Is Software Aging related to Software Metrics?

Domenico Cotroneo, Roberto Natella, Roberto Pietrantuono
Dipartimento di Informatica e Sistemistica,
Università degli Studi di Napoli Federico II,
Via Claudio 21, 80125, Naples, Italy
{*cotroneo, roberto.natella, roberto.pietrantuono*}*@unina.it*

***Abstract*—This work presents an empirical analysis aiming at investigating what kind of relationship exists between software aging and several static features of the software. While past studies on software aging focused on predicting the aging effects by monitoring and analytically modeling resource consumption at runtime, this study intends to explore if the *static features* of the software, as derived by its source code, presents potential relationships with software aging. We adopt a set of common software metrics concerning program structure, such as size and cyclomatic complexity, along with some features specifically developed for this study; metrics were then computed from ten complex software applications affected by aging. A statistical analysis to infer their relationship with software aging was carried out. Results encourage further investigations in this direction, since they show that software aging effects are related to the static features of software.**

***Keywords*-Software Aging; Software Complexity Metrics; Aging-Related Bugs**

I. Introduction

Long-running systems are known to experience a continued and growing degradation of software's internal state and a progressive performance loss due to the phenomenon of software aging [1]. This phenomenon has been widely observed in the past in several operational systems [2], [3]. In the literature, software aging has been typically addressed by estimating the time to failure due to aging (usually referred to as *Time-To-Exhaustion*, TTE), in order to take appropriate proactive actions. This can be achieved by monitoring resource consumption at runtime [2], [4], and by using analytical models [5], [6].

In this work, we intend to analyze the phenomenon by a different perspective. Our goal is to investigate if software aging has a relationship with software metrics. Software metrics are a measure of the complexity of a program with respect to the properties of the program's text [7]; as such, they are computed statically and do not require to execute the program. We intend to carry out an analysis based on these static features to figure out if there is some characteristic of the software that can induce, or simply be indicative of, *aging-related bugs* in a program. To this aim, we set up an empirical study based on a set of ten software applications whose software aging issues are known.

It is reasonable that it is more likely for a developer to omit release operations (e.g., to release locks or memory) in complex and large code than in a relatively simple and small piece of code; similarly, round-off errors may be related to the amount of operations, fragmentation to the number of files opened, and so on. Hence, we assume that the software size, its complexity, the usage of some kinds of programming structures related to the resource management, the use of arithmetic operators, and other features are related to the occurrence of aging-related bugs.

Being able to relate the software complexity with software aging can be useful to mitigate the problem of software aging. For instance, by estimating the extent of aging issues in their software, developers could be able to invest the correct amount of testing efforts for detecting and fixing aging-related bugs, which can be a time-consuming activity and requires tailored techniques and tools [8]. Moreover, the software aging estimate can be useful to refine analytical models for selecting the best software rejuvenation schedule. The results from this study encourage this possibility, since they show that software aging effects are related to the static features of software.

The rest of the paper is organized as follows. In Section II, past work on software aging and complexity metrics is reviewed. A detailed description of the empirical study is provided in Section III, and the results are presented in Section IV. Section V summarizes the conclusions of the study, and presents some future research directions.

II. Related work

Software metrics have been widely used in the past for predictive/explanatory purposes. Much work was on investigating relationships between several kinds of software metrics and the number of faults in a program. Statistical techniques have been adopted in order to build regression models, also known as *fault-proneness* models. Such models allow developers to focus their attention on fault-prone software modules.

In [9], authors used a set of 11 metrics and an approach based on regression trees to predict most faulty modules. In [10], authors investigated metrics to predict the amount of post-release faults in five large Microsoft's software projects. They adopted the well-known statistical technique of Principal Component Analysis (PCA), in order to transform the original set of metrics into a set of uncorrelated variables,

978-1-61284-346-9/11/$26.00

with the goal of avoiding the problem of redundant features (*multicollinearity*). The study in [11], then extended in [12], adopted logistic regression to relate software measures and fault-proneness, in the context of homogeneous software products. Studies in [13], [14] investigated the suitability of metrics based on the software design.

In many cases, common metrics provide good prediction results, even across several different products. However, it is still difficult to claim that a given regression model or a set of regression models is general enough to be used even with very different products, as discussed in [10]. On the other hand, they are undoubtedly useful within an organization that collects faults data iteratively, during its development process. If a similar relationship could be found for aging-related bugs, developers would better deal with this phenomenon before the operational phase, e.g., by tuning techniques for detecting aging-related bugs according to such predictions.

III. Experimental design

A. Empirical data

In this study, we consider a set of complex software systems that revealed to be affected by software aging phenomena. Unfortunately, there are few field data studies that analyzed aging-related bugs, and they do not provide the raw data or detailed information about aging-related bugs. Instead, most studies analyzed the resource consumption trends, which are the manifestation of aging-related bugs.

The list of the considered systems is provided in Table I. Systems that come with detailed information are divided in modules, and the memory depletion trend caused by each module is provided. We focus on memory depletion since memory has the lowest TTE among the resources of computer systems [2], [3], and memory management bugs (e.g., memory leaks) represent the most considerable software aging source.

Table I: Aging data.

Software	Module	Aging (MB/h)
Sun JVM	GarbageCollector	2.933400
	JITCompiler	0.183600
CARDAMOM	Trace	3.574644
	Common	0.000102
	Repository	0.000029
	LoadBalancing	0.000041
	ORBSupport	0.000016
CARDAMOM OTS	Xerces	0.000008
	TAO	10.608754
Apache	Httpd	0.034442

A software aging analysis of Sun Java Virtual Machine (JVM) was conducted in [15]; that work pinpointed the memory depletion trends attributable to the Garbage Collector and the JIT Compiler, respectively. CARDAMOM is a CORBA-based middleware, and its memory leaks have been studied in [8]: the analysis found memory leaks in several modules. Moreover, the analysis included two OTS products, namely the Xerces XML parser and the TAO ORB, which are also considered in this study. Finally, we include the Apache web server, which was extensively studied in [3].

We base our analysis on resource consumption trends, assuming that *the software aging trends are correlated with the number and the severity of aging-related bugs*. Other factors affecting aging trends are represented by the *hardware* and the *workload* of the system, since they influence the type and rate of system operations. To minimize the workload influence, we consider *worst-case* aging trends of each system; this was possible since they were analyzed under several workload conditions. Moreover, the studies on the Sun JVM and CARDAMOM were conducted within our research group on the same hardware and operating system, thus minimizing the bias of testbed configuration; although the study on the Apache web server has been made on a different testbed, we decided to include it in the analysis, in order to compare our results with this well-known case study on software aging.

B. Software metrics

The adopted software metrics are summarized in Table II. These were automatically extracted by using the Understand tool for static analysis [16]. We include several metrics that revealed to be correlated with bug density in complex software—in this study, we evaluate if these metrics are correlated also with the specific class of *aging-related bugs*.

Table II: Software metrics.

Type	Metrics	Description
Program size	*LOC*, *CountLineCodeDecl*, *CountLineCodeExe*, *CountLineComment*, *CountDeclFileCode*, *CountDeclFileHeader*, *CountDeclClass*, *CountDeclFunction*, *CountLineInactive*, *CountStmtDecl*, *CountStmtExe*, *RatioCommentToCode*	Metrics related to the amount of lines of code, declarations, statements, and files
McCabe's cyclomatic complexity	*AvgCyclomatic*, *MaxCyclomatic*, *MaxNesting*, *CountPath*, *SumCyclomatic*, *SumEssential*	Metrics related to the control flow graph of functions and methods
Halstead metrics	*Program Volume*, *Program Length*, *Program Vocabulary*, *Program Difficulty*, *Effort*, *Bugs Delivered*	Metrics based on operands and operators in the program
Resource management	*Memory allocations*, *Memory deallocations*, *Files opened*, *Files closed*, *Difference in memory allocations/deallocations*, *Difference in files opened/closed*	Metrics based on resource management primitives (e.g., *mallocs*)

The first subset of metrics is related to the "size" of the program in terms of lines of code (e.g., total number of lines, and number of lines containing comments or declarations) and files. These provide a rough estimate of software complexity, and they have been already used as simple predictors of fault-prone modules [17]. Further metrics are here adopted, in order to improve fault prediction models, such as the McCabe's cyclomatic complexity and the Halstead metrics [18]. These are based on the number of paths in the code and the number of operands and operators, respectively. We hypothesize that these metrics are connected to aging-related bugs, since the complexity of error propagation (which is the distinguishing feature of aging-related bugs [19], [20]) may be due to the complex structure of a program. Finally, we introduce a set of metrics related to resource management, specifically defined for this study. These metrics are based on the number of calls of resource management primitives in the program; in particular, we focused on primitives for memory allocation and filesystem access, since the presence of these primitives introduce the opportunity of resource leakage.

IV. Results

A. Is there a correlation between metrics and aging?

To identify a relationship between metrics and aging, our first step is to evaluate the correlation with individual metrics. We evaluated the Pearson correlation coefficient between each metric and aging trends; this coefficient can be used to test a linear relation between two variables [7].

The Pearson coefficient for each metric are shown in Table III; metrics with a statistically significant relationship (p-value < 0.05) are highlighted. For instance, almost all the metrics related to the program size are correlated with software aging. This confirms the hypothesis that there exists a relationship between software aging and software complexity. Although some metrics do not exhibit a linear correlation with statistical significance, we do not exclude them from subsequent analysis: there could be a non-linear relationship (alone or in combination with other metrics) not found by this preliminary test.

Table III: Pearson correlation coefficient between individual metrics and aging trends.

Metric	Pearson coeff.	p-value
LOC	0.9202	0.0002
CountLineCodeDecl	0.9209	0.0002
CountLineCodeExe	0.9195	0.0002
CountLineComment	0.9061	0.0003
CountDeclFileCode	0.9282	0.0001
CountDeclFileHeader	0.9112	0.0002
CountDeclClass	0.9274	0.0001
CountDeclFunction	0.9307	0.0001
CountLineInactive	0.8960	0.0005
CountStmtDecl	0.9233	0.0001
CountStmtExe	0.9064	0.0003
RatioCommentToCode	-0.3632	0.3023
AvgCyclomatic	-0.1388	0.7021
MaxCyclomatic	0.8428	0.0022
MaxNesting	0.5486	0.1005
CountPath	0.4815	0.1589
SumCyclomatic	0.9126	0.0002
SumEssential	0.9148	0.0002
n_1 (mean)	-0.0792	0.8278
n_2 (mean)	-0.0663	0.8556
N_1 (mean)	0.0189	0.9587
N_2 (mean)	0.0322	0.9296
Volume (mean)	0.0228	0.9502
Length (mean)	0.0244	0.9466
Vocabulary (mean)	-0.0683	0.8513
Difficulty (mean)	-0.0250	0.9453
Effort (mean)	0.3980	0.2547
Bugs delivered (mean)	0.0572	0.8752
n_1 (variance)	-0.1544	0.6701
n_2 (variance)	-0.0143	0.9688
N_1 (variance)	0.5934	0.0705
N_2 (variance)	0.6502	0.0418
Volume (variance)	0.5928	0.0709
Length (variance)	0.6188	0.0565
Vocabulary (variance)	-0.0185	0.9595
Difficulty (variance)	0.3366	0.3415
Effort (variance)	0.9221	0.0001
Bugs delivered (variance)	0.8717	0.0010
Memory allocations	0.9280	0.0001
Memory deallocations	0.9208	0.0002
Diff. mem. alloc.-dealloc.	-0.6538	0.0403
Files opened	0.8410	0.0023
Files closed	0.8920	0.0005
Diff. files opened-closed	0.1556	0.6678

B. Can we build a regression model for software aging?

We adopted statistical regression models to obtain a quantitative relationship between software metrics and aging. Since a linear correlation with some metrics was observed, we evaluated a multiple linear regression model (see the Appendix for mathematical definitions). In order to build this model, we have to deal with the mutual correlation among metrics (e.g., a software with high LOC will probably have a high number of function declarations); this correlation leads to an unstable model, since small variations in the data set may result in large variations of the model [21]. Therefore, we adopted the *stepwise* method to build the model, that is, distinct variables are introduced or removed from the model and a statistical significance test is performed to select the best model [22]. This method produced a linear model with only one variable, namely *CountDeclFunction*, which is shown in Figure 1.

Table IV provides some statistics about this model in the first column. The model is characterized by a high standard deviation of residuals (1.3185 MB/h). This high variance significantly affects the prediction for software modules with a low aging trend ($\ll$ 1 MB/h), leading to a very high average relative error ($1.686 \cdot 10^6\%$). Moreover, the independent variables are characterized by a very high inter-correlation, since only one variable has been introduced into the model by the stepwise method.

Table IV: Regressions statistics.

Procedure	Stepwise	PCA	Stepwise	Stepwise	Stepwise
Sample	All software	All software	*LittleAging* group	*BigAging* group	*BigAging* group (except Trace)
$\mathbf{R^2}$	0.8662	0.9013	0.9896	0.9277	0.9999
Adjusted $\mathbf{R^2}$	0.8494	0.7770	0.9791	0.8915	0.9999
Standard Error (MB/h)	1.3185	1.6013	5.3911E-06	1.4625	0.0127
Average relative error (%)	1.686E+06	1.096E+06	11.4922	154.5025	6.81

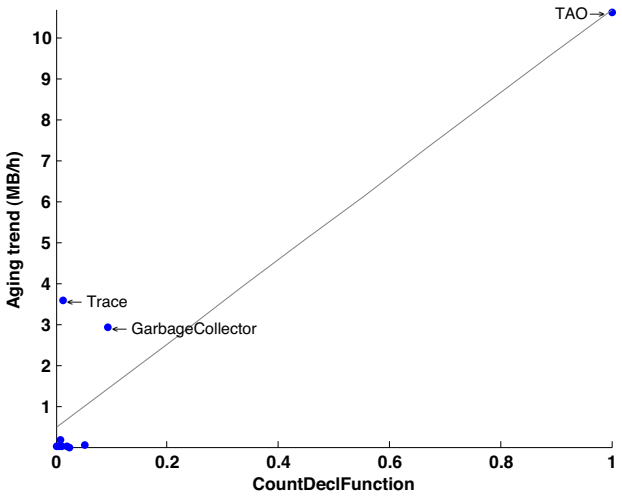


Figure 1: Linear regression over all software.

In order to obtain a more precise model, we also adopted the Principal Component Analysis (PCA) technique, which transforms the dependent variables in a low number of uncorrelated variables [21]; nevertheless, this approach did not improve the model (see the second column in Table IV). We also evaluated an exponential and a logarithmic model, but they also were not able to provide better precision.

We hypothesized that the lack of a simple and precise model is due to heterogeneity of the software modules. A noticeable feature of the dataset is the very large range of values in the aging trends—they differ by several orders of magnitude. Therefore, we separated the dataset into two disjoint groups, namely *BigAging* and *LittleAging*, which were individually analyzed (Table V). We only considered two groups due to the low number of software modules.

After splitting the dataset, we applied the stepwise method to the groups; we obtained a much more precise model in both cases (see the third and fourth columns in Table IV). Both the models satisfy the hypotheses of residuals' homoscedasticity, normality, and uncorrelation with the independent variables. In particular, the model for the *LittleAging* group is characterized by a low standard deviation and an acceptable average relative error (about 11%). Figure 2a shows the linear model for the *LittleAging* group. The dependent variables included in this model (*RatioCommentToCode* and *CountLineInactive*) do not appear to be representative of software complexity; however, given the high correlation between the dependent variables, we conclude that the aging trend of software of this group is related to the program size (both the dependent variables belong to this type of metrics).

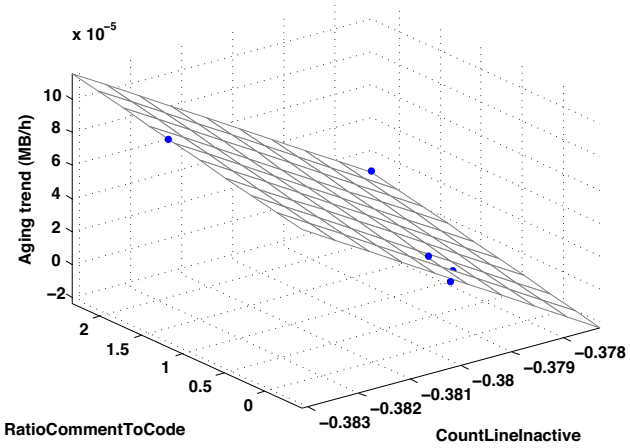
Table V: Software grouped by order of magnitude of aging trends.

Module	**Aging (MB/h)**	**Group**
TAO	10.608754	BigAging
Trace	3.574644	BigAging
GarbageCollector	2.933400	BigAging
JITCompiler	0.183600	BigAging
Httpd	0.034442	BigAging
Common	0.000102	LittleAging
LoadBalancing	0.000041	LittleAging
Repository	0.000029	LittleAging
ORBSupport	0.000016	LittleAging
Xerces	0.000008	LittleAging

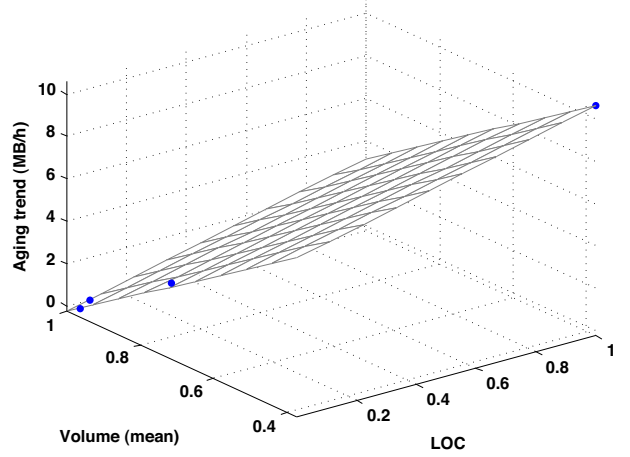
Although the model for the *BigAging* group is significantly better than the initial model, it is still characterized by a high error (about 154%). We suspected that the error was due to the presence of the Trace module in the group, since it is characterized by a low complexity and a high aging trends (e.g., it has a low *CountDeclFunction*, as shown in Figure 1). Therefore, we treated this sample as an outlier and removed it from the group; the resulting model was much more accurate (see the fifth column in Table IV and Figure 2b), with a low average relative error (about 6%). We believe that this result was due to the immaturity of the Trace module, which was affected by severe aging-related bugs even if it was a relatively simple module [8]. In this group, aging is related to the size of the modules (*LOC*); the model also accounts for the complexity of the code (*Volume*).

Finally, in order to explain the difference between the two groups, and to apply the right model to a new software (i.e., not included in our dataset), we evaluated if it is possible to classify the modules into groups using software metrics.

To select the best features (i.e., metrics) to use in the classifier, we applied a feature selection technique, namely the *Independent Features* procedure [23]. This procedure performs a statistical test for each individual feature, indicating that the difference in the means is unlikely to be



(a) *LittleAging* group.



(b) *BigAging* group.

Figure 2: Linear regression models.

random variation; if the difference is sig times lower than the standard error, then the feature is not deemed useful for classification. The test is performed by evaluating

$$se(A-B) = \sqrt{\frac{var(A)}{n_A} + \frac{var(B)}{n_B}} \quad (1)$$

$$\frac{|mean(A) - mean(B)|}{se(A-B)} > sig \quad (2)$$

where A and B are the same feature measured for the two classes, and n_A and n_B the number of samples in the classes.

We considered several sets of features obtained using different values of sig. We evaluated the effectiveness of each set of features using the *leave-one-out* procedure: $n-1$ samples are used for training a classifier, and the remaining sample is used for testing the classifier; this step is repeated for different splittings of the dataset. For classification we adopted the two-class SVM classifier, using the LIBSVM implementation [24].

Table VI shows the results of leave-one-out validation of the best classifier ($sig = 3.4$). This classifier is effective in 9 out of 10 cases; it is not accurate in the case of the Trace module, which was previously shown to be an anomalous sample. Feature selection and leave-one-out validation were repeated without the Trace Service, and the best classifier correctly classified the samples in all cases. This result supports the use of software metrics for classifying the software with respect to software aging. The metrics of the best classifier were *Volume (mean)*, *Effort (mean)*, *Volume (variance)*, N_1 *(variance)*, N_2 *(variance)*, *Length (variance)*.

Table VI: Leave-one-out validation (L = *LittleAging*, B = *BigAging*) for $sig = 3.4$.

Module	Reference class	Predicted class
GarbageCollector	B	B
JITCompiler	B	B
Trace	B	L
Common	L	L
Repository	L	L
LoadBalancing	L	L
ORBSupport	L	L
Xerces	L	L
TAO	B	B
Httpd	B	B

V. Conclusions and Future Work

In this study, we investigated a relationship between software metrics and software aging on ten software applications. The results of the analysis can be summarized as follows:

1) Software applications belong to two distinct groups, namely, software in which the aging effects are negligible (*LittleAging*), and software significantly affected by software aging (*BigAging*). We did not find a simple model able to predict aging effects of both groups at the same time, thus they had to be analyzed separately.
2) There exist two precise multiple linear regression models for modeling the two software application groups. Aging trends in the *LittleAging* group seem to be related with the program size, while the complexity of the program in terms of operands and operators (i.e., Halstead metrics) should be also taken into account for the *BigAging* group.
3) It is possible to classify software applications in one of the two groups by using software metrics. The Halstead metrics turned out to be the most suitable for this purpose.

These results encourage the use of software metrics for coping with software aging at development time. The classification of a new software could be made by identifying its category (*LittleAging* or *BigAging*), and then applying a tailored linear regression model. However, to achieve this goal more research is needed in the following directions:

1) To extend the analysis and validate the results by including additional software. Unfortunately, the limited amount of published data on software aging does not allow to draw definitive conclusions.
2) To analyze the relationship with actual aging-related bugs. Due to the lack of data about aging-related bugs, we based our analysis on the *worst-case* aging trend exhibited by the programs. This will require to collect and analyze data from bug repositories to discover the actual aging-related bugs. The expected number of aging-related bugs could be useful for developers to evaluate their progress in fixing these bugs.

APPENDIX
DEFINITIONS

Multiple linear regression model A.1 *The linear relationship between the independent variables X and the expected dependent variable $\widehat{Y}$:* $\widehat{Y} = b_0 + b_1X_1 + b_2X_2 + \ldots + b_kX_k + e$

Sum of Square Total A.2 *The deviance between the samples and the average value:* $SST = \sum_i (y_i - \overline{y})^2$

Sum of Square of Regression A.3 *The deviance between the model and the average value:* $SSR = \sum_i (\widehat{y}_i - \overline{y})^2$

Sum of Square of Error A.4 *The deviance between the samples and the model:* $SSE = \sum_i (y_i - \widehat{y}_i)^2$

$\mathbf{R^2}$ A.5 *The share of the total deviance explained by the model:* $R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$

Adjusted $\mathbf{R^2}$ A.6 *The R^2 adjusted to account for the number of variables in the model:* $R^2_{adj} = 1 - \frac{SSE}{n-k-1} / \frac{SST}{n-1}$

Standard Error A.7 *The standard deviation of residuals:* $S = \sqrt{\frac{SSE}{n-2}} = \sqrt{\sum_i (y_i - \widehat{y}_i)^2 / n-2}$

Average Relative Error A.8 *The relative size of residuals with respect to the samples:* $E = \frac{1}{n} \sum_i |y_i - \widehat{y}_i| \,/\, |y_i|$

ACKNOWLEDGMENT

This work has been partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the 7th Framework Programme (FP7), and by the Italian Ministry for Education, University, and Research (MIUR) within the framework of the project "Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures" (DOTS-LCCI, http://dots-lcci.prin.dis.unina.it), DM1407.

REFERENCES

[1] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Intl. Symp. on Fault-Tolerant Computing*, 1995.
[2] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A Methodology for Detection and Estimation of Software Aging," in *Intl. Symp. on Software Reliability Engineering*, 1998.
[3] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, vol. 55, no. 3, 2006.
[4] K. Vaidyanathan and K. Trivedi, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," in *Proceedings of the International Symposium on Software Reliability Engineering*, 1999.
[5] Y. Bao, X. Sun, and K. Trivedi, "A Workload-Based Analysis of Software Aging, and Rejuvenation," *IEEE Transactions on Reliability*, vol. 54, no. 3, 2005.
[6] K. Vaidyanathan and K. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Trans. on Dependable and Secure Computing*, vol. 2, no. 2, 2005.
[7] V. Basili and B. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, vol. 27, no. 1, 1984.
[8] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory Leak Analysis in Mission-Critical Middleware," *Journal of Systems and Software*, vol. 83, no. 9, 2010.
[9] S. Gokhale and M. Lyu, "Regression Tree Modeling for the Prediction of Software Quality," in *Intl. Conf. on Reliability and Quality in Design*, 1997.
[10] N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *28th Intl. Conf. on Software Engineering*, 2006.
[11] G. Denaro, S. Morasca, and M. Pezzè, "Deriving models of software fault-proneness," in *14th Intl. Conf. on Software Engineering and Knowledge Engineering*, 2002.
[12] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *24th Intl. Conf. on Software Engineering*, 2002.
[13] A. Binkley and S. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Run-time Failures and Maintenance Measures," in *20th Intl. Conf. on Software Engineering*, 1998.
[14] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. on Software Engineering*, 1996.
[15] D. Cotroneo, S. Orlando, and S. Russo, "Characterizing Aging Phenomena of the Java Virtual Machine," in *26th IEEE Symp. on Reliable Distributed Systems*, 2007.
[16] Scientific Toolworks Inc., *What metrics does Understand have?*, http://www.scitools.com/documents/metrics.php.
[17] T. Ostrand, E. Weyuker, and R. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, 2005.
[18] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. on Software Engineering*, vol. 25, no. 5, 1999.
[19] M. Grottke and K. Trivedi, "Software Faults, Software Aging and Software Rejuvenation," *Journal of the Reliability Engineering Association of Japan*, vol. 27, no. 7, 2005.
[20] M. Grottke, A. Nikora, and K. Trivedi, "An Empirical Investigation of Fault Types in Space Mission System Software," in *Intl. Conference on Dependable Systems and Networks*, 2010.
[21] T. Khoshgoftaar, E. Allen, K. Kalaichelvan, and N. Goel, "Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, vol. 13, no. 1, 1996.
[22] T. Khoshgoftaar and J. Munson, "Predicting software development errors using software complexity metrics," *IEEE J. on Selected Areas in Communications*, vol. 8, no. 2, 1990.
[23] S. M. Weiss and N. Indurkhya, *Predictive Data Mining: A Practical Guide*. Morgan Kaufmann, 1997.
[24] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, http://www.csie.ntu.edu.tw/~cjlin/libsvm.