How Long does a Bug Survive? An Empirical Study

Gerardo Canfora*, Michele Ceccarelli**, Luigi Cerulo**, Massimiliano Di Penta*
∗ Dept. of Engineering-RCOST, University of Sannio, Italy
∗∗ Dept. of Science, University of Sannio, Italy
{canfora, ceccarelli, lcerulo, dipenta}@unisannio.it

***Abstract*—Corrective maintenance activities (bug fixing) can be performed a long time after a bug introduction, or shortly after it. Such a time interval, i.e., the bug survival time, may depend on many factors, e.g., the bug severity/harmfulness, but also on how likely does the bug manifest itself and how difficult was to fix it.**

This paper proposes the use of survival analysis aimed at determining the relationship between the risk of not fixing a bug within a given time frame and specific source code constructs—e.g., expression operators or programming language constructs—changed when fixing the bug. We estimate the survival time by extracting, from versioning repositories, changes introducing and fixing bugs, and then correlate such a time—by means of survival models—with the constructs changed during bug-fixing.

Results of a study performed on data extracted from the versioning repository of four open source projects—Eclipse, Mozilla, OpenLDAP, and Vuze—indicate that long-lived bugs can be characterized by changes to specific code constructs.

Keywords: Bug Fixing, Survival Analysis, Empirical Study.

I. Introduction

When developers maintain or evolve a software system, they introduce bugs. These bugs survive in the system until someone reveals a failure—i.e., how the bug manifests itself as an unexpected system behavior—and how who discovers the bug reports the failure to the development team.

The time a bug can survive in a system since its introduction can sensibly vary, depending on various factors. In some cases, the bug could survive for a long time, because it does not cause major problems to the system behavior, e.g., it seldom manifests itself, or its effects do not compromise the system functionality. In other cases, however, the bug could remain in the system for a long time because it is difficult to identify. In general, trying to ensure that bugs—and especially critical bugs—are fixed before issuing a new system release on the market, is quite desirable because it would limit post-release corrective maintenance. For this reason verification & validation (V & V) activities, i.e., code inspection, static analysis and testing, need to be performed; in some cases, however, the limited time or resources available make a thorough V & V infeasible, and a prioritization would be necessary. Approaches such as that of Gyimóthy *et al.* [1] use source code metrics to identify likely fault-prone classes that require a better V & V.

With respect to other works aimed at prioritizing V & V activities, we are interested to understand whether changes to specific code construct, such as control flow statements, synchronization constructs, exception handling, or different kinds of expressions, could lead to bugs having a longer survival time than others. Some authors, in particular Kim *et al.* [2], [3], characterized a change by a set of features extracted from it, such as the change length, variation of cyclomatic complexity, plus the textual content of the change, and used machine-learning algorithms to predict bugs from their introducing changes. Rather than aiming at building a bug predictor, as Kim *et al.* did, our aim is to identify what kinds of code constructs, introduced in the context of a bug introducing change and then fixed in the context of a bug fixing, could relate to bugs having a longer (or shorter) survival time than others. Such findings would be useful to provide developers with indications whether code containing particular constructs (or combinations of them) deserves a better V & V, not just because it is likely fault-prone, but also because the defect, if present, may remain in the system for a long time.

This paper reports an empirical study aimed at characterizing the survival of bugs in four open source systems, Eclipse, Mozilla, OpenLDAP, and Vuze. Specifically, we characterize each bug with respect to specific source code constructs involved in fixing changes, for example control structures, expression operators, exception handling, etc. We measure the bug survival time by identifying the bug-fixing commits and—through CVS annotations—the bug introducing changes, and correlate—using the Cox proportional hazard model [4]—such a time with code constructs extracted from the bug-fixing change. We focus on two research questions, namely:

1) whether there are specific code constructs—identified in bug fixing changes—that specifically characterize long-lived bugs as opposed to short-lived bugs, and
2) whether the interaction—i.e., co-occurrence—of specific code features characterizes long- and short-lived bugs.

The study indicates that there are specific code constructs that characterize bugs having a longer survival time than others, while some other constructs characterize bugs being fixed in a relatively short time. Above all, in many cases a bug survival time can be better characterized in terms of the interaction of multiple constructs, which can have an effect different from the single constructs.

The paper is organized as follows. Section II provides background notions on survival analysis. Section III describes the approaches used to identify bug-introducing changes and to extract constructs from bug-fixing changes, and explains how we characterized bug survival time with respect to code constructs changed during bug-fixing. Section IV describes the empirical study we performed. Results are reported and discussed in Section V, while Section VI discusses the threats to validity. Section VII discusses the related literature, and Section VIII concludes the paper and identifies directions for future work.

II. Background on Survival Analysis

Survival analysis is a set of statistical procedures for which the outcome variable of interest is *time until an event occurs*. The event is usually meant as death for biological organisms or failure for mechanical systems, but in general it may be any designated experience of interest that may happen to a subject (e.g., the survival time of a bug since its introduction in the system). The survival time variable, T, is referred as a random variable, as it gives the time a subject has "survived" over some follow up period. The survival function $S(t) = Pr(T > t)$ gives the probability that a subject survives longer than some specified time t. The survival function does not increase as t increases and conventionally, $S(0) = 1$, is the start of a study, and, for time $t \to \infty$, $S(\infty) \to 0$. The hazard function $h(t)$ gives the instantaneous potential per unit time for the event to occur, given that the subject has survived up to time t. Survival and hazard functions are in essence opposed concepts, in that the survival function focuses on surviving whereas the hazard function focuses on failing, given survival up to a certain time point. The instantaneous failure rate, i.e., the probability to fail at time t, is given by $f(t) = S(t)h(t)$. Survival and hazard functions are related by the following equations:

$$S(t) = \exp\left[-\int_0^t h(u)\,\mathrm{d}u\right] \qquad h(t) = -\frac{dS(t)}{dt}\frac{1}{S(t)}$$

The goal of survival analysis is to estimate survival and hazard functions from data (Kaplan–Meier estimator [5]), and to assess the relationship of explanatory variables (covariates) with survival time.

Survival analysis offers two classes of regression models to perform such tasks: parametric models, when survival time follows some known distribution [6], such as Exponential, Weibull, or Log-logistic; and proportional hazard models—e.g., the semi-parametric Cox proportional hazard model [4]—making no assumption about the distribution of survival time, and thus suitable for cases like ours where such an assumption does not always hold.

In both cases the response variable of interest is the hazard function $h(t)$ which is expressed in terms of a baseline hazard function $h_0(t)$, a set of explanatory variables, $X =< X_1, X_2, \ldots, X_n >$, and regression coefficients $\beta =< \beta_1, \beta_2, \ldots, \beta_n >$. An explanatory variable could be also the product of two or more other variables, modeling their interactions.

The Cox proportional hazard model has an hazard function defined as: $h(t) = h_0(t) \cdot e^{\beta X}$. It is less restrictive than parametric models as the baseline hazard function, $h_0(t)$, is left unspecified, and the only assumption is that the hazards of two individuals, with different X values, are proportional. Under such assumptions it is possible to estimate β in the exponential part of the model by maximizing a likelihood function. The β parameters can be used to define the hazard ratio (HR) effect size measure for each explanatory variable, which is the multiplicative effect of such a variable on the risk of the event to occur. In general, a HR is defined as the hazard for one individual divided by the hazard for a different individual distinguished by their values for the X variables. The exponential of a parameter β_i is the HR between an individual, X^i, with the i-th explanatory variable set to one and all others set to zero, and a reference individual, X^0, with all variables set to zero:

$$\frac{h_{X^i}(t)}{h_{X^0}(t)} = \frac{h_0(t) \cdot e^{\beta_i}}{h_0(t) \cdot e^{\beta_1 \cdot 0 +, \beta_2 \cdot 0 +, \ldots, + \beta_n \cdot 0}} = e^{\beta_i}$$

A HR greater than one for an explanatory variable (feature) indicates that the presence of such a feature lead to a higher probability than for other cases of the event (bug fixing in our case) to occur. In other words, $HR > 1$ indicates, for that variable, a lower survival time than in other cases. Vice versa, a $HR < 1$ indicates a higher survival time than in other cases. Further details on survival analysis can be found in a book by Kleinbaum and Klein [6].

III. Identifying Bug-Introducing Changes and Survival Time

To build a bug survival model we need to extract, from versioning systems, the dependent variable (survival time) and independent variables (code constructs changed in bug fixing). This section explains how this is done through a sequence of four steps.

A. Step 1: identification of bug fixing from the commit note

First, we download the versioning system (CVS in our study) log and extract the information relevant for our study, specifically for each commit the file changed, its revision, the commit timestamp, the committer id, and the commit note. We cluster together related commits into change sets using the heuristic by Zimmermann *et al.* [7], which groups together commits performed by the same committer, having the same commit note and a temporal distance smaller than 200 seconds.

Then, we restrict our attention to change sets referring to bug fixes, i.e., those matching a pattern such as *bug #ID*, *issue #ID*, or similar, where *#ID* is a valid bug ID from the bug tracking system of the project [8]. For projects (such as Vuze in our study) where commit notes did not contain explicit references to bug IDs, we select change sets where the commit notes contain patterns referring to a likely bug fixing—e.g., *issue fixed*—identified by means of a manual analysis of the log. Finally, we further restrict to only bug fixings for which a limited set of files (maximum 3) were changed. In other words, we only consider very focused bug fixings for which it is possible to precisely characterize the fix change in terms of modified code constructs.

B. Step 2: identification of bug-introducing changes

To identify the change(s) in which a bug was introduced, we used an approach inspired by the work of Kim *et al.* [2], [3]. Specifically, we rely on the CVS annotation which, given

TABLE I
CHANGED CONSTRUCTS EXTRACTED FROM THE SOURCE CODE.

NAME	DESCRIPTION	EXAMPLES
expr..array	Array access	[]
expr..arith	Arithmetic operators	$+, -, *, /, \%$
expr..bit	Bit operators	&, |, <<, >>
expr..boolean	Boolean operators	||, &&, !
expr..comparison	Comparison operators	$>, \geq, <, \leq, ==$
expr..field	Structure field access	$->$, .
call	Function call	foo(), printf(...)
ctrl	Control structures	for, while, if, switch, do
decl	Declarations	class, interface, extends
exception	Exception handling	try, catch, throw, finally
import	Java import	import
obj	Object creation/referencing	new, super, this
qualifier	Qualifiers	private, public, protected, static
sync	Synchronization constructs	synchronized

a file revision, indicates for each file line the revision when the last change to that line occurred. In essence, the approach works as follows:

1) First, for each file f_i involved in the bug fixing, we extract the file revision *before* the bug fixing.
2) Then, we identify blank lines and lines that only contain comments using an island parser developed in Perl.
3) Finally we use the *annotate* option of CVS to identify, for each source line, when it was changed before the bug fixing. In doing this, we exclude the blank and comment lines identified in the previous step. This produces, for each file f_i, a set of revisions $r_{i,j}$ that likely introduced the bug in that file, excluding however cases where the revision when the bug was introduced is revision 1.1, i.e., the entire file was created.

C. Step 3: identification of code constructs changed during bug-fixing

To identify source code constructs that were changed in the bug fix, we use a tokenizer implemented in Perl to extract tokens from f_i revisions before and after the fix, and to identify tokens which occurrences changed between the two revisions. In this study, we consider different kinds of tokens—reported in Table I—corresponding to various programming language constructs. We group together related tokens, e.g., all Boolean operators, all arithmetic operators, control structures, etc. Finally, it is worthwhile noticing that we used different analyzers for Java, C, and C++, in that some constructs apply to specific languages only, e.g., $->$ only to C and C++, *try*, *catch*, *new* only to C++ and Java, *synchronized* only to Java.

It is important to explain that in this context we extracted the constructs modified *when fixing* the bug, rather than *when introducing* it. The reason is that we are interested to identify what code constructs were changed when fixing a bug as, instead, the bug introducing change would have involved many other constructs related to that change, and that do not necessarily induced the bug.

D. Step 4: estimation of the survival time

To estimate how long did a bug remain in the system, we identify: (i) *the latest fixing timestamp*, i.e., the highest timestamp among all commits of the bug-fixing change set. That is, when all changes needed to fix the bug were committed; and (ii) *the latest bug introducing timestamp*, i.e., the latest revision among the ones identified in *Step 2*.

TABLE II
CHARACTERISTICS OF THE DATA SET USED IN THE STUDY.

SYSTEM	TIME INTERVAL CONSIDERED	# BUG FIXINGS	SAMPLE SIZE
Mozilla	Apr 1998–Feb 2011	43,568	3,858
OpenLDAP	Aug 1008–Feb 2011	2,808	522
Eclipse	May 2001–Aug 2008	16,077	5,137
Vuze	Jul 2003–Apr 2010	1,591	604

IV. EMPIRICAL STUDY

The *goal* of this study is to analyze the relation between source code constructs modified in bug-fixing changes and the permanence of such bugs, i.e., survival time, in the system. The *purpose* is to understand how a longer (or shorter) survival time correlates with specific kinds of source code changes. The *quality focus* is related to software fault-proneness, and specifically to exploring code constructs that can characterize bug fixings performed a short time (or a long time) after the bug has been introduced. The *perspective* is mainly of researchers interested to explore the nature of bug fixings, with the aim of identifying construct that could contribute to increase, or decrease, the permanence of a bug in a software system.

The *context* consists of a sample of bugs extracted from four open source software systems, belonging to different domains and developed with different programming languages, namely two C/C++ systems—Mozilla and OpenLDAP—and two Java systems, Eclipse and Vuze. Mozilla[1] is a suite comprising a Web browser, an email client, and other Internet utilities. OpenLDAP[2] is an open source implementation of the Lightweight Directory Access Protocol (LDAP). Eclipse[3] is an open-source integrated development environment. It is a platform used both in open-source communities and in industry. Vuze[4]—known also as Azureus—is an open source BitTorrent client written in Java. BitTorrent is a protocol that allows to exchange files over the Internet. Table II reports, for the four projects, some overall information relevant for our study, i.e., the period of time considered, the number of bug-fixing change sets, and their subset considered in our study. The sample consists of randomly selected bugs among those occurred in the time interval considered. As explained in Section III—since we are interested to relate fixings of specific code constructs with the survival time—we only considered bug fixings involving up to 3 files.

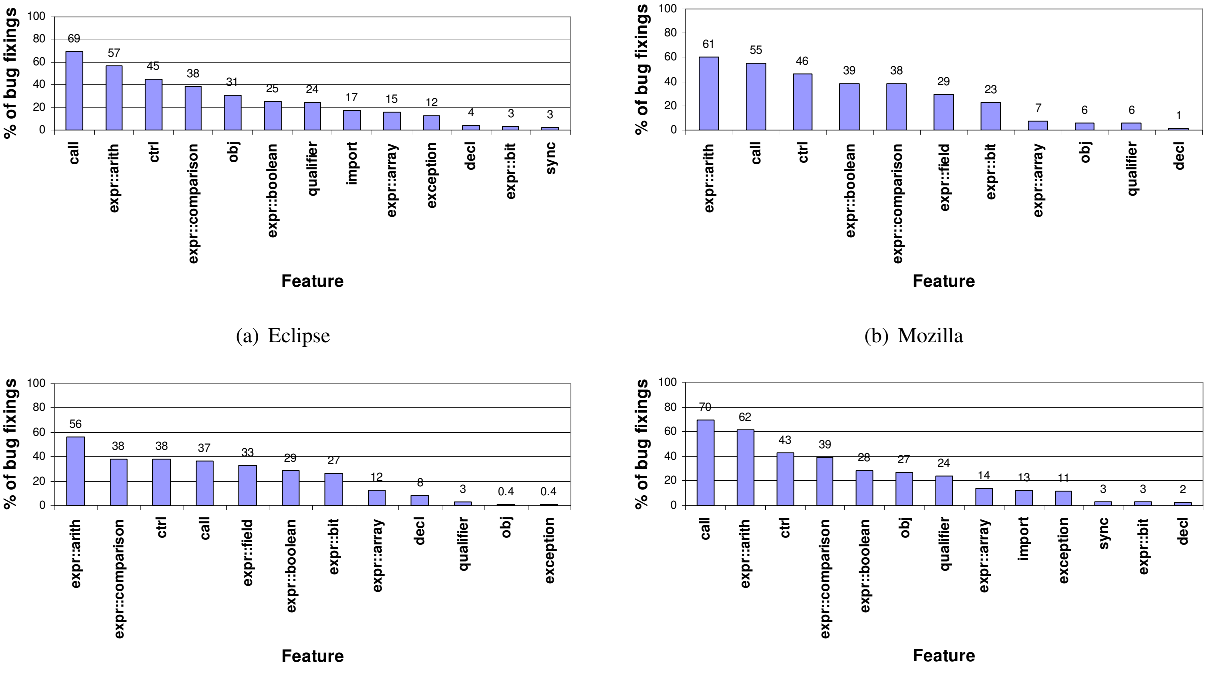
Figure 1 shows, for each system, the distribution of the permanence of a bug into a system, i.e., survival time. Figure 2 shows, for the four systems, the percentages of bug fixing change sets—in the considered sample—involving different kinds of code constructs identified with the procedure described in Section III.

[1] http://www.mozilla.org

[2] http://www.openldap.org

[3] http://www.eclipse.org

[4] http://www.vuze.com



(a) Eclipse

(b) Mozilla

(c) OpenLDAP

(d) Vuze

Fig. 2. Percentages of bug fixings involving different kinds of code constructs.

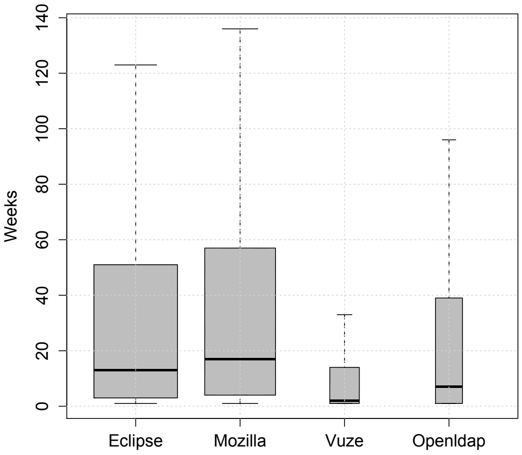


Fig. 1. Distribution of the bug survival time (in week) for each system.

A. Research Questions

This study aims at addressing the following research questions:

- **RQ1:** *What are the code constructs characterizing long-lived bugs, as opposed to bugs quickly removed?* The rationale of this research question is to understand whether bug fixes performed shortly after their introduction consist in different kinds of changes than bug fixes performed a long time after their introduction. In this study, we characterize changes based on the programming language constructs being modified. The conjecture is that some language constructs—e.g. pointer (de)referencing, exception-related keywords, bit-manipulation operators—can represent symptoms of subtle bugs, difficult to spots or, on the other way around, of bugs easier to manifest and fix than others. The analysis carried out in this research question would help to understand the origin of bugs that tend to remain in the system for a short or for a longer time, and could therefore suggest to better test the changed artifacts when the change involved some specific programming language constructs. For example, it can happen that errors related to some specific expression operators can be easily found, while an improper exception handling might result more difficult to find and thus the error would remain in the system for a long time.
- **RQ2:** *Is there any significant interaction between code constructs with respect to their effect on the risk of a bug to remain in the system for a long time?* The rationale here is similar to **RQ1**, however in this case we do not consider each code construct separately, we rather consider their interaction, e.g., a short-lived (or long lived) bug is fixed by changing together a conditional statement and a comparison operator, or a loop statement and an exception handling construct. It can happens for example that a change involving a particular code construct increases, or decreases, the survival time of

a bug when it is applied in conjunction with another construct.

B. Analysis Method

This section describes the statistical procedures used to address the research questions formulated above.

The selection of parameters to be included in the model is performed using the Akaike Information Criterion (AIC) [9]. The AIC measures the goodness of fit of a statistical model, and specifically relies on the concept of information entropy to provide an estimation of the information lost when a particular model is used. The AIC is often used (as in our case) with an iterative procedure that computes the AIC for different models (in our case, models considering different combinations of the considered code constructs). To select the optimal model we used the *stepAIC* procedure of the *MASS* R package, using $1,000$ steps as the maximum stopping criterion. Such a number has been calibrated increasing the number of steps until we did not observe any change in the obtained model.

We repeat the AIC procedure for different kinds of survival models, namely parametric models where the bug survival in terms of code constructs is fitted with respect to some distributions (logistic, exponential, or Weibull), and the semi-parametric Cox proportional hazard model. Also, for parametric models, we use the χ^2 goodness-of-fit test to check whether the empirical model fits the theoretical distribution, H_0*: there is no significant difference between the theoretical and the empirical model*, while for the Cox model we use appropriate tests, i.e.,the Likelihood ratio test, the Wald test, and the logrank test (for these tests, the null hypothesis must be rejected in order to have a significant Cox model) [4]. In addition, the Cox proportional hazard assumption is validated through a proportion test (using χ^2 goodness-of-fit) on each construct involved in the model.

For both **RQ1** and **RQ2**, we show, for the four systems, a table with the set of code constructs (single constructs for **RQ1**, both single constructs and pairwise interaction of constructs for **RQ2**) that have a significant effect on the survival time, together with the HRs defined in Section II. Also, we plot and discuss the empirical Kaplan–Meier estimated survival curves, in which the x-axis indicates the time (in weeks) and the y-axis indicates the observed proportion of not-fixed (survived) bugs in which the construct being plotted is present. The curves are plotted with respect to a *baseline* curve, that is the average survival curve of all subjects. This means that the survival curve of a construct with $HR > 1$ is below the baseline, while it is above when $HR < 1$.

V. Results

This section reports and discusses results of the empirical study defined in Section IV. Raw data and working data sets are available for replication purposes[5].

TABLE III
HR OF DIFFERENT CODE CONSTRUCTS FOR THE COX MULTIVARIATE MODEL (WITHOUT INTERACTIONS).

CODE CONSTRUCT	ECLIPSE	MOZILLA	OPENLDAP	VUZE
expr..arith	–	**1.14**	–	**1.27**
expr..array	**1.13**	**0.83**	–	1.22
expr..comparison	**1.09**	1.06	–	1.17*
expr..field	–	**1.11**	–	–
call	**1.12**	–	–	1.16
exception	1.08	–	–	–
obj	**1.09**	**1.36**	–	–
qualifier	**1.09**	–	–	–
sync	–	NA	NA	1.63*

A. RQ1: What are the code constructs characterizing long-lived bugs, as opposed to bugs quickly removed?

First, we use AIC to determine which distribution was better suited to build the survival model without interaction, as well as to select the code constructs to be used in the model. In all cases, the Weibull model exhibited the lowest AIC, indicating that possibly the Weibull distribution could be suitable to model the bug survival in terms of fixed code constructs. However, when building the Weibull model—and the same happened for other parametric models, i.e., logistic and exponential—the χ^2 test provided a p-value< 0.05, i.e., the model significantly deviates from the distribution. For this reason, we have decided to adopt the Cox proportional hazard model (although its AIC is higher), which is a semi-parametric model and does not require a fitting with a particular distribution. The Cox models identified by the AIC passed all tests (Likelihood ratio test, Wald test, and logrank test) with a p-value < 0.05, and did not reject the χ^2 hypothesis (p-value > 0.05) for the test of proportional hazard assumption with the exception of the *ctrl* statements for Eclipse, that was therefore excluded.

Table III reports HRs provided by the Cox proportional hazard model. Significant factors (p-value <0.05) are shown in boldface, while marginally significant ones ($0.5 \leq$ p-value < 0.1) are annotated with a "*". "NA" indicates that a code construct is not applicable for the programming language used in that system (e.g., *synchronize* does not exist in C), while "–" indicates that the code construct was not included in the Cox model by the AIC. As it can be noticed, the OpenLDAP column is empty. This is because the AIC procedure, for the model without interaction, was not able to select any construct.

Results shown in Table III suggest that, although there are code constructs that—without considering the interaction with others—significantly affect the bug survival time, this is not always consistent across systems, and, in some cases, the constructs do not have a significant effect on the survival time. Nevertheless, some of the HRs can provide some indications already:

- *Array access* was included in all models, although it does not have a significant effect for Vuze. Despite such a construct was involved in a small percentage of bug fixings ($\leq 15\%$, as shown in Figure 2), it seems to have

[5] http://www.rcost.unisannio.it/mdipenta/survival-data.tgz

TABLE IV
HR OF DIFFERENT CODE CONSTRUCTS FOR THE COX MULTIVARIATE MODEL (WITH INTERACTIONS).

CODE CONSTRUCT	ECLIPSE	MOZILLA	OPENLDAP	VUZE
expr..arith	–	**1.14**	–	0.96
expr..array	**1.14**	1.03	1.22	1.27*
expr..bit	–	1.06	1.23	**2.89**
expr..boolean	–	0.95	–	0.76
expr..comparison	1.10	1.04	–	1.06
expr..field	–	**1.34**	1.11	–
call	1.06	**1.10**	1.06	0.86
ctrl	**1.17**	1.02	1.14	1.03
decl	1.15	0.65	1.23	–
exception	**1.28**	–	–	**1.63**
import	0.97	NA	NA	–
obj	0.88*	1.20	–	1.52
qualifier	–	1.22	**2.48**	**0.48**
sync	0.85	NA	NA	**3.47**
expr..arith:call	–	–	–	**1.51**
expr..arith:decl	–	1.97	–	–
expr..arith:obj	–	–	–	**0.46**
expr..arith:qualifier	–	–	–	**2.46**
expr..array:obj	–	0.74	–	–
expr..array:expr..bit	–	–	1.69	–
expr..array:expr..comparison	–	**0.73**	–	–
expr..array:qualifier	–	–	**0.09**	–
expr..bit:call	–	–	–	**0.20**
expr..bit:ctrl	–	**0.79**	**0.56**	–
expr..bit:expr..field	–	0.86	–	–
expr..bit:qualifier	–	1.32	–	–
expr..boolean:call	–	–	–	1.53*
expr..boolean:expr..bit	–	1.21*	–	–
expr..comparison:ctrl	0.87*	–	–	–
expr..comparison:decl	0.74*	–	–	–
expr..comparison:exception	–	–	–	0.56*
expr..comparison:expr..bit	–	1.19	–	–
expr..comparison:import	1.17*	NA	NA	–
expr..comparison:qualifier	–	–	–	1.42
expr..comparison:sync	**1.60**	–	–	–
expr..comparison:obj	–	1.38*	–	–
expr..field:call	–	**0.77**	–	–
expr..field:decl	–	–	**0.35**	–
ctrl:exception	**0.75**	–	–	–
decl:call	–	–	**2.92**	–
import:exception	**0.80**	NA	NA	–
obj:call	**1.29**	–	–	–
obj:decl	1.32	–	–	–
obj:exception	1.20*	–	–	–
qualifier:call	–	**0.69**	–	–
sync:ctrl	–	NA	NA	**0.30**

an effect on the bug survival time. Such an effect is significant for Eclipse and Mozilla, although in the first case the HR is (slightly) greater than one (1.13) and in the second case is lower than one (0.83). The HR for Vuze is higher than one, though the effect is not statistically significant. Thus, bugs involving array access have lower HR for Mozilla (C/C++) than for the Java systems; this could be due to the fact that array-related bugs could be more difficult to find in C/C++ rather than in Java, e.g., because Java does boundary checking.

- *Arithmetic expressions* are typical bug fixings, and have a statistically significant effect for Mozilla and Vuze, with HRs greater than one in both cases. Indeed, making a mistake (even a small one) in arithmetic expressions is quite frequent, as shown in Figure 2. Despite being quite frequent, as indicated in the first line of Table III, bugs involving arithmetic expressions seem to remain in the system for a shorter time than other bugs.
- *Object instantiation* has a significant effect for Eclipse and Mozilla, however in the first case the HR is ~ 1, while it is greater than one (1.36) in the second case.
- *Synchronization* primitives have a high (1.63) and marginally significant HR for Vuze. Vuze is a system that makes a massive usage of network connections, thus synchronization is more important than for Eclipse. Note that synchronization primitives were considered for Java systems only, as for C/C++ these are not part of the language, but handled by specific system calls instead.

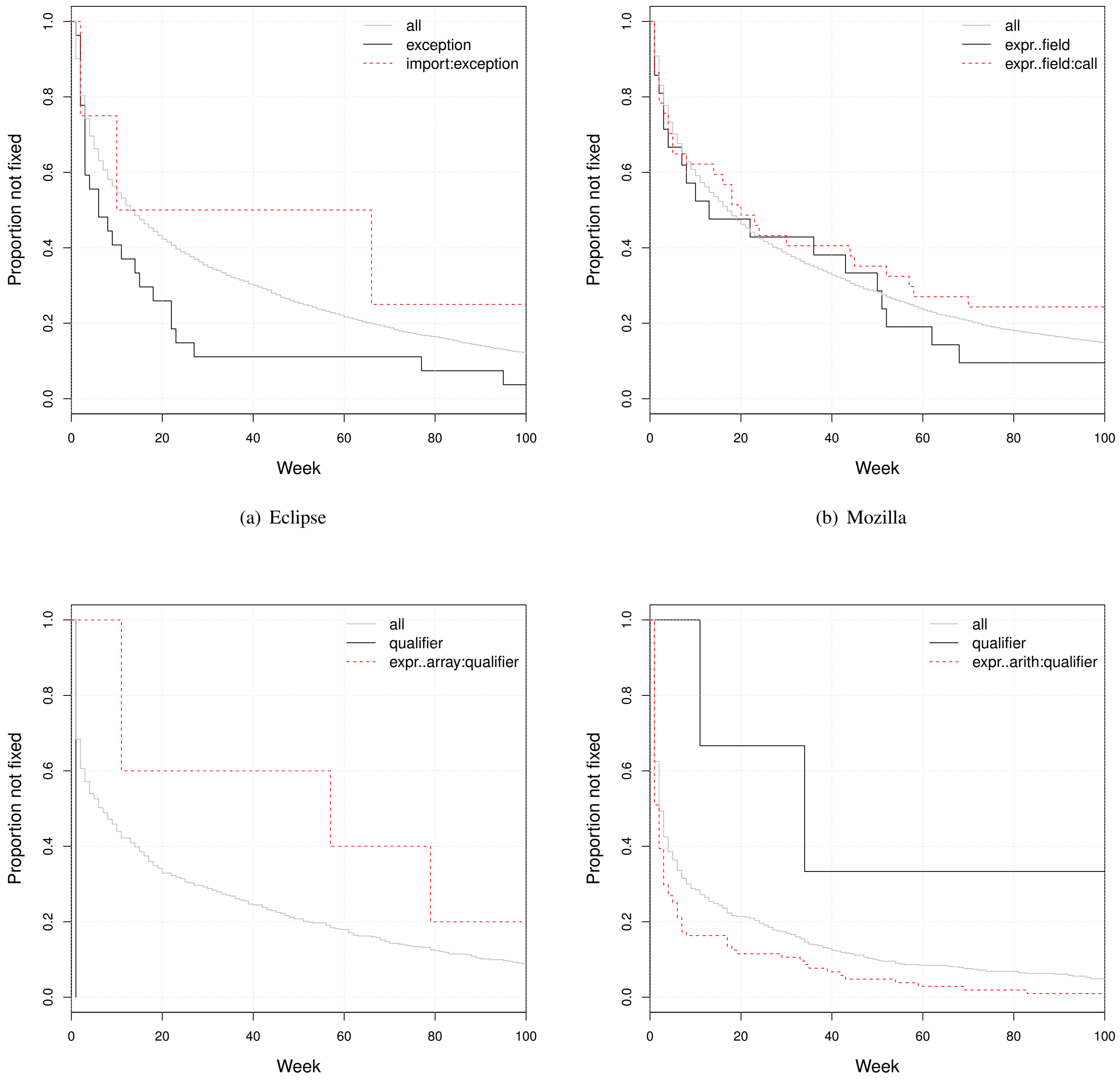
B. RQ2: Is there any significant interaction between code constructs with respect to their effect on the risk of a bug to remain in the system for a long time?

Table IV reports HRs of factors and factors interactions that were chosen by the AIC procedure. Also in this case significant factors have HRs highlighted in boldface, and marginally significant ones are highlighted with a "*". The selected Cox model successfully passed the likelihood ratio test, the Wald test, the logrank test (p-value < 0.01 in all cases), and the proportional hazard assumption is verified for all factors.

As it can be noticed from the top part of Table IV, there are factors that were not considered in the models without interaction (or were not significant), while they are included in the models with interaction. This is for example the case of bit manipulation expressions (though significant only for Vuze) and Boolean expressions (though not significant), of exceptions (selected and significant for both Java systems, Eclipse and Vuze). Specifically, it can be noticed that exceptions exhibit a high HR, suggesting that changing exception handling well characterizes bugs being fixed in a short time: this is quite intuitive as an improper exception handling is a kind of bug quite usual in many object-oriented systems, and in many cases (though not always) solving it could be straight-forward.

When looking at interactions, we can notice that many of them concern either combinations of different kinds of expressions, or combinations of expressions with other code constructs. Moreover, the HR of an explanatory variable may exhibit an opposite ratio when considered in association with another explanatory variable. This leads to an increment or a decrement in the survival time of a bug, as shown in the survival plots of Figure 3. The figure shows the proportion of not-yet fixed bugs involving a certain code construct, if compared with the overall proportion of "all" bugs. For example:

- *Bit-expressions and control-flow constructs* significantly interact for both C/C++ systems and exhibit an HR < 1, i.e., 0.79 for Mozilla and 0.56 for OpenLDAP. The usage of bit operators in control flow constructs is, indeed, something that could be difficult to understand and thus makes bug fixing difficult.
- *Qualifiers and other code constructs* in OpenLDAP and Vuze. Changes in qualifiers exhibit a significant HR both



(a) Eclipse

(b) Mozilla

(c) OpenLDAP

(d) Vuze

Fig. 3. Survival curves showing the effect of variable interactions.

in OpenLDAP (HR=2.48) and Vuze (HR=0.48). It is interesting to note that the HR is completely inverted when the qualifier variable interacts with changes in expressions, i.e., array expressions in OpenLDAP (HR=0.09) and arithmetic expressions in Vuze (HR=2.46). This suggests that, when expressions are changed, bugs survive longer in OpenLDAP than in Vuze (Figures 3(c) and 3(d)). In particular, for OpenLDAP it can be noticed that bugs related to qualifiers only were all removed within 1 week, while those also relating expressions have a longer survival time. One reason for such a difference is that, while for Java programs changes in qualifiers mainly deal with visibility (*public*, *protected*, *private*), in C qualifiers related to declaring a variable as *extern* or *static*, and declaring a static variable where not appropriate could induce errors not easy to fix.

- *Synchronization and control-flow* in Vuze: although we have previously discussed how synchronization constructs exhibit a high HR, as they constitute an important problem for Vuze and thus have to be fixed quite quickly, their interaction with control-flow constructs could either make the bug unlikely to manifest, or make it hard to fix.

- *Comparison expression and synchronization* in Eclipse: intuitively this fix looks relatively similar to the previous one, as comparison expression are often used within control flow constructs, in this case the HR is higher than one. A possible interpretation could be that synchronizations could be more difficult to be performed in a networking system (Vuze) rather than in a IDE (Eclipse).
- *Interaction of exception handling with other code constructs:* as we have explained above, exception handling *per se* represents a construct correlated with a high HR, i.e., quick bug fixing. However, when such a construct interacts with other constructs, HR would change. For example, in Vuze there is a marginal interaction with *comparison expressions* (HR=0.56), in Eclipse a significant interaction with control-flow constructs (HR=0.75), imports (HR=0.80, see Figure 3(a)), and a marginally significant interaction with object instantiation/access (HR=1.20). Besides the last cases, when the fixing of an exception also requires handling control flow or conditionals, the bug could become harder to find or to fix. A similar situation could occur when, other than adding a *catch*, it could be necessary to *import* the right exception that should be caught or thrown.
- *Field access operators:* although alone (see the top-side of Table IV) they are included in Mozilla and OpenLDAP and exhibit a HR > 1, when they interact with other constructs (in Mozilla and OpenLDAP) HR tend to become lower than one. This is the case, for instance, of function calls in Mozilla (HR=0.77, see Figure 3(a)), bit operators in Mozilla (HR=0.86, though not significant), and declarations in OpenLDAP (HR=0.35). Thus, especially in C code, the access to data structures increases the bug survival time, if changes to such an access are combined with changes to declarations (e.g., of the structure itself) or with the usage of the field access within a function call.

C. Examples

This section aims at providing a qualitative discussion of some exemplar bugs among those we examined, for which we found, by reading the versioning system commit notes and/or the bug reports, information about the nature of the bug fixing and we were able to relate it with the code constructs being changed and to find an interpretation of why it was likely that the bug survived in the system more or less than other bugs.

In Eclipse, bug #90084[6] was likely introduced by a change occurred on 2004-06-03, and then fixed on 2005-04-04, by changing an exception handler and importing the proper exception, i.e., interaction of code constructs *import::exception*, which for Eclipse shows a HR=0.80, thus a longer time to fix than other bugs. Comments added to the bug were: (i) *Frederic Fusier 2005-04-02 10:30:57 EST - "Created attachment 19481 [details] Trace of 2 ResourceException"*, (ii) *John Arthorne 2005-04-04 11:11:18 EDT - "Agreed, this is a bug in ProjectPreferences. The entire block in comment #3 needs to be inside an IWorkspaceRunnable with appropriate scheduling rule to prevent another thread from creating/deleting the resource after the IResource.exists() check is performed."*

In Mozilla, bug #288357[7] related to the interaction of expression field access ("$->$") and method invocations (i.e,, *expr..field:call*, HR=0.77, longer survival time than others) likely introduced on 2004-03-03, for which a first patch was submitted on 2005-05-20, with the following comment: *"Yeah. I'm not sure if I should check it in. It's a bit risky, fixes a bug that won't come up much at all, and isn't really the right way to do absolute positioning in columns."*. The concern in the comment was actually right, as after almost another year, on 2006-04-17, the following comment (with patch) was posted on Bugzilla: *"This patch is a better fix. It forces the abs-pos container to always be the first in flow, making block and inline containers consistent. The block code currently doesn't move abs-pos children across block continuations so they just stay there in the first-in-flow and everyone's happy"*, and three days after the change was finally committed.

In OpenLDAP, bug #3499[8] was likely introduced 2004-07-19 and then fixed on 2005-01-20, by modifying declarations, control statements, field and bit expressions. For such a system, the interactions *expr..field:decl* and *expr..bit:ctrl* (HR=0.56) exhibit a HR of 0.35 and 0.56 respectively, indicating a survival time longer than other bugs. The commit note posted by *ando* said: *"the attribute mapping features of rwm seem to be very broken. Here few issues related to ITS#3499 are fixed but there's some work to do yet"*. Basically, developers were fixing problems only partially solved in the past, thus the bug remained in the system for a long time.

In Vuze, there was a case in which the bug was introduced on 2008-02-12, and fixed only 24 hours after, because developers found that the code did not work in a particular environment for which it was targeted. The bug concerned arithmetic expressions and qualifiers (HR=2.46, i.e., shorter survival time than others), and the commit note by *khai_m_nguyen* says *"Remove the use of Constants. URL_PREFIX_NO_PORT for the LightBoxBrowserWindow because it doesn't work on dev environment when the server is local and set to 8080. The bug that URL_PREFIX_NO_PORT was meant to fix will now have to be fixed in the AJAX instead"*.

VI. Threats to Validity

This section discusses the main threats to the validity of our study.

Construct validity threats concern the relationship between theory and observation. Such threats are mainly related to the reliability of the measurements on which our study is based on. The first assumption is on the bug survival time. The approach we adopted is inspired to the work of Kim *et al.* [2], [3] (relying on CVS annotations and other heuristics), can only provide a set of bug introducing changes, and only

[6] https://bugs.eclipse.org/bugs/show_bug.cgi?id=90084

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=288357

[8] http://www.openldap.org/its/index.cgi/Archive.Software%20Bugs?id=3499;selectid=3499;usearchives=1;statetype=-1

relies on the fact that a line changed in a bug fixing was last modified in a given file revision. CVS does not tell whether the bug fixing changed a code construct actually introduced in that revision. Also, since a fixing originates in more bug introducing changes—i.e., each line modified in the fixing could have been last modified in a different file revision—we choose the time and date of the most recent among these changes as the bug-introducing timestamp. The other source of imprecision is related to the token-based approach for capturing changed code constructs by analyzing the source code file before and after the bug fixing. Although we have manually analyzed a set of changes to check the correctness of the changed code construct extraction, we cannot exclude imprecisions in our data set.

Conclusion validity concerns the relationship between the treatment and the outcome. First, we have chosen a survival model (Cox proportional hazard model) that does not require a fitting with any specific distribution. Also, we discuss the effect of each fixed code construct, taking into account its statistical significance in the overall model comprising all code constructs selected using the AIC procedure. Other that considering the statistical significance of each construct, we report and discuss an effect size measure, i.e., the HR: this is even more important than reporting statistical significance, as the HR shows whether the construct has an effect towards a survival time longer or shorter than other bugs.

Threats to *internal validity* concern factors that can influence our observations. We observe a relation between the survival time and a factor that could be one of its symptoms, i.e., the code constructs being fixed. However, this cannot lead us to claim any cause-effect relationship. As said in the introduction, there are other—sometimes more important—factors that can affect the bug survival time, e.g., the impact such a bug has on system behavior, how frequently does the bug manifests itself, etc. Nevertheless, we provide some qualitative explanation supporting the fact that the investigated code constructs can, at least, play a role together with other factors. It should also be clear that what we observe is the time between the bug introduction and its fixing, while we do not observe the discussion length in the bug tracking system. This is out of scope of this paper and will be investigated in future work.

Threats to *external validity* concern the generalization of our findings. Although we performed our analyses on four different systems, belonging to different domains and developed with different programming languages, we are aware that a further empirical validation on a larger set of systems would be beneficial to better support our findings. Also, the code constructs we considered are only a limited set of symptoms for bugs with a long or short survival time. Further studies need to consider other variables related to source code—e.g., code complexity, data flow-related features—as well as to human factors.

VII. Related Work

As mentioned in the introduction, Kim *et al.* [10], [2] propose an approach to identify bug-introducing changes. The approach relies on CVS annotations and on other heuristics, such as focusing on annotations related to code lines and not comments or blank lines, to reduce the number of false positives. Then Kim *et al.* [3], relying on the bug-introducing change detection approach, use machine learning techniques such as Support Vector Machines (SVM) to predict when a change was likely introducing bugs. In their model, the independent variables are a series of code constructs involved by bug-introducing changes, and the dependent variable is a software artifact (e.g., a class) fault-proneness. While we share with them the usage of the approach to identify bug-introducing changes and the extraction of source code change features, we try to correlate changes to the bug survival rather than to the bug occurrence. That is, we believe that our approach is complementary to what Kim *et al.* proposed, i.e., once predicted that an artifact could be buggy, it could be useful to understand how long would the bug survive in the system.

Weiss *et al.* [11] propose an approach aimed at predicting the time needed by developers to fix a bug once the bug report has been opened. The approach is based on the textual analysis of bug reports and on the usage of K-Nearest Neighbor (KNN) clustering to find similar bugs and use their fixing time for prediction purposes. While we share with them the effort toward analyzing the bug fixing time, in our case (i) we analyze the bug survival since its introduction rather than since its opening on the bug-tracking system, and (ii) we rely on code change features rather than on bug report text.

To the best of our knowledge, the only (preliminary) study aimed at analyzing the time to fix a bug since its introduction was performed by Kim *et al.* [12], who report the distribution of bug-fixing times for ArgoUML and PostgreSQL, as well as information about the top bug-fixing times.

The Cox proportional hazard model has been used by Wendel *et al.* [13] to study the occurrence of bugs in Eclipse; while we share with them the use of the same survival model, our intent is different as we aim at modeling the bug survival time rather than the bug occurrence.

Other related studies deal with defect prediction and prevention. Mockus *et. al.* [14] use a fine-grained analysis to predict defect correction effort and the time-interval for which such an effort is needed. Several researchers analyze the statistical distribution of the bug occurrence in a software system [15], [16], [17], finding that Weibull and exponential distributions capture defect-occurrence behavior across a wide range of systems. Calzolari *et al.* [18] use the predator prey model borrowed from ecological dynamic system to model maintenance and testing effort. They find that, when programmers start to correct code defects, the effort spent to find new defects has an initial increase, followed by a decrease when almost all defects are removed.

Kim and Ernst [19] propose an algorithm to prioritize the fixing of warnings detected by tools such as *FindBugs*, *Jlint* and *PMD*. They focus on warnings removed by bug-fixing directly affecting source code lines containing the warning itself. They find that this represents a very small percentage of warning/vulnerability removal. Di Penta *et al.* [20] investigate the decay of statically detected (using tools such as *Splint*, *Rats*, or *Pixy*) vulnerabilities in three networking systems, Samba, Squid, and Horde. They found that vulnerability decay follows Weibull or exponential distributions, and that some vulnerabilities more important for the particular kind of application—e.g., command injections for Web-based systems, or buffer-overflows for other networking applications—tend to be removed before others. As for warnings and vulnerabilities, bug survival can be modeled by proper distributions; however, we found that parametric models based on Weibull or exponential distributions are not appropriate, while our results show the applicability of the Cox proportional hazard model.

VIII. Conclusion and Work-in-Progress

This paper reported a study, aimed at characterizing bug survival time—since their introduction until their removal—with respect to source code constructs introduced during bug-fixing. The study has been conducted on a sample of bugs extracted from four open source systems, Eclipse, Mozilla, OpenLDAP and Vuze. We modeled—using the Cox proportional hazard model—the relation between the bug survival time and a series of independent variables, i.e., changed code constructs and their interaction. Then, we identified constructs and their interactions exhibiting an hazard ratio (HR) higher (or lower) than one. Bugs concerning such constructs would have a lower (higher) survival time than the average. We also performed an assessment to understand to what extent could survival models be used to predict the likelihood that a bug survives for more than a given time frame.

Results indicate that there are constructs, and above all, interactions between specific constructs, that are significantly correlated with the survival time and lead to HR higher (lower) than one when the constructs are involved alone in a bug fix, while lead to HR lower (higher) than one when involved together. Specifically, we found examples of constructs—e.g., exception handling constructs—that alone correlate with a low survival time, while if changed together with other constructs (such as control-flow statements) correlate with high survival times, as the change could be less obvious to be performed, or the bug might manifest itself only under certain conditions.

Findings reported in this paper can be useful to understand what kinds of code constructs can be related to bugs that will remain in the system for a long time, thus suggesting developers to better inspect or test the source code containing such constructs, as well as to the quality control team to better verify patches before these will be committed in the system.

There are several directions for future work. First, we plan to use a more sophisticated source code analysis to identify changes in code constructs—not considered in this study, e.g., constructs related to the program data-flow or complexity—that, we suppose, can well describe a bug survival. Also, we plan to investigate the effect of other factors—e.g. human factors—on bug survival time. Last, but not least, we plan to extend the empirical study on a larger set of bugs and systems.

References

[1] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897–910, 2005.

[2] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 Sept. 2006, Tokyo, Japan*. IEEE CS, 2006, pp. 81–90.

[3] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 181–196, 2008.

[4] D. Cox, "Regression models and life-table," *Journal of Royal Statistical Society*, vol. 34, pp. 187–220, 1972.

[5] E. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.

[6] D. Kleinbaum and M. Klein, *Survival Analysis: A Self-Learning Text*. Springer, 1997.

[7] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 563–572.

[8] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *19th International Conference on Software Maintenance (ICSM 2003), 22-26 September 2003, Amsterdam, The Netherlands*, 2003, pp. 23–.

[9] H. Akaike, "A new look at the statistical model identification," *Automatic Control, IEEE Transactions on*, vol. 19, no. 6, pp. 716 – 723, Dec. 1974.

[10] S. Kim and E. J. Whitehead, "How long did it take to fix bugs?" in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*. ACM, 2006, pp. 173–174.

[11] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*. IEEE CS, 2007, p. #1.

[12] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 2006 international workshop on Mining software repositories (MSR '06)*. ACM, 2006, pp. 173–174.

[13] M. Wedel, U. Jensen, and P. Göhner, "Mining software code repositories and bug databases using survival analysis models," in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement (ESEM '08)*. ACM, 2008, pp. 282–284.

[14] A. Mockus, D. M. Weiss, and P. Zhang, "Understanding and predicting effort in software projects," in *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA: IEEE CS, 2003, pp. 274–284.

[15] W. Jones, "Reliability models for very large software systems in industry," in *International Symposium on Software Reliability Engineering*, 1991, pp. 35–42.

[16] P. Luo Li, M. Shaw, J. D. Herbsleb, B. K. Ray, and P. Santhanam, "Empirical evaluation of defect projection models for widely-deployed production software systems," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*. ACM, 2004, pp. 263–272.

[17] A. Wood, "Predicting software reliability," *IEEE Computer*, vol. 9, pp. 69–77, 1999.

[18] F. Calzolari, P. Tonella, and G. Antoniol, "Maintenance and testing effort modeled by linear and nonlinear dynamic systems," *Information & Software Technology*, vol. 43, no. 8, pp. 477–486, 2001.

[19] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, 2007, pp. 45–54.

[20] M. Di Penta, L. Cerulo, and L. Aversano, "The life and death of statically detected vulnerabilities: An empirical study," *Information & Software Technology*, vol. 51, no. 10, pp. 1469–1484, 2009.