Source Code Survival with the Kaplan Meier Estimator

Giuseppe Scanniello
University of Basilicata, Potenza, Italy
Email: giuseppe.scanniello@unibas.it

***Abstract*—The presence of dead code may affect the comprehensibility, the readability, and the size of source code so increasing the effort and the cost for maintenance. The elimination of dead code needs a huge cost and effort for recognizing and eliminating code that is not effectively used. The goal of this work consists in defining an approach based on the Kaplan Meier estimator to analyze dead code. The validity of the approach has been preliminarily assessed on a case study constituted of fifty-eight versions of five open source software systems implemented in Java. The results suggested that two out of the five systems where implemented avoiding as much as possible the introduction of dead code.**

Keywords - ***Dead code; Kaplan Meier estimator; empirical study; software maintenance***

I. Introduction

Survival analysis deals with the death or the survival of biological organisms and the failure of mechanical systems [1]. In case of biological organisms, death is unambiguous. This is not the same for other kinds of biological events or physiological events (e.g., heart attack). Also for mechanical systems, failures may not be well-defined and ambiguous. For example, a failure may be partial or not well localized in time.

The survival can be modeled by employing a survival function. This kind of function is a property of any random variable that maps a set of events (e.g., death or failure) onto time, thus capturing the probability that a system or a biological organism will survive for a specified time. A survival function assumes different meaning according to the field where the function is applied. For mechanical systems, for instance, a survivor function can be used to measure the time until a failure to a given part happens.

Similar to traditional engineering disciplines, a survival function in software engineering can be used to measure the failure of a software or its aging[1] [2], [3]. Differently from the other disciplines, in the software engineering field it could be interesting to analyze the dead code (i.e., unnecessary and inoperative source code that can be removed). Dead code can be added or removed across different subsequent versions of an evolving software system. Unused code can remain in case refactoring operations are not performed to remove it.

The presence of dead code may affect the comprehensibility, the readability, and the size of source code increasing then the effort and the cost for maintenance and testing [4], [5], [6]. The elimination of dead code needs a huge effort that is mostly required to recover and to eliminate unused code [7], [8]. Therefore, it seems reasonable to eliminate dead code from a software system only in case its elimination is sufficiently paid back by an improved maintainability and testability of the cleaned source code.

In this paper, an approach is proposed to analyze the death of source code through different versions of evolving software systems. The approach is based on the Kaplan Meier estimator [9], a well known and widely used survival function. The validity of the approach has been assessed on a case study constituted of fifty-eight versions of five open source software systems implemented in Java.

The remainder of the paper is organized as follows. In Section II the approach is highlighted. The design of the case study and the obtained results are presented in Section III and Section IV, respectively. Related work is discussed in Section V, final remarks and future work conclude.

[1]It indicates a progressive performance degradation or a sudden hang/crash of a software system due to exhaustion of operating system resources, fragmentation, and accumulation of errors

II. The Approach

This section first presents the Kaplan Meier estimator [9] and then discusses how it has been modified to better handle the specific domain.

A. General Formulation

Let $S(t)$ be the probability that an item/subject from a given population will have a lifetime exceeding t. For a sample from this population, whose size is N, if the observed times until the death/failure of items are $t_1 \leq t_2 \leq t_3 \leq ... \leq t_N$, the following survival function can be defined:

$$S(t_i) = 1 - \frac{e_i}{n_i}$$

n_i (i.e., the the number of items "at risk") is the number of survivors less the number of censored cases (i.e., the number of items removed from the sample before the final outcome is observed), while e_i is the number of events (i.e., the death/failure of an item). This function computes the percentage of survival items for a given time.

To take into account the fact that the survival of items at the time t_i is conditioned by their survival from t_1, to $t_i - 1$, the following cumulative survival function can be computed:

$$C(t_i) = S(t_i) * C(t_{i-1})$$

Lower the value of the cumulative survival function, higher the probability of death/failure at the time t_i is.

B. The Estimator

To employ Kaplan Meier in the specific domain, the following mapping has been proposed:

- **Sample**. The set of methods.
- **Lifetime**. The releases of a software.
- **Death**. The number of source code blocks that become unused in a given version. For unused source code blocks, different granularity levels could be used. The estimator is applied at method level in this work.
- **Censored**. The number of methods removed from the sample when passing from a version of a software to the subsequent one.

The survival theory assumes that death or failure happens just once for each subject in the sample. This holds also for the proposal presented in the paper. In particular, a method that is unnecessary and inoperative in a version can come back from the dead in the subsequent version only in case it is added as a new method. This was possible since the estimator works on samples (i.e., the number of methods), whose size can variate over the subsequent versions. Accordingly, n_i is the number of survivors increased of the number of new added methods less the number of censored ones. The study of recurring events (e.g., a dead method in a version can come back from the dead) might be relevant and it is subject of future work.

III. Case Study

In this section, the design of the investigation is presented following the guideline proposed by Yin in [10]. For replication purposes, an experimental package is available on the web[2]. A prototype of a supporting system implementing the approach is also available.

A. Definition and Context

The perspective of the study is from the point of view of the *researcher*, assessing whether the estimator can be used to study unused methods in evolving software systems, and of the *project manager*, getting decision support to the elimination of unused methods.

The case study was conducted on five open source software systems implemented in Java:

1) **JEdit** is a programmer's text editor with an extensible plug-in architectures.
2) **JFreeChart** is a tool supporting the visualization of bar charts, pie charts, scatter plots, and more.
3) **JHotDraw** is a GUI framework for technical and structured Graphics.
4) **JUnit** is a programmer-oriented framework for testing Java code.
5) **PMD** is a source code analyzer for Java software. It finds unused variables, empty catch blocks, and so forth.

[2]www.scienzemfn.unisa.it/scanniello/SurvivalAnalysis/

TABLE I
DESCRIPTIVE STATISTICS

System	Version	Classes	Constr.	Meth.	KLOCs
JEdit	15	251-341	430-826	2627-5986	47.5-105.1
JFreeChart	6	86-131	156-218	655-997	6.9-13
JUnit	20	24-160	19-64	202-1383	1.1-8.7
JHotDraw	8	309-472	490-719	3631-5398	32.2-86
PMD	9	641-727	189-194	5360-6029	48-54.5

TABLE II
JEDIT RESULTS

Ver.	At Risk	Event	Cens.	$S(t)$	$C(t)$
3.0	1648	934	0	0.434	0.434
3.0.1	1650	937	0	0.433	0.188
3.0.2	1651	938	0	0.432	0.081
3.1	1701	989	25	0.419	0.034
3.2	1978	1081	30	0.454	0.015
3.2.1	1950	1082	0	0.446	0.007
3.2.2	1952	1083	1	0.446	0.003
4.0	2360	1272	11	0.462	0.001
4.0.2	2349	1271	0	0.459	0.000
4.0.3	2349	1271	0	0.459	0.000
4.1	2658	172	15	0.936	0.000
4.2	4169	165	153	0.961	0.000
4.3	6843	168	327	0.976	0.000
4.3.1	6518	0	0	1.000	0.000
4.3.2	6521	170	2	0.974	0.000

Table I shows some descriptive statistics about these systems. The name of the system is shown in the first column, while the second column reports the total number of versions considered in the study. The minimum and maximum numbers of classes, constructors, methods, and line of code (KLOCs) for each system are reported in last four columns, respectively.

B. Planning

The software systems used in the case study have been selected according to their availability on the web and their non-trivial size. The attention has been focused on systems that have been largely investigated in the software maintenance field (e.g., [11], [12]). Furthermore, the investigation has been performed randomly selecting for each system a subset of consecutive distribution versions. An investigation on all the available distributions should not significantly affect the results of the study.

A software prototype of a supporting system has been also implemented to automate the approach and to conduct the case study. The prototype is implemented in Java and integrates JTombstone (available at jtombstone.sourceforge.net) to detect dead code.

IV. Results and Discussion

In the following, the results and the threats that may affect their validity are presented and discussed.

A. Achieved Results

The results obtained on the versions of each system are summarized in Tables II, III, IV, V, and VI. For each table, the first column shows the version of the distribution, while the number of methods at risk are reported in the second.

TABLE III
JFREECHART RESULTS

Ver.	At Risk	Event	Cens.	$S(t)$	$C(t)$
0.5.6	546	187	0	0.658	0.658
0.6.0	744	446	47	0.401	0.264
0.7.0	817	430	4	0.474	0.125
0.7.1	985	507	10	0.486	0.061
0.7.2	1019	510	3	0.500	0.030
0.7.3	1023	505	2	0.507	0.015

TABLE IV
RESULTS ON JUNIT

Ver.	At Risk	Event	Cens.	$S(t)$	$C(t)$
2.0	205	54	0	0.737	0.737
2.1	231	55	0	0.762	0.562
3.0	279	68	2	0.757	0.425
3.2	209	0	0	1.000	0.425
3.4	255	9	3	0.965	0.410
3.5	256	0	1	1.000	0.410
3.6	259	0	0	1.000	0.410
3.7	259	0	0	1.000	0.410
3.8	403	6	2	0.986	0.404
3.8.1	415	0	24	1.000	0.404
3.8.2	416	0	12	1.000	0.404
4.0	642	0	1	1.000	0.404
4.1	645	0	0	1.000	0.404
4.2	660	0	0	1.000	0.404
4.3.1	681	0	12	1.000	0.404
4.4	762	60	3	0.922	0.372
4.5	1072	60	0	0.945	0.352
4.6	1108	60	0	0.946	0.333
4.7	1149	60	1	0.948	0.316
4.8	1161	60	0	0.949	0.300

TABLE V
JHOTDRAW RESULTS

Ver.	At Risk	Event	Cens.	$S(t)$	$C(t)$
7.0.7	2451	0	0	1.000	1.000
7.0.8	3157	0	52	1.000	1.000
7.0.9	4213	0	171	1.000	1.000
7.1	4733	0	124	1.000	1.000
7.2	5834	0	42	1.000	1.000
7.3	6024	0	38	1.000	1.000
7.3.1	6000	0	2	1.000	1.000
7.4.1	6940	0	76	1.000	1.000

TABLE VI
PMD RESULTS

Ver.	At Risk	Event	Cens.	$S(t)$	$C(t)$
1	3703	2502	0	0.325	0.325
2	4061	2611	11	0.358	0.116
3	4121	2657	40	0.356	0.041
4	4366	2809	14	0.357	0.015
5	4356	2813	0	0.355	0.005
6	4358	2815	0	0.355	0.002
7	4361	2812	3	0.356	0.001
8	4359	2812	0	0.355	0.000
9	4364	2813	0	0.356	0.000

The number of events and censored methods for each version is shown in the fourth column. The last two columns report the values of the survival and cumulative survival functions, respectively. For the first version of each system, the values of the survival and the cumulative functions obviously coincide.

The results indicate that JHotDraw (see Table V) is the only software system that did not have any concerns with the presence of dead code. The values of the survival and cumulative survival functions coincide for all the distributions and are always equal to 1. This result may indicate that JHotDraw has been developed paying attention to avoid introducing unused source code (methods in our case). Caution is needed and a special conceived future analysis will be performed to verify this result. Similarly, for each version of JUnit (see Table IV) we can observe high values of the survival function (i.e., mostly close to 1). Furthermore, the values of the cumulative function slightly decrease through the analyzed versions. Differently, the values of the cumulative function computed on the selected distribution versions quickly decrease for JEdit, JFreeChart, and PMD.

The results of this preliminary study suggest that JHotDraw does not require refactoring operations since unused source code is not present. The absence of dead code could be due to the fact that developers avoid as much as possible introducing unused methods or removed them before the deployment of each release. Similarly, the developers of JUnit paid a sufficient attention in introducing new dead code when coding. Conversely, developers did not pay attention to the introduction of unused code in the other three systems. With regard to JEdit, it is possible to observe that the values of the survival functions are close to 1 between the the distribution 4.1 and the distribution 4.3.2. On these distributions the the values of the cumulative function are close to 0. This finding could indicate that developers introduced a lower number of unused methods starting from the version 4.1. The low values for the cumulative function are due to the unused methods introduced in the early versions. Future work could be devoted to analyze the versions 4.0.3 and 4.1 to understand whether unused methods have been removed in the version 4.0.3 .

B. Threats to Validity

The reliability of JTombstone (i.e., the tool used in the identification of unused methods) may threaten the generalizability of the results. In fact, the correctness and the completeness of the set of the identified methods that are actually dead code may strongly affect the results. To reduce as much as possible this threat the case study should be replicated using different tools for dead code detection. Besides, the best solution should be the manual identification of dead method. This is practically impossible on large software systems.

Also, the use of open source software systems may threaten the validity of the results. In fact, this kind of systems is mainly developed according to a mass collaboration. This contrast with the more centralized models of development typically used in the greater part of the software companies. Therefore, it will be worth applying the approach on commercial software to confirm or contradict the achieved results.

V. RELATED WORK

This section discusses approaches based on survival analysis or that use this kind of analysis to estimate relevant aspects of software projects. For example, Sentas *et al.* [2] propose statistical tools for studying and modeling the distribution of the duration of software projects. One of the most important advantages of the approach is that data from completed and uncompleted projects can be used for the estimation. Similarly, Sentas *et al.* [13] define a method to predict the duration of a software project.

The survivability of software projects is investigated in [3]. The main feature of the method is the construction of probabilistic models based on data of complete and ongoing projects. As for this paper, the authors conduct a case study on open source projects to validate their proposal.

In [14] a method for empirically evaluating the availability of deployed software systems is presented. The method is based on survival analysis and uses information from operational customer support and inventory systems. Differently, Wilson *et al.* [15] propose a software reliability model based on the order-statistic paradigm. The objective of this model is to estimate the unknown number of bugs in the software and to predict failures.

In the software maintenance field the results of an interesting empirical study are presented in [16]. The main objective of the study is to investigate whether programmers unfamiliar with a system to maintain start by actively working on code or spend time to passively explore the system before performing changes. Since some of the participants to the study do not complete the maintenance tasks, the authors use Kaplan Meier and the Cox Proportional Hazard model to analyze the data.

Differently from the work presented here, the approaches described above do not consider dead code and many of them do not use the Kaplan Meier estimator to study this phenomenon in evolving software systems.

VI. Conclusion and Future Work

Dead code is a very common phenomenon. It increases the file size by hundreds of kilobytes, so excessively using memory and possibly augmenting the execution time. The presence of dead code also means more code to read, thus leading to higher costs, especially during testing and maintenance activities [4], [5], [6]. Regarding the testing activities, dead code may have also the effect of carrying around untested code that contains bugs. It could happen that dead code may inadvertently become operative later and then causing possible failures. Differently, maintainers must uselessly spend time to understand dead code while performing maintenance tasks.

In this paper, an approach based on the Kaplan Meier estimator is proposed to study how dead code affects evolving software systems. The approach has been implemented in a prototype of a supporting software system. Then, the approach and the prototype have been validated on several versions of five open source software systems, all implemented in Java. The most remarkable result is that on two out of five systems the indicator showed that developers avoided as much as possible to introduce dead code. Conclusive results cannot be drawn because of the preliminary nature of the empirical investigation.

A future direction for this work could consist in investigating the possible relation between the proposed survival function and measures for estimating software quality factors (e.g., maintainability, efficiency, or understandability). This future investigation will enable to comprehend whether the survival analysis could be also applied for estimating software quality factors. It would be also worth defining a strategy to suggest when and whether refactoring operations must be performed.

It will be also worth adapting the approach to handle source code at different granularity level (e.g., at statement block level). Furthermore, the tool prototypes could be extended to: *(i)* study software implemented using different programming languages (e.g., C and C++) and *(ii)* access information in software configuration management tools (e.g., CVS and SVN). Finally, future work will be devoted to compare the proposed estimator with other survival functions.

VII. Acknowledgment

I would like to thank the Marilena Nolé, who developed the software prototype implementing the approach presented in the paper.

References

[1] R. C. Elandt-Johnson and N. L. Johnson, *Survival models and data analysis / Regina C. Elandt-Johnson, Norman L. Johnson*. John Wiley & Sons, New York, 1980.

[2] P. Sentas, L. Angelis, and I. Stamelos, “A statistical framework for analyzing the duration of software projects,” *Emp. Softw. Eng.*, vol. 13, no. 2, pp. 147–184, 2008.

[3] I. Samoladas, L. Angelis, and I. Stamelos, “Survival analysis on the duration of open source projects,” *Inf. and Softw. Tech.*, vol. 52, pp. 902–922, September 2010.

[4] M. Kiewkanya and P. Muenchaisri, “Measuring maintainability in early phase using aesthetic metrics,” in *WSEAS*, 2005, pp. 25:1–25:6.

[5] Z. Huang and L. Carter, “Automated solutions: Improving the efficiency of software testing,” 2003.

[6] Y.-F. Chen, E. R. Gansner, and E. Koutsofios, “A c++ data model supporting reachability analysis and dead code detection,” *IEEE Trans. on Softw. Eng.*, vol. 24, pp. 682–694, 1998.

[7] B. Andreopoulos, “Satisficing the conflicting software qualities of maintainability and performance at the source code level,” in *WER*, 2004, pp. 176–188.

[8] C. Jones, “The economics of software maintenance in the twenty-first century,” 2006.

[9] E. L. Kaplan and P. Meier, “Nonparametric estimation from incomplete observations,” *J. of the Amer. Stat. Ass.*, vol. 53, no. 282, pp. 457–481, 1958.

[10] R. K. Yin, *Case study research: Design and methods*. Thousand Oaks, CA: Sage, 1994.

[11] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, “Design pattern recovery through visual language parsing and source code analysis,” *J. Syst. Softw.*, vol. 82, pp. 1177–1193, July 2009.

[12] G. Scanniello, A. D’Amico, C. D’Amico, and T. D’Amico, “Using the kleinberg algorithm and vector space model for software system clustering,” in *ICPC*, 2010, pp. 180–189.

[13] P. Sentas and L. Angelis, “Survival analysis for the duration of software projects,” *In Software Metrics*, vol. 0, p. 5, 2005.

[14] A. Mockus, “Empirical estimates of software availability of deployed systems,” in *ESEM*. New York, NY, USA: ACM, 2006, pp. 222–231.

[15] S. P. Wilson and F. J. Samaniego, “Nonparametric analysis of the order-statistic model in software reliability,” *IEEE Trans. Softw. Eng.*, vol. 33, pp. 198–208, March 2007.

[16] A. Hutton and R. Welland, “An experiment measuring the effects of maintenance tasks on program knowledge,” in *EASE*. BCS, 2006.