod_{i=1}^{n}R_i^{V_i}$, where R_i is the reliability of component *i*. If we consider a constant failure rate λ_i of component *i*, the reliability of each component can be obtained by $R_i=e^{-\lambda_i\tau_i}$, where τ_i is the expected time that each visit to component *i* caused. In the following, we will give a clearly description of the reliability estimation of LOS.

Firstly, we use the number of function calls to calculate the one-step transition matrix as illustrated in Fig. 5. In an execution, component *A* calls itself 90 times and *B* 10 times and the corresponding transition probabilities are 0.9 from *A* to *A* and 0.1 from *A* to *B*. For LOS, we regard the execution in a component as a state and the end of the execution as the absorbing state, thus a DTMC with 9 states is established. The one-step transition probability is calculated according to the total number of weights among and in components. It is observed that *kernel* is the initial state and has an edge to the final state in our experiment. Once we get the one-step transition matrix, the fundamental matrix and the expected execution times of each component can be obtained through the way we demonstrated before.

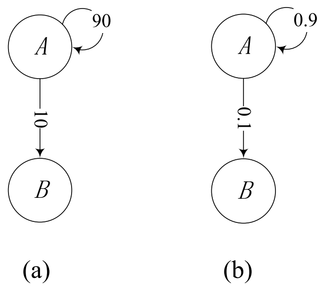

Fig. 5. An illustration of calculating transition probabilities. (a) transition times between Component *A* and *B*; (b) transition probabilities.

Secondly, we need to know the constant failure rate and expected visiting time in each component. *Ftrace* gives us an entry to obtain the execution time of each function, thus we can get the expected visiting time of each component by the average of the time that each function in the component cost. Then the problem is how we get the failure rate of each component. It is obvious that the failure of a system is closely related to the fault of the system, which means that the system with more faults is more likely to fail. Based on this, it is reasonable to give an estimate of the failure rate according to the initial fault in the system, which has a verification in the Goel-Okumoto model [35]. Additionally, we assume that the failure rate of a component can be calculated as $\lambda_i=a_i\times k$, where a_i is the initial fault in the component, *k* is a rate constant. There are many methods can be utilized to estimate the initial number of faults, such as [36][37], here we obtain this estimation using the fault density approach [36]. Thus, the expected number of faults in component *i* is $a_i=\frac{FD\times l_i}{10000}$, where l_i is the number of lines in component *i* and *FD* is the fault density as the number of faults per 10000 lines of code. Since the fault density of the LOS is not known, we vary the fault density from 1 to 8. Considering that the LOS is hard to fail, thus the failure rate is small and we set the rate constant as $k=10^{-8}$.

Finally, the reliability of LOS as a function of the fault density is shown in Fig. 6 (a). It is found that the reliability of the LOS drops with the increasing of the fault density. Besides, with the increasing of *FD*, the decreasing percentage of the reliability becomes slow. To investigate the reliability among different LOS versions, the reliabilities of 10 LOS versions (3.15, 3.16, 3.17, 3.18, 3.19, 4.0, 4.1, 4.2, 4.3, 4.4) are compared with the same *FD* (*FD* = 1), as illustrated in Fig. 6 (b). It is found that the reliability of LOS decreases slightly with the increasing of the sequence numbers. This could be interpreted that with the evolution of LOS, it becomes more and more complicated. Consequently, the reliability of LOS decrease.

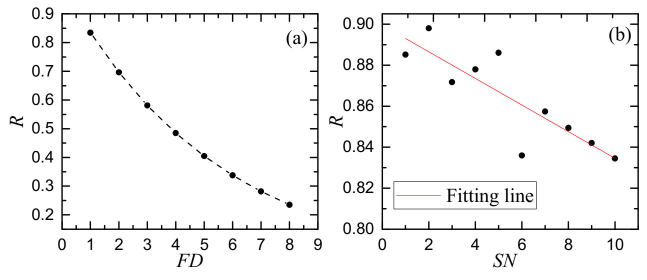

Fig. 6. (a) the reliability as a function of *FD*; (b) the reliability of LOS as a function of the sequence numbers (*SN*). Note, *SN* is the sequence number which is assigned to each version according to their release date. i.e., the sequence number of version 3.15 is 1 and that for version 4.4 is 10.

VI. Conclusion

In this paper, we build the runtime LOS as a weighted network to explore the execution status of LOS. The topological properties of the weighted LOS network are analyzed and a notable scale-free phenomenon of the weight distribution is found. Besides, the manifestations of different components in the runtime LOS are investigated, and the result shows that *kernel*, *mm* and *fs* are the key components in the execution process of LOS. This indicates that LOS mainly deals with three main events, i.e., process management, memory management and disk operation, when it is running. Moreover, the execution status of LOS is modeled as a DTMC and the reliabilities of 10 LOS versions ranging from 3.15 to 4.4 are estimated. It is found that with the development of LOS, its reliability declines. Our work may shed a light on the testing and monitoring processes of software systems and give a way to estimate the reliability of large software systems.

Acknowledgment

The authors would like to thank Dr. Wen-Bo Du from Beihang University for kindly providing valuable advices to this manuscript.

References

[1] R. Albert, H. Jeong, A.L. Barabási, Internet: Diameter of the world-wide web, Nature, vol. 401, pp. 130-131, September 1999.

[2] M.E.J. Newman, Scientific collaboration networks. I. Network construction and fundamental results, Phys. Rev. E, vol. 64, pp. 016131, July 2001.

[3] J. Zhang, X.B. Cao, W.B. Du, K. Q. Cai, Evolution of chinese airport network, Physica A: Statistical Mechanics and its Applications, vol. 389, pp. 3922-3931, September 2010.

[4] W.B. Du, X.L. Zhou, O. Lordan, Z. Wang, C. Zhao, Y.B. Zhu, Analysis of the Chinese Airline Network as multi-layer networks, Transportation Research Part E: Logistics and Transportation Review, vol. 89, pp. 108-116, May 2016.

[5] H. Jeong, B. Tembor, R. Albert, Z.N. Oltvai and A.L. Barabási, The large-scale organization of metabolic networks. Nature, vol. 407, pp. 651-654, October 2000.

[6] P. Erdős, A. Rényi, On random graphs I, Publ. Math. Debrecen, vol. 6, pp. 290-297, 1959.

[7] D.J. Watts, S.H. Strogatz, Collective dynamics of small-world networks, Nature, vol. 393, pp. 440-442, June 1998.

[8] A.L. Barabási, R. Albert, Emergence of scaling in random networks, Science, vol. 286, pp. 509-512, October 1999.

[9] A. Barrat, M. Barthélemy, A. Vespignani, Weighted evolving networks: coupling topology and weight dynamics, Phys. Rev. Lett., vol. 92, pp. 228701, June 2004.

[10] T. Zhou, G. Yan, B.H. Wang, Maximal planar networks with large clustering coefficient and power-law degree distribution, Phys. Rev. E, vol. 71, pp. 046141, April 2005.

[11] W.X. Wang, B.H. Wang, B. Hu, G. Yan, Q. Ou, General dynamics of topology and traffic on weighted technological networks, Phys. Rev. Lett., vol. 94, pp. 188702, May 2005.

[12] R. Pastor-Satorras, A. Vespignani, Epidemic spreading in scale-free networks, Phys. Rev. Lett., vol. 86, pp. 3200, April 2001.

[13] H.X. Yang, W.X. Wang, Y.C. Lai, Y.B. Xie, B.H. Wang, Control of epidemic spreading on complex networks by local traffic dynamics, Phys. Rev. E, vol. 84, pp. 045101, October 2011.

[14] W. Wang, Y.C. Lai, Abnormal cascading on complex networks, Phys. Rev. E, vol. 80, pp. 036109, September 2009.

[15] W.B. Du, Z.X. Wu, K.Q. Cai, Effective usage of shortest paths promotes transportation efficiency on scale-free networks, Physica A: Statistical Mechanics and its Applications, vol. 392, pp. 3505–3512, September 2013.

[16] W.B. Du, X.B. Cao, M.B. Hu, W.X. Wang, Asymmetric cost in snowdrift game on scale-free networks, Europhys. Lett., vol. 87, pp. 60004, October 2009.

[17] Z. Wang, A. Szolnoki, M. Perc, Optimal interdependence between networks for the evolution of cooperation, Sci. Rep., vol. 3, pp. 2470, August 2013.

[18] A. Szolnoki, M. Perc, Reentrant phase transitions and defensive alliances in social dilemmas with informed strategies, Europhys. Lett., vol. 110, pp. 38003, May 2015.

[19] C. Liu, W.B. Du, W.X. Wang, Particle swarm optimization with scale-free interactions, PLoS One, vol. 9, pp. e97822, May 2014.

[20] Y. Gao, W.B. Du, G. Yan, Selectively-informed particle swarm optimization, Sci. Rep., vol. 5, pp. 9295, March 2015.

[21] C. Castellano, S. Fortunato, V. Loreto, Statistical physics of social dynamics, Rev. Modern Phys., vol. 81, pp. 591, May 2009.

[22] Z. Wang, Y. Liu, L. Wang, Y. Zhang, Freezing period strongly impacts the emergence of a global consensus in the voter model, Sci. Rep., vol. 4 pp. 3597, January 2014.

[23] W.B. Du, Y. Gao, C. Liu, Z. Zheng, Z. Wang, Adequate is better: particle swarm optimization with limited-information, Appl. Math. Comput., vol. 268, pp. 832-838, October 2015.

[24] S. Jenkins, S.R. Kirk, Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution, Inform. Sci., vol. 177, pp. 2587-2601, June 2007.

[25] G. Concas, M. Marchesi, S. Pinna, N. Serra, Power-laws in a large object-oriented software system, IEEE Trans. Softw. Eng., vol. 33, pp. 687-708, October 2007.

[26] P. Louridas, D. Spinellis, V. Vlachos, Power laws in software, ACM Trans. Softw. Eng. Methodol., vol. 18, pp.2, September 2008.

[27] S. Valverde, R.F. Cancho, R.V. Sole, Scale-free networks from optimal design, Europhys. Lett., vol. 60, pp.512, September 2002.

[28] C.R. Myers, Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs, Phys. Rev. E, vol. 68, pp. 046116, October 2003.

[29] L. Subelj and M. Bajec. Community structure of complex software systems: Analysis and applications, Physica A: Statistical Mechanics and its Applications, vol. 390, pp. 2968–2975, August 2011.

[30] Y.C. Gao, Z. Zheng and F.Y. Qin. Analysis of Linux kernel as a complex network, Chaos, Solitons & Fractals, vol. 69, pp. 246-252, December 2014.

[31] H.Q. Wang, Z. Chen, G.P. Xiao and Z. Zheng. Network of networks in linux operating system, Physica A Statistical Mechanics & Its Applications, vol. 447, pp. 520-526, April 2016.

[32] https://code.google.com/p/byte-unixbench

[33] Robert Love, Linux kernel development, Pearson Education, 2010.

[34] R.C. Cheung, A user-oriented software reliability model, IEEE Trans. Software Eng, vol. 6, pp 118–125, March 1980.

[35] A.L. Goel, K. Okumoto, Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures, IEEE transactions on Reliability, vol. 3,pp. 206-211, 1979.

[36] M. Lipow, Number of Faults per Line of Code, IEEE Transactions on Software Engineering, vol. SE-8, pp. 437 – 439, July 1982.

[37] Swapna S. Gokhale Kishor S. Trivedi. Reliability Prediction and Sensitivity Analysis Based on Software Architecture, Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on, 2002, pp. 64-75.