Uncovering Causal Relationships between Software Metrics and Bugs

Cesar Couto, Christofer Silva
Department of Computing
CEFET-MG, Brazil
{*cesar,christofer*}*@decom.cefetmg.br*

Marco Tulio Valente, Roberto Bigonha
Department of Computer Science
UFMG, Brazil
{*mtov,bigonha*}*@dcc.ufmg.br*

Nicolas Anquetil
RMoD Team
INRIA, Lille, France
nicolas.anquetil@inria.fr

***Abstract*—Bug prediction is an important challenge for software engineering research. It consist in looking for possible early indicators of the presence of bugs in a software. However, despite the relevance of the issue, most experiments designed to evaluate bug prediction only investigate whether there is a linear relation between the predictor and the presence of bugs. However, it is well known that standard regression models cannot filter out spurious relations. Therefore, in this paper we describe an experiment to discover more robust evidences towards causality between software metrics (as predictors) and the occurrence of bugs. For this purpose, we have relied on Granger Causality Test to evaluate whether past changes in a given time series are useful to forecast changes in another series. As its name suggests, Granger Test is a better indication of causality between two variables. We present and discuss the results of experiments on four real world systems evaluated over a time frame of almost four years. Particularly, we have been able to discover in the history of metrics the causes – in the terms of the Granger Test – for 64% to 93% of the defects reported for the systems considered in our experiment.**

***Keywords*-Bug Prediction; Causality; Software Metrics; Granger Test**

I. Introduction

Bug prediction is an important challenge for software engineering research [3], [9], [25]. The goal is to build reliable predictors that can indicate in advance those components of a software system that are more likely to fail. The availability of this information is of central value to most software quality assurance procedures. For example, it allows quality managers to allocate more time and resources to test — or even to redesign and reimplement — those components predicted as defect-prone.

Due to its relevance to software quality, various bug prediction techniques have already been proposed. Essentially, such techniques rely on different predictors, including source code metrics (e.g. coupling, cohesion, size) [3], [22], [24], change metrics [14], static analysis tools [2], [6], [21], and code smells [7]. However, the typical experiments designed to evaluate bug prediction techniques usually do not investigate whether the discovered relationships indicate cause-effect relations or whether they are mere *statistical coincidences*. More specifically, it is well known that regression models – the most common statistical technique used by bug predictors – cannot filter out spurious relations [11]. In other words, events that represent mere coincidences can undermine the predictions performed by standard regression models, especially when the proposed models are applied to systems maintained during years or decades.

Therefore, in this paper we describe an experiment to discover more robust evidences towards causality between software metrics (as predictors) and the occurrence of bugs. For this purpose, we have relied on a statistical hypothesis test proposed by Clive Granger to evaluate whether past changes in a given time series are useful to forecast changes in another series. Granger Test has been originally proposed to evaluate causality between time series of economic data (e.g. to show whether changes in oil prices cause recession) [12], [13]. Although extensively used by econometricians, the test has already been applied in bioinformatics (to identify gene regulatory relationships [20]) and recently in software maintenance (to detect change couplings that are spread over an interval of time [4]).

The experiment described in this paper relies on a public dataset constructed by D'Ambros *et al.* to evaluate bug prediction techniques [8], [9]. This dataset provides bi-weekly time series for seventeen object-oriented metrics, over a period of almost four years, for four real-world Java systems. The contributions of our work are: (a) an extension of D'Ambros dataset with a new time series including the mapping of 5,028 bugs reported for the considered systems to their respective classes; (b) a methodology to systematically mine for Granger-causality relationships between software quality metrics and defects at the class-level; and (c) a report on the results and lessons learned after using this methodology to mine for causalities between D'Ambros time series (software metrics) and our new time series (defects). Particularly, we have been able to discover in the history of metrics the causes – in the terms of the Granger Test – for 64% to 93% of the defects reported for the systems considered in our experiment. Moreover, for each defective class we have been able to identify the particular metrics that have Granger-caused the reported defects.

The paper is organized as follows. We start with an overview of Granger Causality (Section II). Next, we describe the datasets and the methodology followed in the experiments reported in the paper (Section III). Section IV reports the results and lessons learned after the experiments described in the previous section. Section V and VI discuss threats to validity and related work, respectively. Section VII presents our contributions and briefly outlines future work.

II. Granger Causality

In this subsection, we first describe a precondition that Granger requires the time series to follow (Section II-A). Next, we discuss the test (Section II-B).

A. Stationary Time Series

An usual pre-condition when applying forecasting techniques – including the Granger Test described in the next subsection – is to require a stationary behavior from the time series [11]. In stationary time series, properties such as mean and variance are constant over time. Stated otherwise, a stationary behavior does not mean the values are constant, but that they fluctuate around a constant long run mean and variance. However, most time series of software metrics and defects when expressed in their original units of measurements are not stationary. The reason is intuitively explained by Lehman's Law of software evolution, which states that software measures of complexity and size tend to grow continuously [19]. This behavior is also common in the original domain of Granger application, because time series of prices, inflation, gross domestic product, etc also tend to grow along time [13].

When the time series are not stationary, a common workaround is to consider not the absolute values of the series, but their differences from one period to the next. For example, suppose a time series $x(t)$. Its *first difference* $x'(t)$ is defined as $x'(t) = x(t) - x(t-1)$.

Example #1: To illustrate the notion of stationarity behavior, we will rely on a time series that measures the number of methods (NOM), extracted for the Eclipse JDT Core system, in intervals of bi-weeks, from 2005 to 2008 [9]. Figure 1 illustrates the behavior of this series. As we can observe, the series is not stationary, since it has a clear growth trend, with some disruptions along the way.

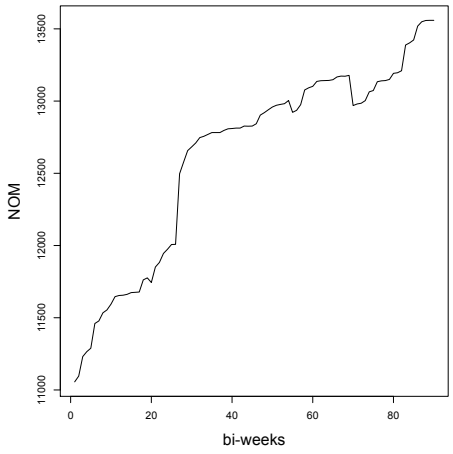

Figure 1. Original NOM series (non-stationary behavior)

Figure 2 shows the first difference of NOM. We can observe that most values are delimited by a constant mean and variance. Therefore, NOM in first difference has a stationary behavior.

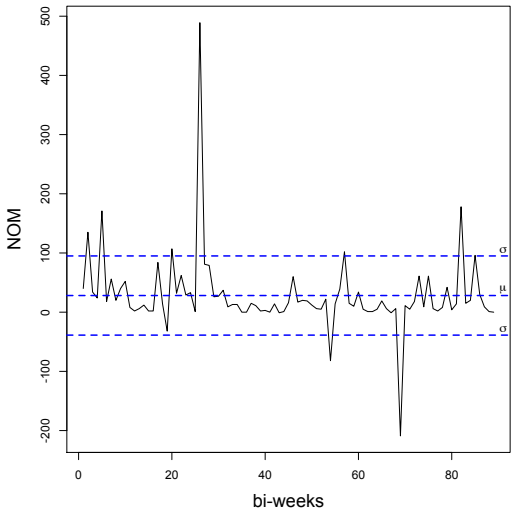

Figure 2. NOM series in first difference (stationary behavior)

B. Granger Test

Testing causality between two stationary time series x and y, according to Granger, involves using a statistical test – usually the F-Test – to check whether x helps to predict y at some stage in the future [12]. If this happens, we can conclude that x Granger-causes y. The most common implementation of Granger's Causality Test uses bivariate and univariate auto-regressive models. A bivariate auto-regressive model includes values both from the independent variable x and from the dependent variable y. On the other hand, a univariate auto-regressive model considers only lagged values of the variable y.

To apply the Granger test, we must first calculate the following bivariate auto-regressive model [4]:

$$\begin{aligned} y_t = c_1 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \cdots + \alpha_p y_{t-p} \\ +\beta_1 x_{t-1} + \beta_2 x_{t-2} + \cdots + \beta_p x_{t-p} + u_t \end{aligned} \quad (1)$$

where p is the auto-regressive lag length (an input parameter of the test). Essentially, this value defines the number of past values – from both x and y – that will be considered in the regressive models. Furthermore, Equation 1 defines a bivariate model because it uses values of x and y, limited by the lag p.

To test whether x Granger-cause y, the following null hypothesis must be rejected:

$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

This hypothesis assumes that the lagged values of x do not add predictive power to the regression. In other words, by testing whether the β coefficients can be equal to zero, the goal is to discard the possibility that the lagged values of x can contribute to the prediction.

To reject the null hypothesis, we must first estimate the following auto-regressive univariate model (i.e., an equation similar to 1 but excluding the values of x):

$$y_t = c_1 + \gamma_1 y_{t-1} + \gamma_2 y_{t-2} + \cdots + \gamma_p y_{t-p} + e_t \quad (2)$$

Finally, to evaluate the precision of both models, we must calculate their residual sum of squares (RSS):

$$RSS_1 = \sum_{t=1}^{T} \hat{u}_t^2 \qquad RSS_0 = \sum_{t=1}^{T} \hat{e}_t^2$$

If the following test

$$S_1 = \frac{(RSS_0 - RSS_1)/p}{RSS_1/(T-2p-1)} \sim F_p, T-2p-1$$

exceeds the critical value of F with a significance level of 5% for the distribution $F(p, T-2p-1)$, the bivariate auto-regressive model is better (in terms of residuals) than the univariate model. In case the null hypothesis is rejected we can conclude that x causes y, in the terms of the Granger test.

Example #2: For our previous Eclipse JDT Core example, we have applied Granger to evaluate whether the number of public methods (NOPM), in the Granger sense, causes NOM. Although the common intuition suggests this relation truly denotes causality, it is not captured by Granger's test. Particularly, assuming $p = 1$ (the lag parameter), the F-test has returned a *p-value* of 0.15, which is superior to the defined threshold of 5%. To explain the lack of Granger-causality, we have to consider that variations in the number of public methods cause an immediate impact on the total number of methods (public, private etc) of the system. Therefore, Granger's application is recommended in scenarios where variations in the independent variable are reflected in the dependent variable only after a certain delay (or lag).

Example #3: To explain the sense of causality captured by Granger in a simple and comprehensive manner, suppose a new time series defined as:

$$NOM'(t) = \begin{cases} NOM(t) & \text{if } t \leq 5 \\ NOM(t-5) & \text{if } t > 5 \end{cases}$$

Basically, NOM' reflects with a lag of five bi-weeks the values of NOM. We have reapplied Granger to evaluate whether NOPM, in the Granger sense, causes NOM' and the result has been positive, assuming $p = 5$. Therefore, knowing the NOPM values at a given bi-week helps to predict the value of NOM'.

Figure 3 illustrates the behavior of both series. For example, we can observe that just before bi-week 30 a significant increase has occurred in the number of public methods. By knowing this information, one could predict an important increase in NOM' in the following bi-weeks. In fact, the figure shows that the mentioned increase in NOPM has been propagated to NOM' in few bi-weeks (we have circulated these events in the presented series).

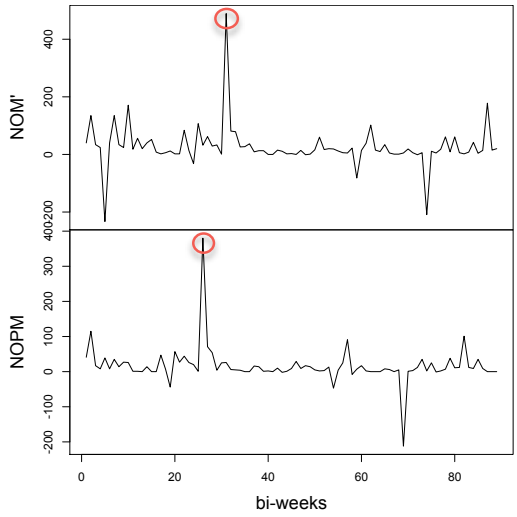

Figure 3. NOPM and NOM' time series. The increase in NOPM values just before bi-week 30 has been propagated to NOM' few weeks later

III. Study Setup

This sections starts by describing the original dataset used in the experiment (Section III-A) and move to describe our extension of this dataset with a temporal series of defects (Section III-B). The section concludes by describing the methodology followed in the proposed experiment (Section III-C).

A. Original Dataset

The study reported on this paper has been based on a dataset made public by D'Ambros *et al.* to evaluate bug prediction techniques [8], [9]. Basically, the provided dataset includes temporal series for seventeen source code metrics, collected at the class-level for the Java-based systems described in Table I[1]. In this table, column Period informs the time interval in which the metrics were collected. On total, the dataset has 4,298 classes, each of them with at least 90 bi-weekly versions (which is equivalent to around three and a half years of the lifetime of the considered systems).

Table I
Systems in the original dataset

System	Period	Classes	Versions
Eclipse JDT Core	1-1-2005 - 5-31-2008	1041	90
Eclipse PDE UI	1-1-2005 - 9-6-2008	1924	97
Equinox	1-1-2005 - 6-14-2008	444	91
Lucene	1-1-2005 - 10-4-2008	889	99

Table II shows the metrics included in the original dataset. It considers six metrics proposed by Chidamber and Kemerer [5] and other eleven metrics, such as lines of code, number of public methods, fan-in, fan-out etc. For each class of the mentioned systems, the dataset provides the values of these metrics in intervals of bi-weeks.

[1]The original dataset includes a fifth system (Mylyn). However, we have not considered this system because the dataset includes information for only 47 bi-weeks of its evolution.

Table II
METRICS INCLUDED IN THE ORIGINAL DATASET

	Metrics	Description
1	WMC	Weighted methods per class
2	DIT	Depth of inheritance tree
3	RFC	Request for class
4	NOC	Number of children
5	CBO	Coupling between object class
6	LCOM	Lack of cohesion in methods
7	FANIN	Number of classes that reference the class
8	FANOUT	Number of classes referenced by the class
9	NOA	Number of attributes
10	NOPA	Number of public attributes
11	NOPRA	Number of private attributes
12	NOAI	Number of attributes inherited
13	LOC	Number of lines of code
14	NOM	Number of methods
15	NOPM	Number of public methods
16	NOPRM	Number of private methods
17	NOMI	Number of methods inherited

B. Time Series of Defects

In our terminology, a bug is a failure in the observable behavior of the system. Bugs are caused by one or more errors in the source code, called defects. Particularly, we counted defects at the class level (since all the metrics considered in the paper are related to classes). Therefore, each class changed to fix a bug is counted as one defect.

The original dataset only provides information on the total number of defects reported for each class. Thus, to apply Granger it was necessary to distribute this number along the bi-weeks considered in the study. To create a time series of defects, we have initially collected the bugs – or more precisely, the maintenance requests – reported in the bug-tracking platforms of the considered systems. Table III, column B, shows the number of bugs opened via Bugzilla or Jira for each of the systems, at the time interval considered in the study. As can be seen in this table, we have collected a total of 5,028 bugs.

To create the time series of defects, we first linked each bug b – reported in the bug-tracking platforms – to the classes changed to fix b, using the following strategy:

1) Suppose that *Bugs* is the set with the IDs of all bugs reported during the time frame of the experiment.

2) Suppose that *Commits* is the set with the IDs of all commits in the version control platforms. Suppose also that $Cmts[c]$ and $Chg[c]$ are, respectively, the developer's comments and the classes changed in each commit $c \in Commits$.

3) The classes changed to fix bug $b \in Bugs$ are defined as:

$$\bigcup_{\forall c \in Commits} \{\ Chg[c] \mid substr(b, Cmts[c])\ \}$$

The set returned by this expression is the union of the classes changed in each commit c for which the textual comments provided by the developer includes a reference to the bug with ID b. The predicate $substr(s_1, s_2)$ tests whether substring s_1 is a substring of s_2.

Finally, suppose we have discovered that in order to fix bug b changes have been applied to the class C. In this case, a defect associated to b was counted for class C at bi-week t when the following conditions held: (a) b has been opened before the ending date of the bi-week t; (b) b has been fixed after the starting date of the bi-week t.

Table III summarizes the main properties of the extracted time series of defects. The table shows three information: the number of bugs we have initially collected in the study (column B), the number of defects that caused such bugs (column D), and the average number of defects per bugs (column D/B). As can be observed, on average each bug required changes in 2.87 defective classes. Therefore, at least in our experiment, changes to fix bugs have not presented a scattered behavior.

Table III
NUMBER OF BUGS (B), DEFECTS (D), AND DEFECTS PER BUGS (D/B)

System	B	D	D/B
Eclipse JDT Core	2398	7313	3.05
Eclipse PDE UI	1821	5547	3.05
Equinox	545	991	1.82
Lucene	264	564	2.14
Total	5028	14415	2.87

C. Methodology

To apply Granger we have relied on the following procedure:

```
 1: foreach c in Classes
 2:   s1= D[c];
 3:   if d_check(s1)
 4:      for i= 1 to 17 do
 5:        s2= M[i][c];
 6:        if m_check(s2) and
 7:           granger(s2,s1);
 8:        endif
 9:      endfor
10:   endif
11: endforeach
```

In this algorithm, `Classes` is the set of all 4,298 classes considered in the study (line 1) and `D[c]` is the time series with the number of defects in each of these classes (line 2). The algorithm relies on function `d_check` (line 3) to check whether the defects in the time series `s1` attends the following preconditions:

- P1: The time series must have at least 30 values (around 30% of the time series size). Therefore, we have eliminated time series related to classes that only existed for a small proportion of the time frame considered in the experiment – usually called dayfly classes [18]. The motivation for this precondition is that probably such classes do not have a long history of defects that qualify their use in predictions.

- P2: Some values in the time series must be different of zero. Basically, the goal is to discard classes that have never presented a defect in their lifetime (probably, because they implement a simple and stable requirement). The motivation for this precondition is that it is straightforward to predict defects for such classes: probably, most of them they will remain with zero defects in the future.

- P3: The time series must be stationary, which is a required precondition to apply Granger, as described in Section II-A. To identify stationary time series we relied on function *adf.test()* from the R statistical system (package *tseries*). This function implements the Augmented Dickey-Fuller test for stationary behavior [11].

Suppose that a given class `c` has passed the previous preconditions. For such classes, suppose also that `M[i][c]` (line 5) is the time series with the bi-weekly variations in the values of the `i`-th metric considered in the experiment, $1 \leq i \leq 17$. The algorithm relies on function `m_check` (line 6) to test whether the time series `s2` – with the series of metrics values – attends the following preconditions:

- P4: The time series must not be constant. We have discovered that for some classes the values of the metrics have never changed during the whole time frame considered in the experiment. Therefore, we decided to discard such series, since variations in the independent variables are the key event to observe when evaluating Granger causality.

- P5: The time series must be stationary, i.e. as defined for the time series of defects, we have discarded the series where the values of the metrics do not fluctuate around a long run mean.

Finally, for series `s2` (metrics) and `s1` (defects) that passed preconditions P1 to P5, function `granger(s2,s1)` calls the Granger test to check whether `s2` Granger-causes `s1` (line 7). In practice, to apply the test we have used function *granger.test()* provided by the *msbvar* package of the R system. The tests were calculated using a significance level of 95% ($\alpha = 0.05$) and the lag ranging from 1 to 4. We counted as causality the calls where the variable *p-value* obtained by applying the F-test is less than or equal to α, i.e., when *p-value* ≤ 0.05.

IV. Study Results

This section reports and discusses the results and lessons learned after the experiment described in Section III.

A. How many time series of defects have passed the defined preconditions?

For each system, Table IV shows three pairs of values, representing respectively the percentage of classes that survived the preconditions P1, P2, and P3 (defined in Section III-C).

Table IV
Percentage of classes conforming successively to preconditions P1, P2, and P3

System	P1	P2	P3
Eclipse JDT Core	92	71	68
Eclipse PDE UI	73	55	47
Equinox	60	38	36
Lucene	73	20	19
Total	77	50	46

We have observed that 77% of the classes have survived preconditions P1 (more than 30 values) and that 50% of the classes have survived both P1 and P2 (at least one defect in their lifetime). In other words, half of the classes have either a short lifetime (which affects their power to provide reliable predictions) or have never been changed to fix bugs. Finally, our sample has been reduced to 46% of the classes after applying the last precondition (test for stationary behavior). Therefore, even considering the series in first differences, some of them have presented a non-stationary behavior.

Lesson Learned #1: To mine the causes of bugs, it is fundamental to remove classes with a short lifetime (that may not provide reliable predictions), classes with zero defects (that make the predictions trivial), and classes with a non-stationary time series of defects (that may statistically invalidate the findings). In the described experiment, our sample has been reduced to 46% of its original size after applying these preconditions.

B. How many defects still exist in the classes that have passed the defined preconditions?

For each system, Table V shows three information: the number of bugs we initially collected in the study (column B), the number of defects at the class level that caused such bugs (column D), and the number of defects detected in the classes that passed the preconditions P1 to P3 (column DVC). The table also shows the percentage of valid defects, i.e. the percentage of defects after removing the series that have not survived preconditions P1, P2, and P3 (column DVC/D).

Table V
Number of bugs (B), defects (D), and defects in valid classes (DVC)

System	B	D	DVC	DVC/D
Eclipse JDT Core	2398	7313	7057	0.96
Eclipse PDE UI	1821	5547	4323	0.78
Equinox	545	991	853	0.86
Lucene	264	564	460	0.82
Total	5028	14415	12693	0.88

The results show that 88% of the defects have been reported in classes that survived preconditions P1, P2, and P3. In other words, by successively applying preconditions P1, P2, and P3 we have eliminated 54% of the classes (as

showed in Table IV), but those classes account for only 12% of the total number of defects considered in the study.

Lesson Learned #2: Other studies in the literature have already showed that most defects are concentrated in few classes [1], [10], [16]. Our experiment reinforces this finding. A more original lesson is the observation that these defects are the most interesting to investigate for the purpose of bug prediction, since they come from classes with long lifetimes and from time series with non-stationary behavior.

C. How many time series of metrics have passed the defined preconditions?

For each system and metric, Table VI shows the percentage of time series that have passed preconditions P4 and P5.

Table VI
PERCENTAGE OF TIME SERIES CONFORMING SUCCESSIVELY TO PRECONDITIONS P4 AND P5

	JDT		PDE		Equinox		Lucene		Total	
	P4	**P5**	**P4**	**P5**	**P4**	**P5**	**P4**	**P5**	**P4**	**P5**
CBO	76	65	94	66	93	76	95	63	88	67
DIT	40	27	66	6	59	6	44	8	54	14
LCOM	58	47	85	50	82	51	78	43	75	49
NOC	10	8	15	6	8	5	27	17	14	8
RFC	80	72	94	69	90	71	90	62	89	70
WMC	76	67	91	62	91	69	87	55	86	64
FANIN	51	39	74	35	77	49	72	53	66	40
FANOUT	63	52	88	58	87	70	80	52	79	57
NOA	45	36	71	40	84	49	59	39	63	40
NOPA	28	22	2	1	29	18	20	14	15	12
NOPRA	25	18	63	36	51	31	50	30	48	29
NOAI	65	60	40	22	28	16	33	13	48	35
LOC	80	72	95	77	92	75	90	62	89	74
NOM	58	47	87	51	84	52	79	48	76	50
NOPM	47	38	80	37	80	44	71	41	68	39
NOPRM	23	17	50	29	70	40	30	20	41	25
NOMI	98	74	78	59	98	80	99	82	89	69

As defined in Section III-C, precondition P4 states that the time series must not be constant. By observing the values in Table VI, we can conclude that constant time series are fairly common for some metrics. For example, for NOC, NOPA, NOPRA, NOAI, and NOPRM more than 50% of the considered classes have presented a constant behavior (column Total). Therefore, the most constant properties of the evaluated classes have been the number of children, the number of attributes (including public, private, and inherited), and the number of private methods. At the other extreme, the number of constant series has been inferior to 15% for CBO, RFC, WMC, LOC, and NOMI.

Table VI also presents the percentage of series that survived precondition P5, which states the series must be stationary. As can be observed, the number of series with non-stationary behavior – even when considering the first differences – is not negligible. For example, for LOC, 89% of the series have survived P4, but only 74% survived P5. Figure 4 presents a non-stationary time series for a Eclipse JDT class. As can be observed, the series has experienced at least four major increases in size, which undermined the role of the mean and variance as representative measurements for its behavior in the long run.

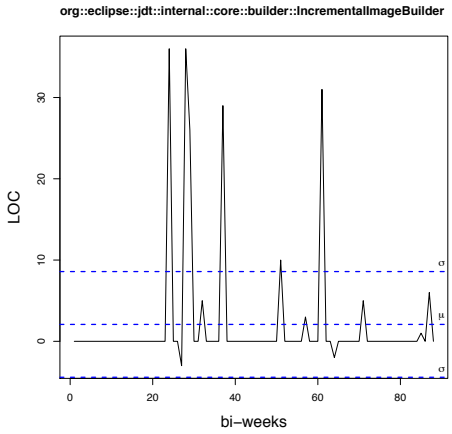

Figure 4. Example of non-stationary time series

Lesson Learned #3: To uncover causal relationships between bugs and software metrics, it is also important to filter out the time series of metrics, removing series with a constant behavior (that do not contribute with valuable predictive power) and with a non-stationary behavior (that statistically invalidates any attempt to perform predictions).

D. How many defects have been anticipated by Granger?

To start answering this question, Table VII shows for each valid class `c` the number of Granger tests with a positive result considering all the series `M[i][c]` ($1 \leq i \leq 17$) and `D[c]`, where `M[i][c]` is one of the seventeen series of metrics for a given class `c` and `D[c]` is the series of defects for this class. For example, for the Eclipse JDT Core in 12% of the classes we have not been able to detect a single causal relation between one of the seventeen series of metrics and the series of defects; in around 13% of the classes Granger has returned a positive result for a single series of metrics, and so on. For the remaining three systems – Eclipse PDE UI, Equinox, and Lucene – the percentage of classes where Granger has not been able to establish a causal connection between metrics and defects has been, respectively, 36%, 47%, and 30%.

The fundamental question is then how many defects have been "predicted" by Granger, i.e. how many defects have been found in the classes where Granger has indicated at least one positive result between the considered metrics and defects. Table VIII shows the results. As can be observed in this table, 84% of the defects have been antecipated by relevant changes in at least one of the series of metrics, according to Granger. The best result has been achieved for the Eclipse JDT

Table VII
PERCENTAGE OF VALID CLASSES WITH n POSITIVE RESULTS FOR GRANGER CAUSALITY

n	JDT Core	PDE UI	Equinox	Lucene
0	12	36	47	30
1	13	14	13	21
2	15	11	13	12
3	9	8	6	7
4	9	5	4	3
5	9	5	3	4
6	8	4	2	5
7	6	4	2	5
8	6	4	2	4
9	4	2	1	3
10	4	3	2	2
11	3	1	1	2
12	1	1	2	2
13	0	0	0	1
14	1	0	1	0
15	0	0	0	0
16	0	0	0	0
17	0	0	0	0
Total	100	100	100	100

Core, where 93% of the defects are related to past changes in the metrics calculated for the changed classes. The worst result was obtained for the Equinox system (64% of coverage by Granger).

Table VIII
NUMBER OF DEFECTS (D), DEFECTS IN VALID CLASSES (DVC), AND DEFECTS PREDICTED BY GRANGER (DPG)

System	D	DVC	DPG	DPG/D
Eclipse JDT Core	7313	7057	6818	0.93
Eclipse PDE UI	5547	4323	4182	0.75
Equinox	991	853	634	0.64
Lucene	564	460	453	0.80
Total	14415	12693	12087	0.84

Lesson Learned #4: By applying the Granger Test, we have been able to discover in the history of metrics the causes for 64% to 93% of the defects reported for the systems considered in our experiment.

E. What are the metrics that have most contributed to predict defects?

For each valid time series of metrics, Table IX shows the percentage of Granger tests that have returned a positive result. For example, the percentage of CBO time series with a Granger-causality with defects was respectively 48%, 38%, 24%, and 39% for Eclipse JDT Core, Eclipse PDE UI, Eclipse Equinox, and Lucene.

As can be observed in this table, the most useful metrics to predict defects in the considered systems have been: RFC (Eclipse JDT Core), NOPA (Eclipse PDE UI), NOPRM (Equinox), and NOPM (Lucene). Conversely, the less useful metrics to predict defects have been: DIT (Eclipse JDT Core, Lucene and Eclipse PDE UI – with other metrics), and NOC (Equinox).

Figure 5 illustrates some of the time series where a Granger-causality has been detected. In this figure, we

Table IX
PERCENTAGE OF METRICS TIME SERIES WITH A POSITIVE RESULT FOR GRANGER CAUSALITY

	JDT Core	PDE UI	Equinox	Lucene	Total
CBO	48	38	24	39	41
DIT	27	31	20	13	27
LCOM	61	40	21	43	47
NOC	42	41	11	33	39
RFC	68	41	27	44	51
WMC	67	41	25	39	50
FANIN	36	31	25	37	34
FANOUT	58	38	24	45	44
NOA	59	38	33	34	45
NOPA	52	61	23	25	46
NOPRA	49	39	21	35	40
NOAI	42	51	18	31	44
LOC	67	41	27	44	50
NOM	60	41	20	47	47
NOPM	59	39	22	48	46
NOPRM	50	38	37	47	42
NOMI	40	31	19	40	35

have circulated the events in the time series of metrics that have probably anticipated similar events in the time series of defects.

Lesson Learned #5: Our findings reinforce previous observations in the literature about the absence of a single universal metric for predicting defects [22]. On the other hand, we found that metrics related to inheritance are not good predictors for defects, at least according to Granger. However, this result is not surprising, since the number of times that subclasses are added/removed or a class is moved up/down in the hierarchy is usually low.

F. What are the lag values that most led to positive results for Granger Causality?

It is well known that the Granger Test is sensitive to the lag selection. For this reason, as described in Section III-C, we have not fixed a single lag, but applied the test successively four times for each pair of series, with the lags ranging from one to four. In this way, whenever one of such lags returned a positive result, we have computed the existence of causality.

Table X shows the lags that have been most successful in returning positive results. When multiple lags returned causality, we chose the one with the lowest *p-value*. As we can note, the results have been different for each system. For Eclipse JDT Core, 49% of the causalities have been established for a lag equals to 1. For Eclipse PDE, the distribution has been almost uniform among the four lags. For Equinox and Lucene, the most successful lags have been equal to 2 and 3, respectively.

We can interpret such results as follows. First, changes were made in the considered systems (which we will call event A). Such changes had an impact in the values of the metrics considered in our study (event B). Frequently, such changes also introduced defects in the source code (event C) and some of them became failures (event D).

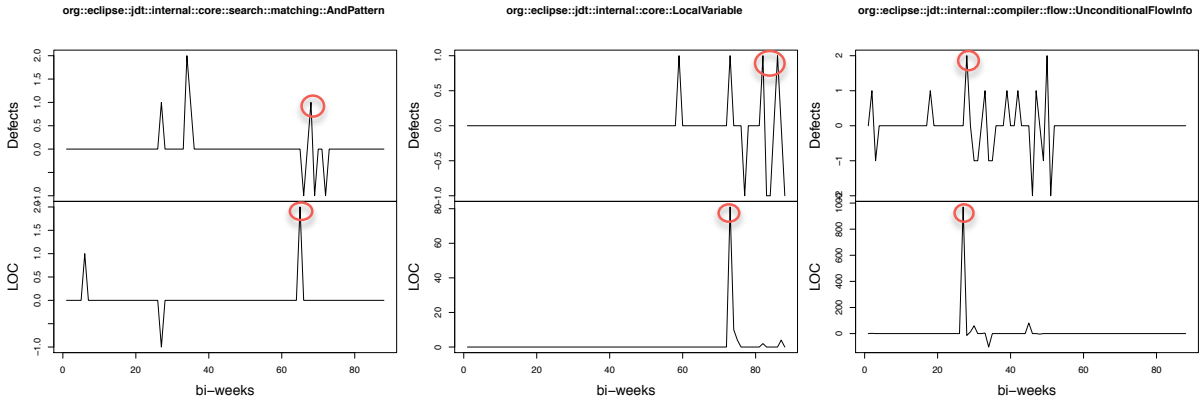

Figure 5. Examples of Granger Causality between LOC and defects.

Table X
PERCENTAGE OF LAG VALUES WITH A POSITIVE RESULT FOR GRANGER CAUSALITY

Lag	JDT Core	PDE UI	Equinox	Lucene
1	49	27	15	22
2	23	23	39	19
3	12	25	26	36
4	16	26	21	23
Total	100	100	100	100

In this description, events A, B, and C can be considered as happening at the same moment and they are succeeded by event D. Essentially, we have used Granger in this experiment to show the existence of causality between events B and D. Following with this interpretation, Granger's lag is the distance between such events in the time. Therefore, the results in Table X suggest that in the case of the Eclipse JDT Core most bugs were perceived by the users in one bi-week. In the Lucene system, this interval has increased to three bi-weeks.

Lesson Learned #6: When applying Granger to uncover causal relations between software metrics and defects, it is fundamental to run the tests with various lags. The reason is that the time elapsed between the inception of a defect in the source code and its perception by the users as a failure can vary significantly from system to system and also among the different types of bugs of a particular system.

V. THREATS TO VALIDITY

In this section, we discuss potential threats to the validity of our study. We have arranged possible threats in three categories: external, internal, and construct validity [23]:

External Validity: Our study on the causal relationships between bugs and software metrics involved four systems publicly provided in the D'Ambros dataset, including three systems from the Eclipse project and one system from the Apache Foundation, with a total of 4,298 classes. Therefore, we believe this sample includes a credible number of classes, representing real-world and non-trivial applications, with a consolidated number of users and a relevant history of bugs. Despite this observation, our findings – as usual in empirical software engineering – cannot be generalized to other systems, specifically to systems implemented in other languages or to systems from different domains, such as real-time systems, embedded systems or even to non-open source systems.

Internal Validity: This form of validity concerns the factors that can influence our observations. A possible threat concerns the R function we have used to discover stationary time series. We have used the function *adf.test()* that receives as parameters the time series to be checked and a lag. Particularly, we have relied on the default lag suggested by the function.

Construct Validity: This form of validity concerns the relationship between theory and observation. A possible threat concerns the way we have linked bugs to defects in classes. Particularly, we have discarded bugs without explicit references in the textual description of the commits. However, the percentage of such bugs was not large (around 25% of the bugs considered in the time frame of the experiment). Moreover, this approach is commonly used in experiments that need to map bugs to classes [9].

VI. RELATED WORK

D'Ambros *et al.* provided the dataset with the historical values of the object-oriented metrics used in our work. By making this dataset publicly available, their goal was to establish a common ground for comparison between bug prediction approaches [8], [9]. They relied on this dataset to evaluate a representative set of prediction approaches reported in the literature, including approaches based on source code metrics, change metrics, bug fixes, and entropy of changes. The authors also propose two

new metrics called churn and entropy of source code. Finally, the authors report a study on the explanative and predictive power of the mentioned approaches. The results showed that churn and entropy of source code have had the best results, achieving a better score in four out of the five analyzed systems. However, the results presented by D'Ambros *et al.* cannot be directly compared with our results, because they make use of standard regression models and we used the Granger Test that is based on bivariate autoregressive models. In a previous work, D'Ambros *et al.* have demonstrated the relationship between well-known design flaws (e.g. Brain Method, Feature Envy, Shotgun Surgery etc) and post-release defects [7].

Basili *et al.* have been one of the first to investigate the use of CK metrics as early predictors for fault-prone classes [3]. In a study on eight medium-sized systems they report on a correlation between the CK metrics (with the exception of the NOC metric) and fault-prone classes. Subramanyam *et al.* have later relied on the CK metrics to predict defect-prone components in an industrial e-commerce application with subsystems implemented in C++ and Java. They concluded that the metrics recommended to predict defects may vary across these two languages. For modules in C++, they report that WMC, DIT, and CBO with DIT have had the most relevant impact on the number of defects. For the modules in Java, only CBO with DIT has had an impact on defects.

Nagappan *et al.* have conducted a study on five components of the Windows operating system in order to investigate the relationship between complexity metrics and field defects [22]. They concluded that metrics indeed correlate with defects. However, they also highlight that there is no single set of metrics that can predict defects in all the five Windows components (which we have also observed in our experiments and summarized as Lesson Learned #5, Section IV-E). As a consequence of this finding, they suggest that software quality managers can never blindly trust on metrics, i.e. in order to use metrics as early bug predictors we must first validate them from the history [25]. Particularly, we consider that the methodology we have proposed in this paper provides guidance to apply this last suggestion. Basically, we have showed that developers can rely on Granger Test to discover in the history the metrics that are most useful to monitor the number of defects in each individual component of a software system.

Later, the study of Nagappan *et al.* has been replicated by Holschuch *et al.* to consider a large ERP system (SAP R3) [16]. However, both studies rely on linear regression models and correlation tests, which consider only the "immediate" relation between the independent and dependent variables. On the other hand, the dependence between bugs and object-oriented metrics may not be immediate, i.e. there may exist a delay or lag in this dependency. In this paper, we presented a new approach for monitoring bugs that considers this lag.

Hassan and Holt's Top Ten List is an approach that highlights to managers the ten most fault-prone subsystems of a given software system, based on the following heuristics: Most Frequently/Recently Modified, Most Frequently/Recently Fixed [15]. The goal is to provide guidance to quality managers, by suggesting they must invest their limited resources on the recommended subsystems. Similarly, our goal is to provide guidance to software managers, but by suggesting the top metrics for each component they must monitor more accurately.

Canfora *et al.* propose the use of the Granger Test to detect change couplings, i.e. set of software artifacts that are frequently modified together [4]. They claim that conventional techniques to determine change couplings fail when the changes are not "immediate" but due to subsequential commits. Therefore, they propose to use Granger Causality Test to detect whether past changes in an artifact a can help to predict future changes in an artifact b. More specifically, they propose the use of a hybrid change coupling recommender, obtained by combining Granger and association rules (the conventional technique to detect change coupling). After an experiment involving four open-source systems, they concluded that their hybrid recommender provides a higher recall than the two techniques alone and a precision in-between the two.

In previous research, we have showed that usually there is not a correspondence between the static location of the warnings raised by FindBugs [17] – a bug finding tool based on static analysis – and the methods changed by software maintainers in order to remove field defects [6]. However, warnings seem to be good indicators for the internal quality of a software system, mainly in terms of adherence to recommended programming practices and correct use of standard libraries. In fact, when we lifted the analysis to the level of projects, we have observed that those systems with a higher density of warnings have also presented a higher density of defects.

VII. Conclusions

To the best of our knowledge, we are the first to apply well-established techniques in the theory of time series to bug prediction. Particularly, in the study of time series, multivariate models – such as the Granger Test – are considered more robust to spurious regressions than traditional univariate models. After using Granger to mine for causalities between time series of metrics (publicly available in a benchmark specifically designed to compare bug prediction techniques) and time series of defects (extracted as part of the work described in this paper), we have been able to associate to the historical values of metrics the causes for 64% to 93% of the defects reported for the systems considered in our experiment. We have also been able to identify for each defective class the particular metrics that have Granger-caused the reported defects. Finally, as described in other studies, we could not identify a "holy grail" for bug prediction, i.e. a small set of metrics that are universally responsible for most of the defects, despite the considered systems. Instead, we have found that the metrics Granger-causing bugs can vary

significantly from system to system and also among the different types of bugs of a particular system.

In the near future, we plan to leverage the experience and knowledge gained from the described experiment to design and implement a tool that can alert developers about future defects, just after changes have been introduced in the repository of versions. For this purpose, for each component we need to identify the metrics that most contributed to bugs – task already done in this work – and to characterize the variation patterns of such metrics that in the past have led to bugs – our next task.

ACKNOWLEDGMENTS

This research has been supported by grants from CAPES, FAPEMIG, and CNPq. We thank Marco D'Ambros for making the dataset with the historical values of the OO metrics publicly available. We also thank Mauro Ferreira for the help with the Granger Test.

REFERENCES

[1] Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, 2007.

[2] Joao Eduardo Araujo, Silvio Souza, and Marco Tulio Valente. Study on the relevance of the warnings reported by Java bug finding tools. *IET Software*, 5(4):366–374, 2011.

[3] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[4] Gerardo Canfora, Michele Ceccarelli, Massimiliano Di Penta, and Luigi Cerulo. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In *26th International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[5] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[6] Cesar Couto, Joao Eduardo Araujo, Christofer Silva, and Marco Tulio Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, pages 1–17. To appear.

[7] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *10th International Conference on Quality Software (QSIC)*, pages 23–31, 2010.

[8] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Journal of Empirical Software Engineering*. To appear.

[9] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *7th Working Conference on Mining Software Repositories (MSR)*, pages 31–41, 2010.

[10] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, 2000.

[11] Wayne A. Fuller. *Introduction to Statistical Time Series*. John Wiley & Sons, 1994.

[12] Clive Granger. Investigating causal relations by econometric models and cross-spectral methods. *Econometrica*, 37(3):424–438, 1969.

[13] Clive Granger. Some properties of time series data and their use in econometric model specification. *Journal of Econometrics*, 16(6):121–130, 1981.

[14] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *31st International Conference on Software Engineering (ICSE)*, pages 78–88, 2009.

[15] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *International Conference on Software Maintenance (ICSM)*, pages 263–272, 2005.

[16] Tilman Holschuh, Markus Pauser, Kim Herzig, Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects in SAP Java code: An experience report. In *31st International Conference on Software Engineering (ICSE)*, pages 172–181, 2009.

[17] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[18] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *4th International Workshop on Principles of Software Evolution (IWPSE)*, pages 37–42, 2001.

[19] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[20] Nitai D. Mukhopadhyay and Snigdhansu Chatterjee. Causality and pathway search in microarray time series experiment. *Bioinformatics*, 23(4):442–449, 2007.

[21] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *27th International Conference on Software Engineering (ICSE)*, pages 580–586, 2005.

[22] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *28th International Conference on Software Engineering (ICSE)*, pages 452–461, 2006.

[23] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. A primer on empirical studies (tutorial). In *Tutorial presented at 19th International Conference on Software Engineering (ICSE)*, pages 657–658, 1997.

[24] Ramanath Subramanyam and M. S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transaction on Software Engineering*, 29(4):297–310, 2003.

[25] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. *Predicting Bugs from History*, chapter 4, pages 69–88. Springer, 2008.