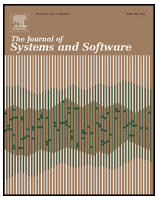
Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



Controversy Corner

Predicting software defects with causality tests

Cesar Couto[a,b,*], Pedro Pires[a], Marco Tulio Valente[a], Roberto S. Bigonha[a], Nicolas Anquetil[c]

[a] *Department of Computer Science, UFMG, Brazil*
[b] *Department of Computing, CEFET-MG, Brazil*
[c] *RMoD Team, INRIA, Lille, France*

ARTICLE INFO

Article history:
Received 6 May 2013
Received in revised form 13 January 2014
Accepted 14 January 2014
Available online xxx

Keywords:
Defect prediction
Causality
Granger test

ABSTRACT

In this paper, we propose a defect prediction approach centered on more robust evidences towards causality between source code metrics (as predictors) and the occurrence of defects. More specifically, we rely on the Granger causality test to evaluate whether past variations in source code metrics values can be used to forecast changes in time series of defects. Our approach triggers alarms when changes made to the source code of a target system have a high chance of producing defects. We evaluated our approach in several life stages of four Java-based systems. We reached an average precision greater than 50% in three out of the four systems we evaluated. Moreover, by comparing our approach with baselines that are not based on causality tests, it achieved a better precision.

1. Introduction

Defect prediction is a central challenge for software engineering research (Basili et al., 1996; Zimmermann et al., 2008; D'Ambros et al., 2010; Kamei et al., 2013). The goal is to discover reliable predictors that can indicate in advance those components of a software system that are more likely to fail. Clearly, this information is of central value for software quality assurance. 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 defect prediction techniques have been proposed. Essentially, such techniques rely on different predictors, including source code metrics (e.g., coupling, cohesion, size) (Basili et al., 1996; Subramanyam and Krishnan, 2003; Nagappan et al., 2006), change metrics (Hassan, 2009), static analysis tools (Nagappan and Ball, 2005; Araujo et al., 2011; Couto et al., 2013), and code smells (D'Ambros et al., 2010). Specifically, in a recent paper we reported a study showing the feasibility of using causality tests to predict defects in software systems (Couto et al., 2012). We relied on a statistical hypothesis test proposed by Clive Granger to evaluate whether past changes to a given source code metrics time series can be used to forecast changes in defects time series. Granger test was originally proposed to evaluate causality between time series of economic data (e.g., to show whether changes in oil prices cause recession) (Granger, 1969, 1981). Although extensively used by econometricians, the test was also used in bioinformatics (to identify gene regulatory relationships, Mukhopadhyay and Chatterjee, 2007) and recently in software maintenance (to detect change couplings spread over an interval of time, Canfora et al., 2010). In our study, we found that 64–93% of the defects in four well-known open-source systems were detected in classes with a Granger-positive result between the respective time series of source code metrics and defects.

In this paper, we leverage this initial study by proposing and evaluating a defect prediction model based on causality tests. More specifically, we not only report that Granger-causalities are common between time series of source code metrics and defects (which is essentially a theoretical result), but we also propose a model that relies on this finding to trigger alarms as soon as changes that are likely to introduce defects in a class are made (i.e., a model that can contribute effectively to software quality assurance practices). Fig. 1 provides details on our approach for defect prediction. In a first step, we apply the Granger test to infer possible Granger-causalities between historical values of source code metrics and the number of defects in each class of the system under analysis. In this first step, we also calculate a threshold for variations in the values of source code metrics that in the past Granger-caused defects in such classes. For example, suppose that a Granger-causality is found between changes in the size of a given class in terms of lines

[*] Corresponding author at: Department of Computing, CEFET-MG, Brazil. Tel.: +55 3186612513.
E-mail addresses: cesarfmc@dcc.ufmg.br, cesar@decom.cefetmg.br, cesarfmc@gmail.com (C. Couto), ppires@dcc.ufmg.br (P. Pires), mtov@dcc.ufmg.br (M.T. Valente), bigonha@dcc.ufmg.br (R.S. Bigonha), nicolas.anquetil@inria.fr (N. Anquetil).

0164-1212/$ – see front matter © 2014 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2014.01.033

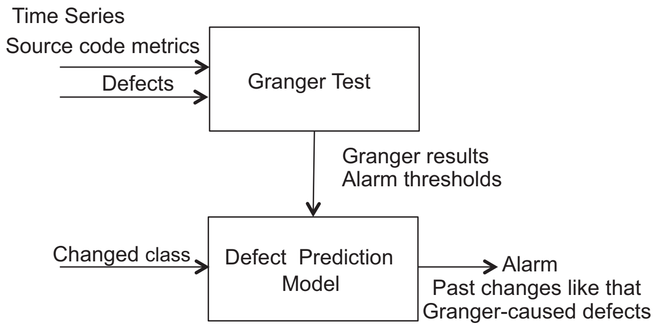


Fig. 1. Proposed approach to predict defects.

of code (LOC) and the number of defects in this class. Considering previous changes in this specific class, we can establish for example that changes adding more than 50 lines of code are likely to introduce defects in this class (more details on how such thresholds are calculated in Section 3.3). Using these thresholds and the Granger results calculated in the previous step, a defect predictor analyzes each change made to a class and triggers alarms when similar changes in the past Granger-caused defects.

Regarding our initial study, we also extended a dataset proposed to evaluate defect prediction approaches, by almost doubling the number of source code versions included in this dataset. Finally, we evaluated our approach in several life stages of four open-source systems included in the aforementioned dataset. Our approach reached an average precision greater than 50% considering three out of the four systems we evaluated. Moreover, our results show that the precision of the alarms changes with time. For example, for the Eclipse JDT Core, we achieved an average precision of 58% considering 144 models covering seven years of the system's history, and including a minimal and maximal precision of 27% and 90%, respectively. On the other hand, we were not able to predict all defects using times series of source code metrics. On average, we achieved recall rates ranging from 13% (Equinox Framework) to 31% (Lucene). In fact, we argue that it is not feasible to expect that alarms based on source code metrics variations can cover the whole spectrum of bugs reported to a system. Finally, we show that our models outperform models that trigger alarms without considering Granger-causality or that are based on linear regression techniques.

The remainder of this paper is organized as follows. We start with an overview on Granger Causality (Section 2). Next, we describe the steps to build the proposed model (Section 3), including the time series extraction, the application of the Granger test, and the identification of thresholds in metrics variations that may lead to defects. Section 4 describes our dataset including time series of source code metrics and defects for four real-world systems (Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Lucene). Section 5 describes a feasibility study designed to illustrate and to evaluate the application of Granger on defects prediction. We present an evaluation of the proposed model in Section 6. Section 7 discusses related work, and Section 8 concludes the paper.

2. Granger causality

In this section, we start first by describing a precondition that Granger requires the time series to follow (Section 2.1). Next, we present and discuss the test (Section 2.2).

2.1. Stationary time series

The 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 (Fuller, 1994). 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 source code 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 (Lehman, 1980). 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 (Granger, 1981).

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 one. More specifically, 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 stationary behavior, we will consider a time series that represents the number of methods (NOM), extracted for the Eclipse JDT Core system, in intervals of bi-weeks, from 2001 to 2008. Fig. 2(a) illustrates this series. As we can observe, the series is not stationary, since it has a clear growth trend, with some disruptions along the way. Fig. 2(b) shows the first difference of NOM. Note that most values are delimited by a constant mean and variance. Therefore, NOM in first difference has a stationary behavior.

2.2. 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 (Granger, 1969). If this happens, we can conclude that x Granger-causes y. The most common implementation of the Granger Causality Test uses bivariate and univariate auto-regressive models. A bivariate auto-regressive model includes past values from the independent variable x and from the dependent variable y. On the other hand, a univariate auto-regressive model considers only past values of the variable y.

To apply Granger, we must first calculate the following bivariate auto-regressive model (Canfora et al., 2010):

$$\begin{aligned} y_t = {} & \alpha_0 + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \cdots + \alpha_p y_{t-p} + \beta_1 x_{t-1} \\ & + \beta_2 x_{t-2} + \cdots + \beta_p x_{t-p} + u_t \end{aligned} \tag{1}$$

where p is the auto-regressive lag length (an input parameter of the test) and u_t is the residual. Essentially, p defines the number of past values—from both x and y—considered by the regressive models. Furthermore, Eq. (1) defines a bivariate model because it uses values of x and y, limited by the lag p.

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

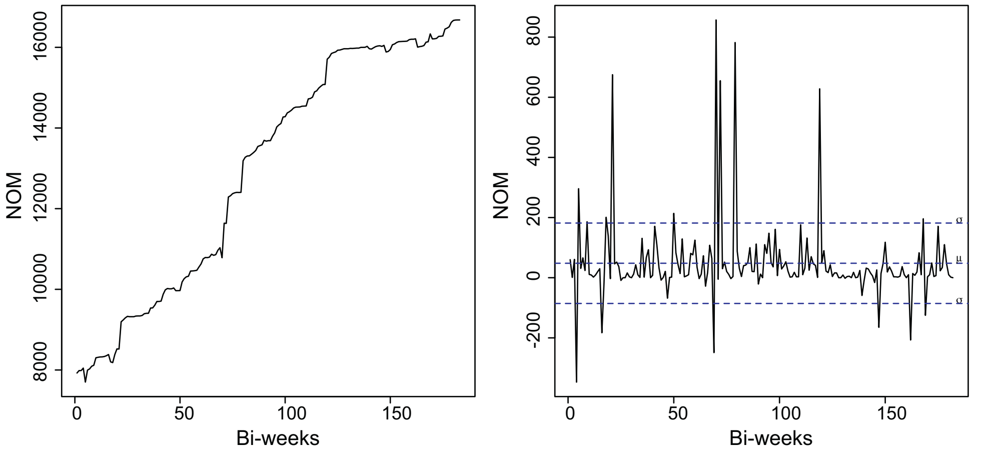
$$H_0 : \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

This hypothesis assumes that past values of x do not add predictive power to the regression. In other words, by testing whether the β coefficients are equal to zero, the goal is to discard the possibility that the values of x 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 Eq. (1) but excluding the values of x):

$$y_t = \gamma_0 + \gamma_1 y_{t-1} + \gamma_2 y_{t-2} + \cdots + \gamma_p y_{t-p} + e_t \tag{2}$$

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033



(a) Original NOM series (non-stationary behavior) (b) NOM series in first difference (stationary behavior)

Fig. 2. NOM for Eclipse JDT Core.

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

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

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. Therefore, the null hypothesis is rejected. In this case, we can conclude that x Granger-causes y.

Example 2. In our previous Eclipse JDT Core example, we 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 returns a p-value of 0.32, which is superior to the defined threshold of 5%. To explain this 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.). Therefore, Granger's application is recommended in scenarios where variations in the independent variable are reflected in the dependent variable after a delay (or lag).

Example 3. To explain the sense of causality captured by Granger in a simple and comprehensive way, 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 reapplied Granger to evaluate whether NOPM causes NOM', in the Granger sense. In this case, the result was positive, assuming $p = 5$. Therefore, knowing the NOPM values at a given bi-week helps to predict the value of NOM′. Fig. 3 illustrates the behavior of both series. For example, we can observe that just before bi-week 21 a significant increase 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 this increase in NOPM propagates to NOM′ in few bi-weeks (we circled both events in the presented series).

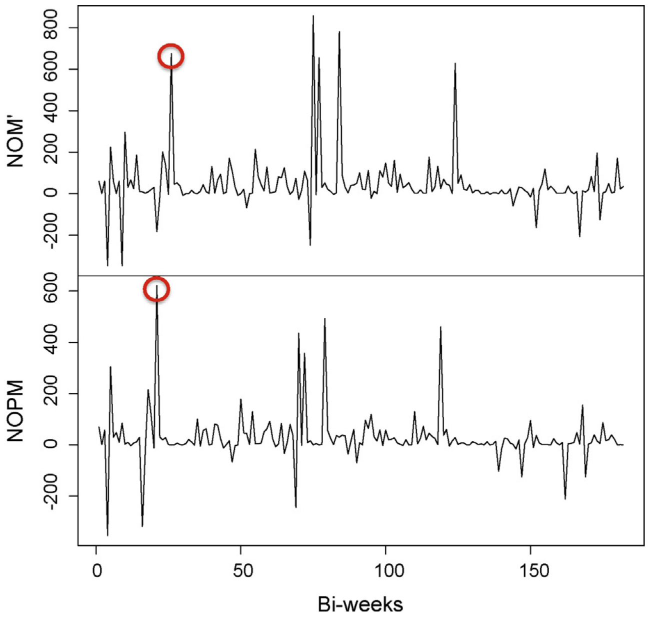


Fig. 3. NOPM and NOM′ time series. The increase in NOPM values just before bi-week 21 has been propagated to NOM′ few weeks later.

Example 4. To illustrate the application of Granger in a real example, Fig. 4 shows the time series of LOC (lines of code) and defects for four classes of the Eclipse JDT Core system. These time series were created in intervals of bi-weeks from 2001 to 2008. In the figure, we circled the events in the time series of LOC that probably anticipated similar events in the time series of defects. For example, in the `SearchableEnvironmentRequestor` class (first series), the increase in LOC just before bi-week 87 generated an increase in the number of defects few weeks later. In this class specifically, a Granger-causality has been detected between LOC and defects, assuming $p = 3$.

3. Proposed approach

The ultimate goal of our approach is to predict defects using a model centered on Granger-causality relations between source code metrics (independent variables) and defects (dependent

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033

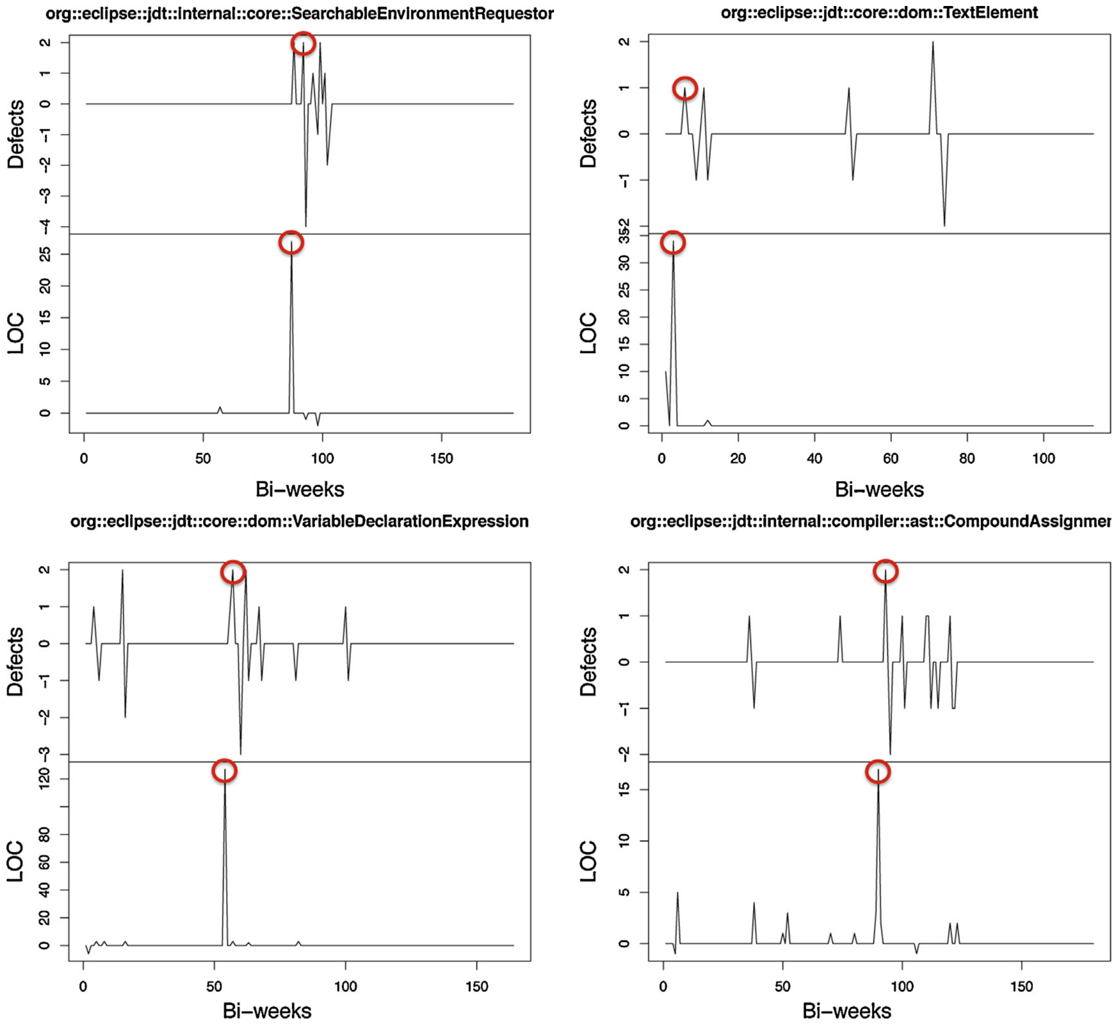


Fig. 4. Examples of Granger causality between LOC and defects.

variable). Our approach relies on historical data such as bug histories (extracted from bug tracking platforms) and source code versions (extracted from version control platforms). This data is used to create time series of source code metrics and defects for the classes of a target system. Next, we rely on the Granger causality test for inferring relations between the time series of metrics and defects. After that, we build a defect prediction model that triggers alarms when changes made to the target system have a high chance of producing defects.

As illustrated in Fig. 5, we propose the following steps to build a defect prediction model:

1. We create time series of source code metrics for each class of the target system. To create such series, source code versions of the target system are extracted from its version control platform in a predefined time interval (e.g., bi-weeks). After that, the values of the considered source code metrics are calculated for each class of each extracted version.
2. We create a time series with the number of defects in each class of the target system from the bugs history. Basically, we map the bugs reported in bug tracking platforms to their respective commits using the bug identifier. Next, the files changed by such commits are used to identify the classes changed to fix the respective defects (i.e., the defective classes). Section 3.1 details the methodology we follow to generate the defects time series.
3. We apply the Granger causality test considering the metrics and defects time series. More specifically, Granger is responsible for identifying Granger-cause relations on time series of source code metrics and defects. Section 3.2 describes the methodology we follow to apply Granger.
4. As a distinguishing aspect of our approach, we identify thresholds for variations in metrics values that may contribute according to Granger to the occurrence of defects. More specifically, we build a model that relies on such thresholds to alert developers about future defects whenever a risky variation in the values of a metric happens due to changes in the system. Section 3.3 describes the proposed approach to identify alarms thresholds.

3.1. Extracting the time series of defects

We consider that bugs are failures in the observable behavior of the system. Bugs are caused by one or more errors in the source code, called defects (IEEE Standard, 1990). We count defects at the class level since our ultimate goal is to trigger alarms due to changes in classes. More specifically, each class changed to fix a given bug is counted as a defective class. Therefore, whenever mentioned that a system has n defects in a given time frame, we are actually stating that we counted n defective classes in this time frame (i.e., classes that were later changed to fix the defect). Classes with multiple

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033

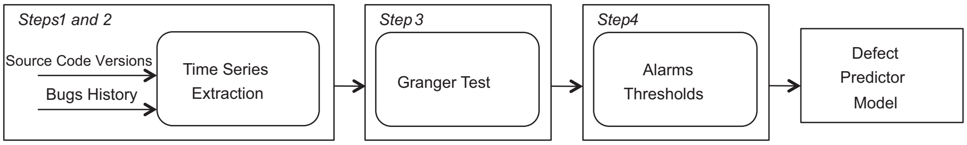


Fig. 5. Steps proposed to build a model for defect prediction.

defects related to the same bug are counted only once; on the other hand, defects in the same class but due to different bugs are counted separately. Finally, we do not consider open or non-fixed bugs.

To create the time series of defects, the bugs—or more precisely, the maintenance requests—reported in the bug tracking platforms must be collected during the same time frame used to extract the source code versions. In a second step, each bug b is linked to the classes changed to fix b, using the following procedure (which is also adopted in other studies on defect prediction (D'Ambros et al., 2010; Zimmermann et al., 2007; Śliwerski et al., 2005)):

1. Suppose that *Bugs* is the set containing the IDs of all bugs reported during the time frame considered in the analysis.
2. Suppose that *Commits* is the set with the IDs of all commits in the version control platform. Suppose also that $Cmts[c]$ and $Chg[c]$ are, respectively, the maintainer's comments and the classes changed by each commit $c \in Commits$.
3. The classes changed to fix a given bug $b \in Bugs$ are defined as:

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

This set is the union of the classes changed by each commit c whose textual comments provided by the maintainer includes a reference to the bug with ID b. The predicate $substr(s_1, s_2)$ tests whether s_1 is a substring of s_2.

Finally, suppose that in order to fix a given bug b changes were applied to the class C. In this case, a defect associated to b must be counted for C during the period in which b remained open, i.e., between the opening and fixing dates of b. More specifically, a defect is counted for the class C at a time interval t whenever the following conditions hold: (a) b has been opened before the ending date of the time interval t; (b) b has been fixed after the starting date of the time interval t.

Fig. 6 shows an example regarding the extraction of a time series of defects with three bugs and three classes and spanning a time interval of five bi-weeks. The left table shows data on the bugs and the right figure shows the time series of defects extracted from these bugs. As we can observe, bug #1 was opened in 2010-01-07 (bi-week 1) and fixed in 2010-03-10 (bi-week 5). In order to fix this bug, changes were applied to the class A. In this case, a defect associated to bug #1 is counted for the class A during five bi-weeks.

3.2. Applying the Granger test

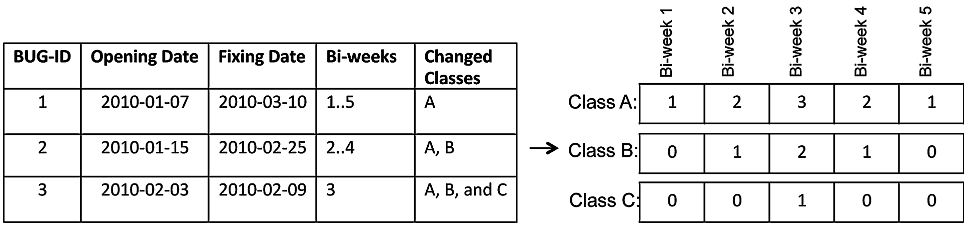
To apply the Granger causality test in order to identify causal relations on the time series of source code metrics and defects, we propose the Algorithm 5. In this algorithm, `Classes` is the set of all classes of the system (line 1) and `Defects[c]` is the time series with the number of defects (line 2). The algorithm relies on function `d_check` (line 3) to check whether the defects in the time series `d` conform to the following preconditions:

Algorithm 5. Applying the Granger test

```
1:  for all c ∈ Classes do
2:    d = Defects[c];
3:    if d_check(d) then
4:      for n = 1 → NumberOfMetrics do
5:        m = M[n][c];
6:        if m_check(m) then
7:          granger(m, d);
8:        end if
9:      end for
10:   end if
11: end for
```

- P1: The time series must have at least k values, where k represents the minimum size that a series must have to be considered by the prediction model. Therefore, time series that only existed for a small proportion of the time frame considered in the analysis—usually called dayfly classes (Lanza, 2001)—are discarded. The motivation for this precondition is the fact that such classes do not present a considerable history of defects to qualify their use in predictions.
- P2: The values in the time series of defects must not be all null (equal to zero). Basically, the goal is to discard classes that never presented a defect in their lifetime (for instance, 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, they will remain with zero defects in the future.
- P3: The time series of defects must be stationary, which is a precondition required by Granger, as reported in Section 2.1.

Suppose that a given class `c` has passed the previous preconditions. For this class, suppose also that `M[n][c]` (line 5) is the time series with the values of the `n`-th source code metric considered in the study, 1≤ `n` ≤ `NumberOfMetrics`. The algorithm relies on function `m_check` (line 6) to test whether time series `m`—a time



BUG-ID	Opening Date	Fixing Date	Bi-weeks	Changed Classes
1	2010-01-07	2010-03-10	1..5	A
2	2010-01-15	2010-02-25	2..4	A, B
3	2010-02-03	2010-02-09	3	A, B, and C

→

	Bi-week 1	Bi-week 2	Bi-week 3	Bi-week 4	Bi-week 5
Class A:	1	2	3	2	1
Class B:	0	1	2	1	0
Class C:	0	0	1	0	0

Fig. 6. Example of extracting time series of defects.

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033

series with metrics values—conforms to the following preconditions:

- P4: The time series of source code metrics must not be constant. In other words, metrics time series whose values never change must be discarded, since variations in the independent variables are the key event to observe when computing Granger causality.
- P5: The time series of source code metrics must be stationary, as defined for the defects series.

Finally, for the time series `m` (source code metrics) and `d` (defects) that passed preconditions P1 to P5, function `granger(m,d)` checks whether `m` Granger-causes `d` (line 7). As described in Section 2, Granger is sensitive to the lag selection. For this reason, in the proposed algorithm the test is not applied for a single lag value, but several times, with the lags ranging from 1 to *l*. In this way, we consider that a metric `m` is a Granger-cause of defects in a given class `c` whenever one of the tested lags return a positive result.

3.3. Calculating thresholds to trigger alarms

As described in Section 2.2, Granger causality test identifies whether an independent variable *x* contributes to predict a dependent variable *y* at some stage in the future. However, the test does not establish the thresholds for relevant variations of the values of *x* that may impact *y*. Therefore, this step aims to calculate the thresholds used by our model to trigger alarms, as follows:

1. For each time series of source code metrics that Granger returned a positive result, we compute the positive variations in the series values, by subtracting the values at consecutive bi-weeks.[1]
2. A threshold to trigger alarms for a given class *C* and metric *m* is the arithmetic mean of the variations of *m* computed for *C*, as defined in the previous step.

Fig. 7 shows a time series for one of the classes of the Eclipse JDT Core system where Granger returned a positive result between the values of LOC and defects. In this figure, we circled the positive variations used to calculate the alarms thresholds. As can be observed, the threshold for the class `BindingResolver` is 33.1, which is the arithmetic mean of the values we circled. The proposed defect prediction model relies on this threshold to alert maintainers about future defects in this class. More specifically, an alarm is triggered by our model for future changes adding at least 34 lines of code to this class.

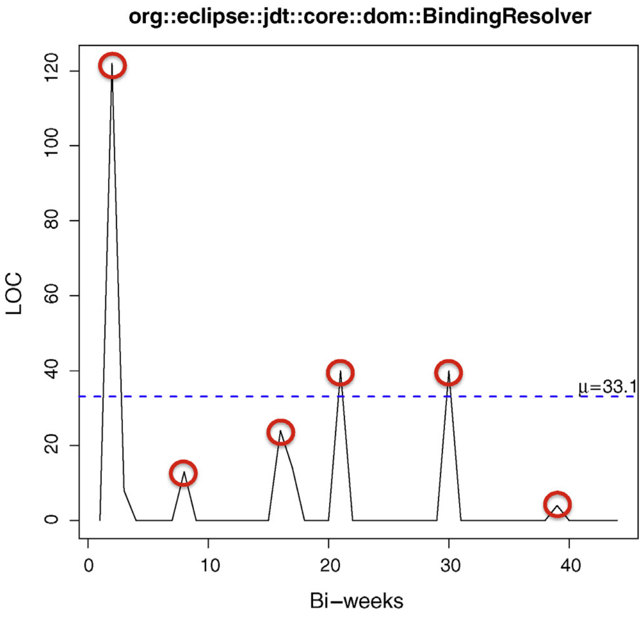


Fig. 7. Example of a threshold for the LOC metric.

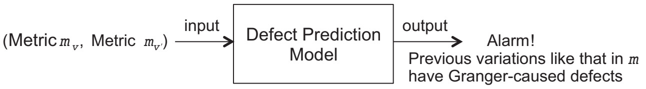

(Metric m_v, Metric $m_{v'}$)
input
Defect Prediction Model
output
Alarm!
Previous variations like that in m have Granger-caused defects

Fig. 8. Defect prediction model

3.4. Defect prediction model

Fig. 8 illustrates the inputs and the output of our prediction model. Basically, the model receives as input two values for a given source code metric *m*, m_v and $m_{v'}$, where m_v is the value of the metric regarding a class *C* that was just changed to fix a bug. Moreover, $m_{v'}$ is the value of the metric in the previous version of *C* in the version control platform. The proposed model verifies whether *m* Granger-causes defects in *C* and whether the difference $(m_v - m_{v'})$ is greater or equal to the threshold identified for variations in the metric values. When both conditions hold, the model triggers an alarm. Basically, such alarm indicates that, according to the Granger test, similar variations in the values of this metric in the past resulted in defects.

Using the Prediction Model: With this model in hand, a maintainer before making a commit in the version control platform with changes to a given class can verify whether such changes may lead to defects. If our model triggers an alarm for a given class warning about future occurrences of defects, the maintainer can for example perform extra software quality assurance activities in this class (e.g., unit testing or a detailed code inspection) before executing the commit.

4. Dataset

The evaluation reported on this paper is based on a dataset made public by D'Ambros *et al.* to evaluate defect prediction techniques (D'Ambros et al., 2010, 2012). This dataset includes temporal series for seventeen source code metrics, including number of lines of code (LOC) and the CK (Chidamber and Kemerer) metrics suite (Chidamber and Kemerer, 1994). The metrics were extracted in intervals of bi-weeks for four well-known Java-based systems: Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Lucene. Table 1 provides detailed information on this original dataset. In this table, column **Period** informs the time interval in which the metrics were collected by D'Ambros et al. In total, the dataset has

Table 1
Original dataset

System	Period	Classes	Versions
Eclipse JDT Core	2005-01-01–2008-05-31	1041	90
Eclipse PDE UI	2005-01-01–2008-09-06	1924	97
Equinox framework	2005-01-01–2008-06-14	444	91
Lucene	2005-01-01–2008-10-04	889	99
Total		4298	377

[1] We decided to compute the positive variations because they typically indicate a degradation in the internal quality of the source code, which may influence the occurrence of future defects. Therefore, at least in principle, it does not make sense to trigger alarms in cases where the variations in the metric values are negatives, i.e., when the source code quality improves.

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033

Table 2
Extended dataset

System	Period	Classes	Versions
Eclipse JDT Core	2001-07-01–2008-06-14	1370	183
Eclipse PDE UI	2001-05-24–2008-04-03	3478	180
Equinox Framework	2003-11-25–2010-10-05	615	180
Lucene	2002-06-22–2009-05-02	760	180
Total		6223	723

Table 4
Number of bugs, defects, and defects per bugs

System	Bugs	Defects	Defective classes	Defects/bugs
Eclipse JDT Core	3697	11,234	833	3.04
Eclipse PDE UI	1798	3566	1019	1.99
Equinox Framework	784	1478	292	1.88
Lucene	335	615	157	1.83
Total	6614	16,893	2297	2.18

4298 classes, each of them with at least 90 bi-weekly versions (which is equivalent to around three and a half years).

4.1. Extended dataset

We extended this dataset as described next: (a) by considering more versions and classes and (b) by creating a time series of defects. Table 2 provides detailed information on our extended dataset. As can observed, our extension has approximately twice the number of versions (723 versions) and 45% more classes (6223 classes). Basically, we extended the original dataset to consider—whenever possible—the whole evolution history of the considered systems, starting from the first version available in their version repositories.

Similar to the original dataset, our extension does not include test classes. Test classes were discarded because they are not related to the core functionality of their systems and therefore they may statistically invalidate attempts to perform predictions. More specifically, we removed the directories and subdirectories whose name starts with the words “Test” or “test”. The number of removed classes is as follows (for the last versions included in our dataset): 3452, 208, 816, and 360 classes for Eclipse JDT Core, Eclipse PDE UI, Equinox Framework, and Lucene, respectively.

Furthermore, we consider a reduced number of source code metrics, as indicated in Table 3. More specifically, we reduced the number of source code metrics from seventeen to seven, for the following reasons:

- The seven metrics we selected cover different properties of code, such as complexity (WMC), coupling (FAN-IN and FAN-OUT), cohesion (LCOM) and size (NOA, LOC, and NOM).
- The metrics related to inheritance—such as Depth of Inheritance Tree (DIT) and Number of Children (NOC)—usually do not present positive results regarding the Granger causality test, at least according to our previous study (Couto et al., 2012).

It is important to highlight that eventual collinear relations between the considered source code metrics values do not have a major impact in our model. Basically, collinear pairs of metrics (like NOM and LOC, possibly) just tend to produce multiple alarms for the same defects, assuming that a Granger-causality is detected between them and defects. For this reason, we did not check for collinearity in our dataset. The following sections describe the extraction process of the time series of source code metrics and defects provided in the dataset.

Table 3
Metrics considered in our dataset.

	Metrics	Description	Category
1	WMC	Weighted methods per class	Complexity
2	LCOM	Lack of cohesion in methods	Cohesion
3	FAN-IN	Number of classes that reference a given class	Coupling
4	FAN-OUT	Number of classes referenced by a given class	Coupling
5	NOA	Number of attributes	Size
6	LOC	Number of lines of code	Size
7	NOM	Number of methods	Size

4.2. Data collection

To create the time series of source code metrics, we extracted the source code of each considered version from the version control platform in intervals of bi-weeks. We then used the Moose platform[2] to calculate the metrics values for each class of each considered version, excluding only test classes. Particularly, we relied on VerveineJ—a Moose application—to parse the source code of each version and to generate MSE files. MSE is the default file format supported by Moose to persist source code models. We extended the Moose platform with a routine to calculate LCOM, since the current version of Moose does not support this metric.

Another important difference between the datasets is the fact that D'ambros' dataset only provides information on the total number of defects for each class. Thus, in order to apply Granger we distributed this value along the bi-weeks considered in our evaluation. To create the time series of defects, we followed the methodology described in Section 3.1. We initially collected the issues (bugs) reported in the Jira and Bugzilla platforms (the bug tracking platforms of the considered systems) that meet the following conditions:

- Issues reported during the time interval considered by our dataset (as described in Table 2).
- Issues denoting real corrective maintenance tasks. Our goal was to distinguish between issues demanding corrective maintenance tasks and issues that in fact are requests for adaptive, evolutive or perfective maintenance. Jira has a field that classifies the issues as *bug*, *improvement*, and *new feature*. Therefore, we collected only issues classified as *bug*. On the other hand, Bugzilla is used mainly for corrective maintenance tasks (at least for Eclipse Foundation systems). Despite that, some issues were classified as *enhancement* in the *Severity* field. Therefore, we also discarded them.
- Issues having *fixed* status. In other words, we discarded *open*, *duplicate*, *invalid*, and *incomplete* issues.

In a second step, we mapped the bugs to defects in classes and created the time series of defects for each class. Table 4 shows the number of bugs opened via Bugzilla or Jira for each of the systems. As can be observed, we collected a total of 6614 bugs. This table also shows the number of bugs we collected, the number of defects that caused such bugs (i.e., number of classes changed to fix such bugs, according to the definition of defects, provided in Section 3.1), the number of defective classes (i.e., number of classes associated to at least one bug), and the average number of defects per bug. As can be observed, on average each bug required changes in 2.18 classes. Therefore, at least in our dataset, changes to fix bugs do not present a scattered behavior.

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

Table 5
Percentage and absolute number of classes conforming to preconditions P1, P2, and P3

System	P1 (%)	Classes	P1 + P2 (%)	Classes	P1 + P2 + P3 (%)	Classes
Eclipse JDT Core	80	1090	59	811	57	779
Eclipse PDE UI	45	1582	26	918	23	788
Equinox Framework	65	397	44	271	36	219
Lucene	59	450	19	142	17	131
Total	57	3519	34	2142	31	1917

5. Feasibility study

In this section, we describe a first study designed to evaluate our approach for defect prediction using Granger causality test. Besides illustrating the use of Granger, we investigate the feasibility of using the approach proposed in Section 3 to predict defects in the dataset described in Section 4. More specifically, we focus on the following questions: (a) How many time series pass the preconditions related to defects (preconditions P1, P2, P3)? (b) How many time series pass the preconditions related to source code metrics (preconditions P4 and P5)? (c) How many classes present positive results on the Granger test? (d) What is the number of defects potentially covered by our approach? (e) What are the lags that most led to positive results regarding Granger-causality? To answer these questions, we used the entire dataset described in Section 4. Therefore, we analyzed 6223 classes, 16,893 defects, and approximately 50,000 time series of source code metrics and defects, with a maximum size of 183 bi-weeks (JDT Core) and a minimal size of 180 bi-weeks (PDE UI, Equinox Framework, and Lucene).

Parameters setting: An important decision when applying the proposed defect prediction model is setting the parameters used in the preconditions, as described in Section 3.2. In practice, we decided to set such parameters in the following way:

- *Minimum size*: We defined that the classes should have a lifetime of at least 30 bi-weeks (approximately one year). Our goal is to select classes with a sufficient history of defects that qualify their use in predictions (and therefore to tackle the cold-start problem that typically happens when making predictions based on historical data (Schein et al., 2002)).
- *Maximum lag*: We computed the tests using a lag ranging from 1 to 6. To set this maximum lag, we analyzed the time interval between the opening and fixing dates of the bugs in our dataset. On average, 84% of the bugs were fixed within six bi-weeks.
- *Significance level*: We computed the tests using a significance level of 95% ($\alpha = 0.05$). We counted as causality the cases where the *p*-value obtained by applying the F-Test was less than or equal to α, i.e., *p*-value ≤ 0.05.

Tool support: The algorithm described in Section 3.2 was implemented in the R statistical system. We considered all times series in first difference (see Section 2.1) to maximize the number of stationary time series—a precondition to apply the Granger test. To identify stationary time series, we relied on function *adf.test()* of the *tseries* package. This function implements the Augmented Dickey–Fuller test for stationary behavior (Fuller, 1994). More specifically, this function receives as parameters the time series to be checked and a lag. Particularly, we relied on the default lag suggested by the function. To apply the Granger test, we used function *granger.test()* of the *msbvar* package.

5.1. Preconditions on time series of defects

The algorithm proposed in Section 3.2 first checks whether the defects times series pass the preconditions P1, P2, and P3 using function `d_check`. Table 5 shows the percentage and the absolute number of classes that survived these preconditions. We can observe that 57% of the classes survived precondition P1 (lifetime greater than 30 bi-weeks) and that 34% of the classes survived both P1 and P2 (at least one defect in their lifetime). Finally, our sample was reduced to 31% of the classes after applying the last precondition (test for stationary behavior). In summary, after checking the preconditions P1, P2, and P3, our sample was reduced significantly.

5.2. Preconditions on time series of source code metrics

The second step of the algorithm described in Section 3.2 relies on function `m_check` to evaluate the preconditions P4 and P5. Considering only the classes passing preconditions P1, P2, and P3, Table 6 shows the percentage of source code time series that passed preconditions P4 and P5. As defined in Section 3.2, precondition P4 states that the time series must not be constant and P5 requires the series to be stationary. By observing the values in Table 6, we conclude that constant time series are common for some metrics. For example, for LCOM, FAN-IN, and NOA approximately 40% of the considered classes presented a constant behavior (column total). Furthermore, we can observe that the number of series with non-stationary behavior—even when considering the first differences—is not negligible. For example, for WMC, 84% of the series survived P4, but only 74% survived P5. In summary, after checking the preconditions P4 and P5, our sample of time series of source code metrics was reduced to 65%.

5.3. Defects covered by Granger

After checking the proposed preconditions, the algorithm computes function `granger` to check the existence of

Table 6
Percentage of time series conforming successively to preconditions P4 and P5.

	JDT Core (%)		PDE UI (%)		Equinox (%)		Lucene (%)		Total (%)	
	P4	P4 + P5	P4	P4 + P5	P4	P4 + P5	P4	P4 + P5	P4	P4 + P5
LCOM	70	66	53	43	64	53	63	59	62	54
WMC	88	84	79	64	84	72	85	84	84	74
FAN-IN	60	57	47	39	50	43	76	73	55	49
FAN-OUT	76	72	80	69	76	67	75	73	78	70
NOA	61	57	60	51	62	53	59	55	61	54
LOC	94	91	92	73	90	79	95	89	93	82
NOM	80	75	76	62	76	66	79	76	77	69
Total	76	72	70	57	72	62	76	73	73	65

Table 7
Percentage and absolute values of classes with *n* positive results for Granger.

n	JDT Core		PDE UI		Equinox		Lucene	
	%	**Classes**	%	**Classes**	%	**Classes**	%	**Classes**
0	62	849	88	3,067	80	491	90	682
1	9	122	4	141	5	31	2	16
2	7	99	2	71	3	18	2	13
3	6	78	2	55	4	23	2	13
4	6	77	2	53	4	22	1	9
5	4	53	1	45	3	20	2	13
6	4	49	1	36	1	7	1	11
7	2	43	0	10	0	3	0	3
Total	100	1370	100	3478	100	615	100	760

Table 8
Classes, Granger positive classes (GPC), number of bugs, number of defects, number of defects in Granger positive classes (DGC).

System	Classes	GPC	Bugs	Defects	DGC	DGC/defects
Eclipse JDT Core	1370	521	3697	11,234	8781	78%
Eclipse PDE UI	3478	411	1798	3566	2391	67%
Equinox Framework	615	124	784	1478	766	52%
Lucene	760	78	335	615	462	75%
Total	6223	1134	6614	16,893	12,400	73%

Granger-causality. Table 7 shows for each class `c` the number of tests with a positive result considering the series `M[n][c]` and `Defects[c]`, where `M[n][c]` is one of the seven series of metrics for a given class `c` ($1 \leq n \leq 7$) and `Defects[c]` is the series of defects for this class. For example, for Eclipse JDT Core, 62% of the classes have no Granger-causality relationship between their defects series and one of the metrics series (Table 7, first line). Stated otherwise, in 38% of the classes in the Eclipse JDT Core (i.e., 521 classes), we were able to detect a Granger-causality relation between the series of defects and at least one of the seven series of metrics; in around 9% of the classes Granger returned a positive result for a single series of metrics, and so on. In the remaining three systems—Eclipse PDE UI, Equinox, and Lucene—the percentage of classes where the test found a Granger-causality connection between metrics and defects was 12% (411 classes), 20% (124 classes), and 10% (78 classes), respectively. In summary, our sample was reduced considerably to 18% (1134 classes) of its original size after applying Granger.

Finally, it is fundamental to check the number of defects in this subset of 1,134 classes. Table 8 shows the following results: number of classes, number of Granger positive classes (column GPC), number of bugs we initially collected, number of defects that caused such bugs, and number of defects detected in our subset of 1134 classes (column DGC). More specifically, considering the classes with at least one positive result for Granger, Table 8 shows that 73% of the defects collected in our dataset were detected in such classes. Therefore, by combining the results in Tables 7 and 8, we conclude that our preconditions and the Granger results reduced our sample to 18% of its original size. However, such classes concentrate 73% of the defects in our dataset. Considering that there are many bugs not related to variations in source code metrics, it is natural to expect that our coverage would be significantly less than 100%. On the other hand, an average coverage of 73% shows that it is at least feasible to rely on Granger to predict defects in software systems.

As previously described, the Granger tests were calculated using a significance level of 95% ($\alpha = 0.05$). In other words, we counted as a Granger-causality the cases where the p-value obtained by applying the Granger test was less than or equal to α, i.e., p-value ≤ 0.05. Table 9 shows the percentage of tests with a positive result distributed in intervals of 1%. As can be observed, approximately 70% of the tests with a positive result returned a p-value less than 0.01 for all considered systems.

Table 9
Granger positive *p*-values.

	JDT Core (%)	PDE UI (%)	Equinox (%)	Lucene (%)
$0.04 < p\text{-value} \leq 0.05$	5	5	3	2
$0.03 < p\text{-value} \leq 0.04$	5	5	3	2
$0.02 < p\text{-value} \leq 0.03$	5	7	6	6
$0.01 < p\text{-value} \leq 0.02$	10	8	8	6
$0.00 < p\text{-value} \leq 0.01$	65	70	70	64
Total	100	100	100	100

5.4. Lags considered by Granger

It is well known that the Granger test is sensitive to the lag selection (Granger, 1981). For this reason, as described in Section 3.2, we do not fix a single lag, but calculate the test successively for each pair of series, with the lags ranging from one to six. Whenever one of such lags returns a positive result, we assume the existence of Granger-causality.

Table 10 shows the lags that were most successful in returning positive results. When multiple lags returned causality, we chose the one with the lowest p-value. As we can note, we achieved different results for each system. For Eclipse JDT Core, 33% of the Granger-causalities were established for a lag equal to six bi-weeks. For Eclipse PDE UI and Equinox, the most successful lag was equal to one bi-week. For Lucene, the distribution was almost uniform among the six lags.

We can interpret such results as follows. First, changes were made to the considered systems (which we will call event A). Such changes have an impact in the values of the metrics considered in our study (event B). Frequently, such changes also introduced

Table 10
Percentage of lags with a positive result for Granger-causality (highest values in bold).

Lag	JDT core	PDE UI	Equinox	Lucene
1	15	**30**	**40**	19
2	17	15	11	12
3	14	15	18	14
4	11	12	10	18
5	10	12	11	17
6	**33**	16	10	**20**
Total	100	100	100	100

defects in the source code (event C) and some of them became bugs reported in the system's bug tracking platform (event D). In this description, events A, B, and C can be considered as happening at the same time and they are succeeded by event D. Essentially, we rely on Granger to show the existence of causality between events C and D. According to this interpretation, Granger's lag is the typical distance between such events in the time. Therefore, the results in Table 10 suggest that in the case of the Eclipse JDT Core and Lucene most bugs were perceived by the developers in six bi-weeks. In contrast, for the Eclipse PDE UI and Equinox, this interval was of just one bi-week, in most of the cases.

To summarize, when applying the Granger test to uncover causal relations between source code metrics and defects, it is important to run the tests with various lags. The reason is that the time between the inception of a defect in the source code and its perception by the maintainers as a bug can vary significantly.

6. Model evaluation

In the feasibility study reported in Section 5, we concluded that, even by reducing our sample to 18% of the classes after applying the preconditions and the Granger test, it was possible to cover 73% of the defects in our dataset. Motivated by such positive results, we decided to conduct a second study to evaluate our model for triggering defects alarms. More specifically, this study aimed to answer the following research questions:

RQ1: **What is the precision and recall of our approach?** With this question, we want to investigate whether our models provide reasonable levels of precision and recall.

RQ2: **How does our approach compares with the proposed baselines?** Our aim with this question is to analyze the precision and recall of our approach when compared to three baselines. The first baseline does not consider the results of the Granger test, the second one does not consider both the preconditions defined in Section 3.2 and the results of the Granger test and the third one uses simple linear regression as prediction technique.

RQ3: **What is the impact of using other functions (different from the mean) for triggering alarms?** As defined in Section 3.3, the thresholds used to trigger alarms for a given class *C* and metric *m* is the mean of the positive variations of the values of *m* computed for *C*. Therefore, our goal with this question is to investigate whether alternative descriptive statistics functions—such as minimum, first quartile, median, third quartile, and maximum—provide better results than the mean when used to trigger alarms.

RQ4: **Regarding their severity, what types of bugs are typically predicted by the proposed models?** Our goal with this research question is to investigate whether our models tend to predict with higher accuracy some particular categories of bugs, in terms of severity. Particularly, we intend to rely on the severity categories informed by the users of the Bugzilla and Jira tracking platforms.

In this section, we start by presenting the methodology followed in our evaluation (Section 6.1). After that, we provide answers and insights for our research questions (Section 6.2). Finally, we discuss threats to validity (Section 6.3).

6.1. Evaluation setup

We performed the following steps to answer the proposed research questions:

1. We divided the time series (considered in their first differences) in two parts. We used the first part (training series) to build a defect prediction model and the second part (validation series) to validate this model. Moreover, we defined that the time series start in the first bi-week with a reported defect. For example, for the Eclipse JDT Core, our training series start in the bi-week 8, because we have not found defects in the previous bi-weeks. We also defined the size of the validation series as including exactly 18 bi-weeks, i.e., approximately six months (which is a time frame commonly employed in studies on defect prediction (D'Ambros et al., 2010; Holschuh et al., 2009; Schröter et al., 2006)). For example, for a time series with 50 bi-weeks in the Eclipse JDT Core, we discarded the first seven bi-weeks (since the first defect appeared only in the 8th bi-week). We used the next 25 bi-weeks for training, and the 18 remaining bi-weeks for validation.
2. We created a defect prediction model for each system according to the methodology described in Sections 3.2, 3.3, and 3.4. More specifically, we first checked the preconditions and applied the Granger test considering the source code metrics (independent variables) and the defects (dependent variable) time series. Next, we identified the thresholds for variations in the metrics values that may have contributed to the occurrence of defects. Finally, we created a prediction model that triggers defects alarms.
3. We defined three baselines to evaluate the models constructed in the Step 2. In these baselines, the way to calculate the thresholds is exactly the one used by our approach, i.e, the arithmetic mean of the positive variations of the metrics. However, they differ on the preconditions and on the use of the Granger test, as described next:
   (a) The first baseline is a model created using time series meeting the preconditions P1 to P5, but that does not consider the results of the Granger test. Therefore, variations in any source code metrics that respect the preconditions can trigger alarms (i.e., even when a Granger-causality is not detected). The purpose of this baseline is to check whether the Granger test contributes to improve the precision of the proposed models.
   (b) The second baseline considers time series meeting only precondition P1 (i.e., this model does not consider the preconditions P2 to P5 and the results of the Granger test). We preserved precondition P1 because it is fundamental to remove classes with a short lifetime that do not help on reliable predictions. An alarm is triggered by variations in any metric that respects the first precondition, even when a Granger-causality is not detected. The central purpose of this second baseline is to evaluate the importance of the preconditions in the proposed model.
   (c) The third baseline considers time series meeting the preconditions P1 to P5, but instead of applying the Granger test, we created a simple linear regression model. More specifically, this model is composed by linear equations that correlate each source code metric separately (independent variable) and the defects time series (dependent variable). We checked the significance of the individual coefficients of the regressions in order to identify if a given metric is effective to predict the occurrence of defects. Therefore, alarms are only triggered due to variations in the metrics whose individual coefficients are statistically significant ($\alpha = 0.05$). The main goal of this third baseline is to evaluate whether the Granger test is more effective than linear regressions to express relations between source code metrics and defects.
4. We evaluated our models using precision and recall measures. Precision evaluates whether the alarms issued by the model are

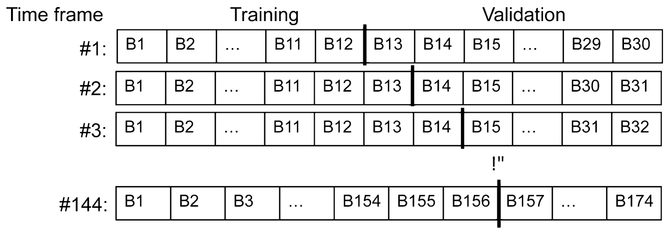

Time frame
Training
Validation
#1: B1 B2 … B11 B12 | B13 B14 B15 … B29 B30
#2: B1 B2 … B11 B12 B13 | B14 B15 … B30 B31
#3: B1 B2 … B11 B12 B13 B14 | B15 … B31 B32
!"
#144: B1 B2 B3 … B154 B155 B156 | B157 … B174

Fig. 9. Training and validation time series (Eclipse JDT Core).

confirmed by defects. To calculate precision, we used only the validation time series, i.e., series with values not considered during the model construction phase. An alarm issued in a given bi-week *t* is classified as a *true alarm* when a new defect is identified at most six bi-weeks after bi-week *t*. Therefore, we calculate precision in the following way:

$$\text{Precision} = \frac{\text{number of true alarms}}{\text{number of alarms}}$$

Conversely, recall measures whether the alarms triggered by our approach cover the defects found in Granger positive classes. To calculate recall we checked whether the occurrences of defects in the validation series were preceded by an alarm. More specifically, we checked whether a defect in a given bi-week *t* was preceded by an alarm in at most six bi-weeks before *t*. We calculate recall in the following way:

$$\text{Recall} = \frac{\text{number of true alarms}}{\text{number of defects}}$$

5. We repeated Steps 1–4 for several time frames, i.e., for multiple training and validation time series. Our main goal is to evaluate the proposed approach in different life stages of the considered systems. Fig. 9 illustrates the time frames considered for the Eclipse JDT Core. As presented, we created and validated 144 different models, considering different time frames. The first time frame has 30 bi-weeks, including 12 bi-weeks to build the model (training time series) and 18 bi-weeks to validate the model (validation time series). To generate a new model, we extended the previous training series in one bi-week. For example, the second time frame has 31 bi-weeks, the third one has 32 bi-weeks, etc. Finally, the last time frame has 174 bi-weeks. For the systems Eclipse PDE UI, Equinox Framework, and Lucene, we created and validated 125, 145, and 115 models, respectively.

6.2. *Results*

In this section, we provide answers to our research questions.

6.2.1. *RQ1: What is the precision and recall of our approach?*

To address this research question, we followed Steps 1–5 described in Section 6.1. Therefore, we created and validated models for each time frame of the systems considered in this evaluation. Our main goal was to evaluate the proposed approach in different life stages of the considered systems. The tables in Fig. 10 shows the values we measured for true alarms, precision, recall, and F-measure for the considered systems. Considering all time frames, the tables also report the following results: maximum value (Max), the top 5%, 10%, and 20% values, minimum value (Min), average, median, and standard deviation (Std. Dev.). As can be observed, our approach reached an average precision ranging from 28% (Eclipse PDE UI) to 58% (Eclipse JDT Core) and a median precision ranging from 31% (Eclipse PDE UI) to 58% (Eclipse JDT Core). Furthermore, some particular models presented high precisions, 90%, 60%, 100%, and 88%, for the Eclipse JDT Core, Eclipse PDE UI, Equinox and Lucene, respectively.

In general terms, we can conclude that our approach reached reasonable levels of precision in many life stages of the considered systems. This result is a distinguishing contribution of our evaluation, since defect prediction approaches typically analyze a single time frame (Couto et al., 2013; Kamei et al., 2013; Holschuh et al., 2009; D'Ambros et al., 2010; Giger et al., 2012). For example, D'Ambros et al. created and validated their defect prediction models for the Eclipse JDT Core for a single time frame (2005-01-01 to 2008-06-17) (D'Ambros et al., 2010). For the same system, we created and validated defect prediction models for 144 time frames achieving an average precision of 58%.

On the other hand, specifically for the Eclipse PDE UI system, our approach obtained an average precision of just 28%. Probably, this result was due to the low mapping rate between bugs and commits in this system. While for Eclipse JDT Core, Equinox, and Lucene we obtained a mapping rate of approximately 70%, Eclipse PDE UI reached a mapping rate around 46% (i.e., from the 3913 bugs reported on the bug tracking platform, only 1798 were linked to a commit on the version control platform).

We can also observe that some of the evaluated models triggered a significant number of true alarms. For example, for the system Eclipse JDT Core, the maximum number of true alarms triggered by a given model was 168 (for the model constructed in the time frame 49). Probably, this result is explained by a major maintenance activity in the system during the validation period of this model. We measured on average 277 classes changed per bi-week in this particular validation period, while this rate considering the entire period of analysis is 218. Fig. 11 illustrates some validation time series where an alarm triggered by this model was later confirmed by the occurrence of defects. In this figure, we circled the true alarms issued by the model.

Despite such encouraging results regarding precision, our approach presented an average recall ranging from 13% (Equinox) to 31% (Lucene) and a median recall ranging from 12% (Equinox) to 30% (Lucene). In practice, this result shows that we were not able to cover all defects in all life stages of the considered systems. We argue that the main reason is the fact that there is a large spectrum of bugs that can be reported for any system. Probably, some types of bugs are less impacted by variations in the values of the source code metrics. For example, we can mention bugs related to usability concerns, internationalization, and JavaDoc documentation.[3] Despite this fact, we achieved reasonable levels of recall in particular time frames. For example, for the Eclipse JDT, Eclipse PDE UI, Equinox, and Lucene systems, the maximum values for recall were 68%, 44%, 31%, and 52%, respectively.

> RQ1: Our approach reached an average precision greater than 50% in three out of the four systems we evaluated. On the other hand, as expected, we were not able to trigger alarms for all defects using times series of source code metrics as predictors. On average, we achieved recall rates ranging from 13% (Equinox Framework) to 31% (Lucene).

[3] As an example, we can mention the following bug reported for the Eclipse JDT Core system: *"Bug 10495 – typo in ASTNode::MALFORMED javadoc, 'detcted' should be 'detected'"*. Because JavaDocs are comments in the source code, a class was changed to fix this bug and a respective defect was included in our defects time series. However, it is not feasible to suppose that bugs like that can be predicted. In fact, an alarm was never raised by our models for this particular bug.

Measure	TA	Pre	Rec	F
Max	168	90%	68%	70%
Top 5%	118	88%	48%	60%
Top 10%	95	82%	40%	55%
Top 20%	82	67%	32%	42%
Min	2	27%	7%	12%
Mean	56	58%	24%	33%
Median	49	58%	23%	32%
Std Dev	34	14%	13%	14%

(a) Eclipse JDT Core (144 models)

Measure	TA	Pre	Rec	F
Max	36	60%	44%	38%
Top 5%	33	41%	36%	33%
Top 10%	31	40%	33%	31%
Top 20%	27	36%	29%	30%
Min	1	6%	6%	7%
Mean	16	28%	24%	24%
Median	16	31%	23%	26%
Std Dev	11	11%	7%	7%

(b) Eclipse PDE UI (125 models)

Measure	TA	Pre	Rec	F
Max	21	100%	31%	44%
Top 5%	9	88%	25%	40%
Top 10%	8	80%	22%	35%
Top 20%	7	75%	19%	27%
Min	1	22%	5%	8%
Mean	5	53%	13%	20%
Median	4	46%	12%	20%
Std Dev	3	21%	7%	10%

(c) Equinox Framework (145 models)

Measure	TA	Pre	Rec	F
Max	16	88%	52%	50%
Top 5%	16	80%	50%	50%
Top 10%	14	78%	45%	48%
Top 20%	13	67%	43%	44%
Min	0	0%	0%	10%
Mean	8	51%	31%	36%
Median	7	48%	30%	37%
Std Dev	5	19%	13%	10%

(d) Lucene (115 models)

Fig. 10. Number of true alarms (TA), precision (Pre), recall (Rec), and F-measure (F).

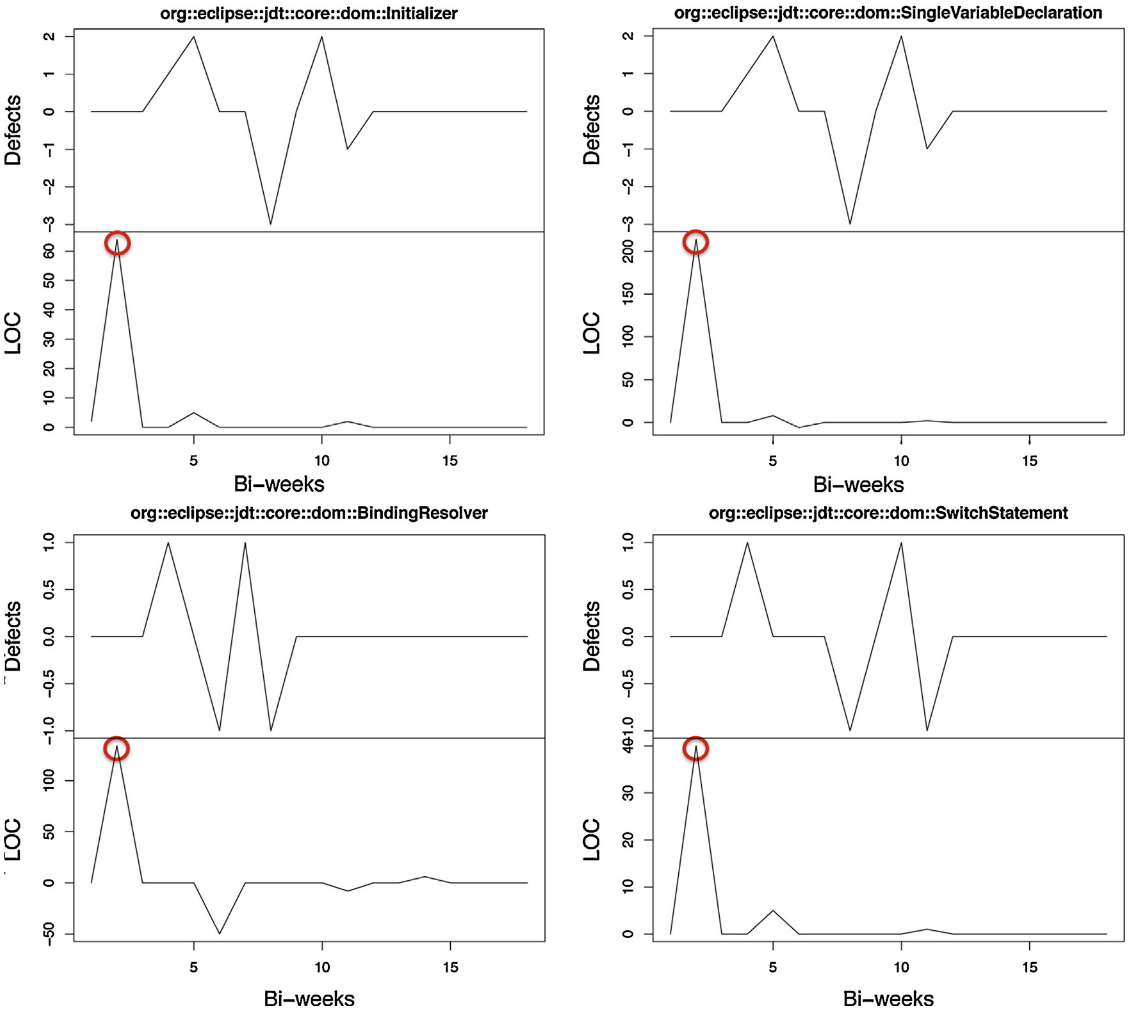


Fig. 11. True alarms raised by our approach.

6.2.2. RQ2: How does our approach compares with the proposed baselines?

This research question aims to compare the precision and recall of our approach with the three baselines. Fig. 12 shows for each time frame the precision results for the following models: (a) proposed approach (*Granger*); (b) *Baseline1* (baseline that does not consider the results of the Granger test); (c) *Linear* (baseline that uses simple linear regression as the prediction technique). As we can note, the initial time frames have no precision results. This lack of precision happened because we discarded results coming from unstable models, i.e., models reporting zero alarms or whose precision values alternate between 0 and 1. As we can observe in the figure, in most time frames, our approach (solid line) shows a precision greater than *Baseline1* (long dash line) and *Linear* (dotted line). To confirm this assumption, for each pair of samples (*Granger* vs. *Baseline1* and *Granger* vs. *Linear*), we applied a non-parametric statistical hypothesis test (Mann–Whitney *U* test) using a significance level of 95%. This test confirmed that the median precision of our approach (*Granger*) is significantly different from *Baseline1* in all systems (Eclipse JDT Core, Eclipse PDE UI, Equinox, and Lucene). Furthermore, the median precision of our approach is also significantly different from *Linear* in three out of the four systems (Eclipse JDT Core, Eclipse PDE UI, and Equinox).

It is worth mentioning that Fig. 12 also shows that in several time frames our approach reached high precision measures. For instance, for Eclipse JDT Core, between time frames 36 and 47, our models achieved a precision ranging from 83% to 90%, with the number of true alarms ranging from 40 to 138. For Eclipse PDE UI, our approach in the time frame 24 reached a precision of 60%, with three true alarms. For Equinox, between time frames 95 and 107, our approach reached a precision ranging from 60% to 75%, with the number of true alarms ranging from 6 to 11. Finally, for Lucene, between time frames 79 and 88, our approach reached a precision ranging from 66% to 87%, with the number of true alarms ranging from 2 to 7.

Fig. 13 shows for each time frame the precision results for the following models: (a) *Baseline1* and (b) *Baseline2* (baseline that only considers precondition P1). As we can observe, in most time frames, *Baseline1* (solid line) shows a precision greater than *Baseline2* (long dash line). In fact, the Mann–Whitney *U* test asserted that the median precision of *Baseline1* is significantly different from *Baseline2* (for this reason, we omitted *Baseline2* from Fig. 12).

It is also important to highlight that the precision results do not present a monotonically increasing behavior, as the evaluated models include more bi-weeks in the respective training time series. For example, the highest precision value for Eclipse JDT Core was achieved in bi-week 42 (precision = 90%) and the second lowest value 69 bi-weeks later (precision = 34%).

Considering recall, Fig. 14 compares our approach with two baselines: *Baseline1* and *Linear*. In general terms, the model based on Granger outperformed the baseline that uses simple linear regression as the prediction technique (*Linear*). The Mann–Whitney *U* test confirmed that the median precision of our approach (*Granger*) is significantly different from *Linear* in all systems. It is worth noting that *Linear* has better recall than the proposed approach in the first bi-weeks considered for the Equinox system. However, it is not possible to reason about this behavior without a deep knowledge on the defects reported for Equinox during such initial bi-weeks. On the other hand, the best recall was achieved by the baseline that does not consider the results of Granger or of any other prediction technique (*Baseline1*). In fact, this result is expected since *Baseline1* triggers more alarms and therefore such alarms have more chances to cover real defects.

To summarize, two conclusions can be derived from our investigation concerned this research question: (i) based on the fact that the *Baseline1* outperformed the *Baseline2*, we can conclude that

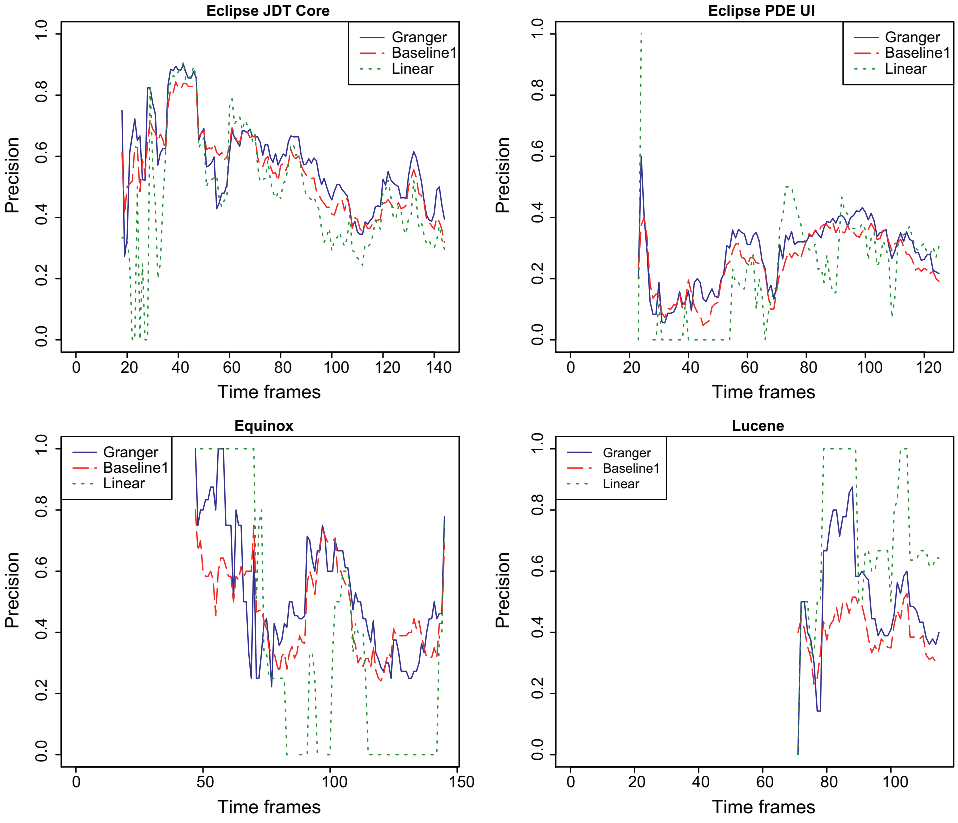


Fig. 12. Precision results for *Granger*, *Baseline1*, and *Linear*.

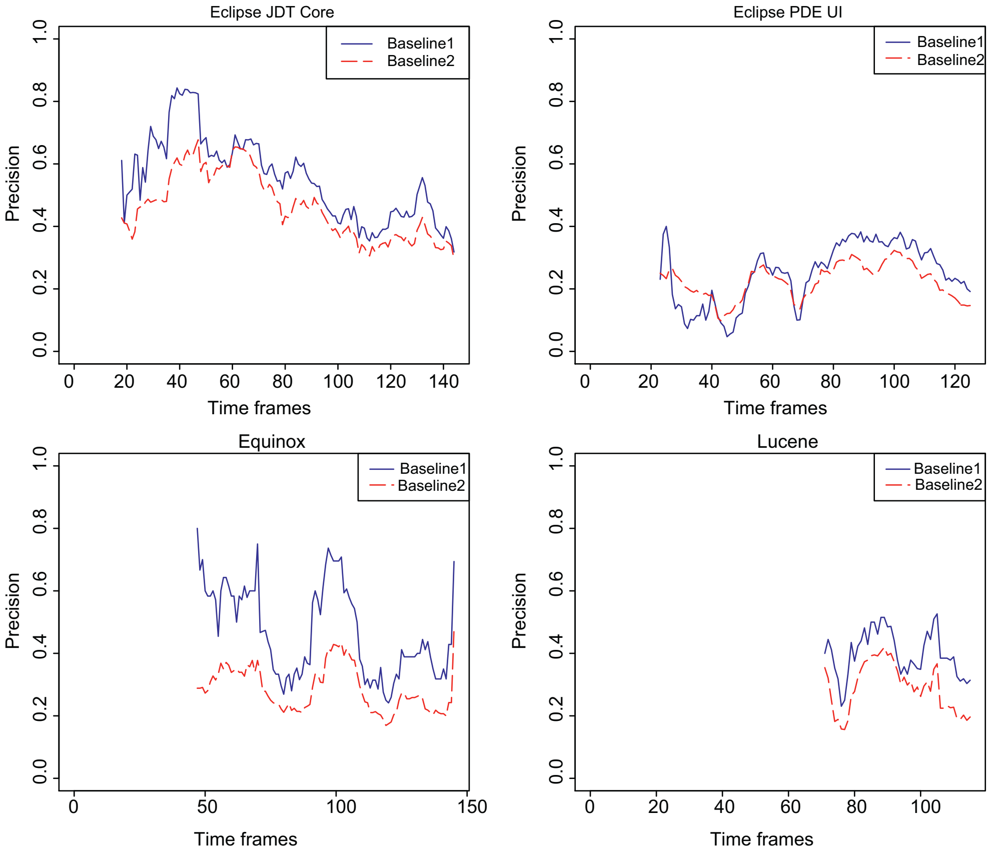


Fig. 13. Precision results for *Baseline1* and *Baseline2*.

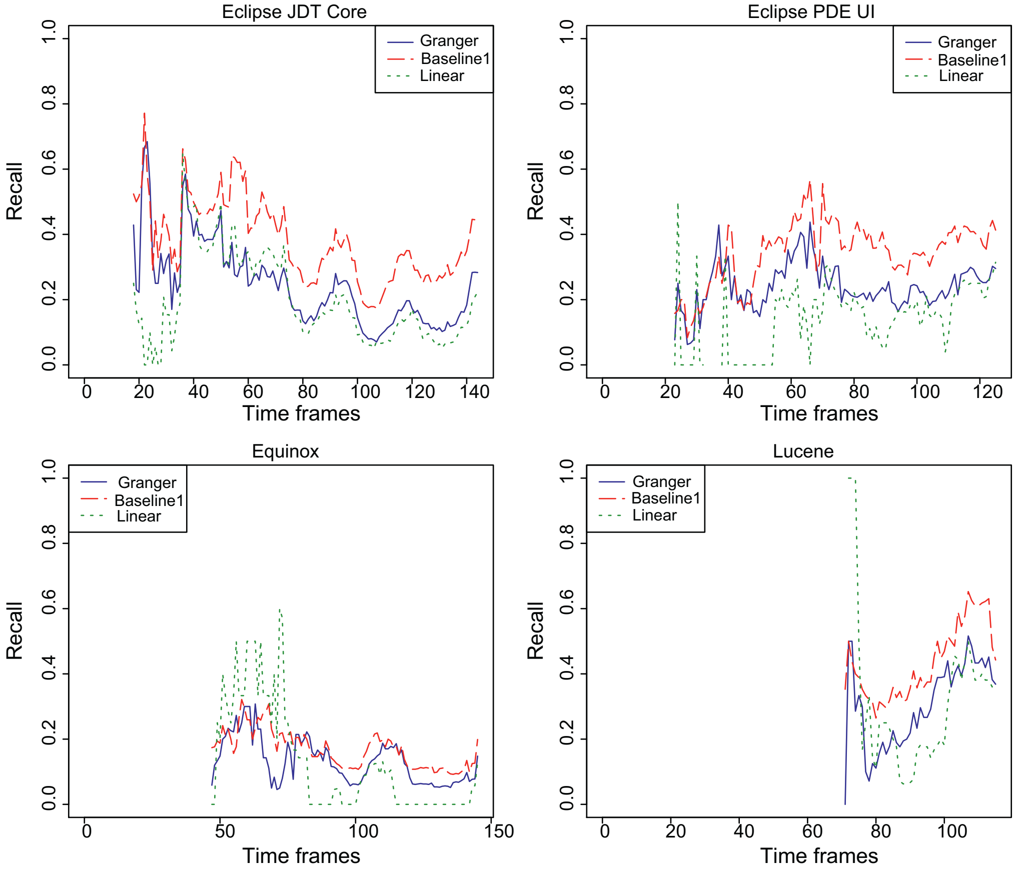


Fig. 14. Recall results for *Granger*, *Baseline1*, and *Linear*.

Table 11
Average precision (Pre) and recall (Rec) results considering alternative threshold functions

Function	JDT Core		PDE UI		Equinox		Lucene	
	Pre	Rec	Pre	Rec	Pre	Rec	Pre	Rec
Min	61%	53%	27%	54%	49%	33%	51%	51%
1st Quartile	60%	45%	27%	50%	50%	30%	52%	40%
Median	59%	37%	27%	43%	49%	24%	53%	36%
Mean	58%	24%	28%	24%	52%	13%	51%	31%
3rd Quartile	57%	26%	25%	29%	51%	17%	50%	29%
Max	51%	15%	28%	16%	51%	10%	50%	20%

when building defect prediction models, it is important to remove classes with zero defects (that make the predictions trivial), classes with a constant behavior (that do not contribute with predictive power), and classes with non-stationary time series (that may statistically invalidate the findings); and (ii) from the fact that *Granger* outperformed both *Baseline1* and *Linear*, we can conclude that it is possible to achieve gains in precision by considering the Granger test to raise alarms for defects (instead of traditional models, based for example on standard regressions).

> RQ2: The precision achieved by our approach was statistically better than the proposed baselines in three out of the four systems, which confirms the gains achieved by considering Granger-causality when predicting defects using source code metrics.

6.2.3. RQ3: What is the impact of using other functions (different from the mean) for triggering alarms?

To answer this third research question, the experiments conducted for answering the first questions were executed again changing only the function used to summarize the variations in the values of the source code metrics. More specifically, in the previous research questions the thresholds used to trigger alarms were defined as the *mean* of the positive variations in the values of the source code metrics that Granger-caused defects in a given class. On the other hand, in this research question, we evaluated five other descriptive statistics functions calculated over such variations: minimal value, first quartile value, median, third quartile value, and maximal value. For example, suppose that during the training window it was inferred that variations in the values of a given metric m Granger-caused defects in a class C. Therefore, a model based on the *minimal* function will trigger alarms for this class, during the validation window, when the value of m after a given change in C is greater than the *minimal* variation of m observed during the training period.

Table 11 presents the average precision and recall achieved by the evaluated models considering the aforementioned descriptive statistics functions. As can be observed, the precision is not affected by the considered functions. For example, for the Eclipse JDT Core system, the precision ranges from 51% (for alarms thresholds calculated using the maximal variation in the source code metrics values) to 61% (for thresholds equal to the minimal variation values). In the other systems, the dispersion was less than this one reported for the Eclipse JDT Core. Regarding recall, the best result is always achieved by the *minimal* function, since this function raises more alarms. Conversely, the worst recall is achieved by the *maximal* function. Therefore, we decided to use the mean function, since it represents an interesting balance between precision and recall.

Fig. 15 shows the number of alarms (Fig. 15a) and the number of true alarms (Fig. 15b) for each model evaluated in our study, considering only the Eclipse JDT Core. As expected, by changing the functions from the minimal to the maximal functions, the number of alarms decreases, i.e., the higher the threshold, the lower the number of alarms. However, Fig. 15b shows that the true alarms change in the same proportion, as we change the number of alarms. For this reason, the precision among the different threshold functions was very similar.

> RQ3: Because the evaluated functions presented similar results, we decided to trigger alarms using the mean variation in the values of the source code metrics, as originally proposed.

6.2.4. RQ4: Regarding their severity, what types of bugs are typically predicted by the proposed models?

To answer the fourth research question, we followed these steps: (a) for each alarm classified as a true alarm in our experiments, we located the defect responsible for this classification (called predicted defect); (b) each predicted defect was then mapped to its respective bug (called predicted bug), following a reverse process from the one described in Section 3.1; (c) for each predicted bug, we retrieved its severity degree in the Jira and Bugzilla tracking platforms. Basically, when reporting a bug, an user of such platforms can rank the bug in one of the following categories: blocker, critical, major, normal, minor, or trivial.

Fig. 16 presents the distribution of the bugs considered in our experiment by severity. The table shows the distribution of the whole population of bugs considered in the study and also the distribution of the bugs predicted by at least one of our models. As we can observe, the distributions are very similar, in all systems in our dataset. For example, 81% of the bugs we evaluated in the Eclipse JDT Core system are normal bugs; for this system, 84% of the bugs predicted by our approach were also classified as normal, regarding their severity.

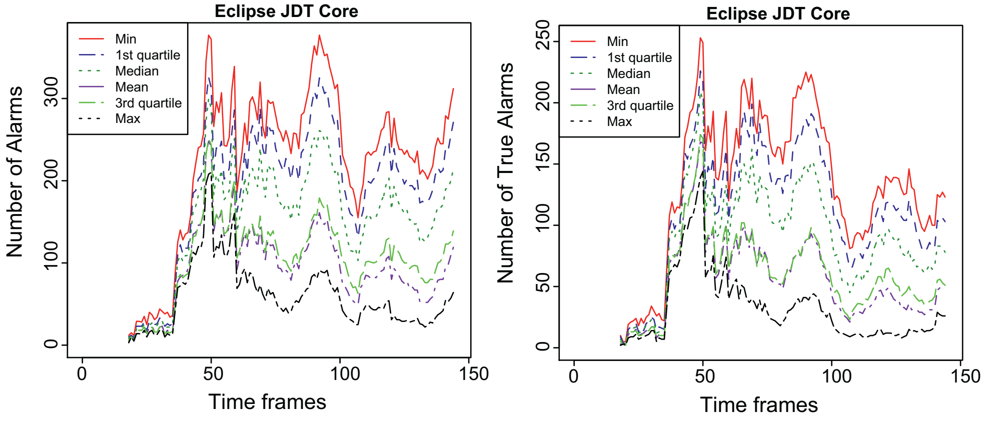
> RQ4: In terms of severity, we were not able to identify a particular category of bugs that the proposed model tends to predict with higher frequency.

6.3. Threats to validity

In this section, we discuss potential threats to the validity of our study. We arranged possible threats in three categories: external, internal, and construct validity (Perry et al., 1997):

External validity: Our study to evaluate the proposed defect prediction model involved four medium-to-large systems, including three systems from the Eclipse project and one system from the Apache Foundation, with a total of 6223 classes. Therefore, we claim this sample includes a credible number of classes, extracted from real-world and non-trivial applications, with a consolidated number of users and a relevant history of bugs. Moreover, we considered seven metrics, covering major source code properties like size, coupling and cohesion. We only omitted inheritance-related metrics, like Depth of Inheritance Tree (DIT), because they did not present good results in our previous study (Couto et al., 2012). Despite these observations, we cannot guarantee—as usual in empirical software engineering—that our findings apply to other metrics or systems, specifically to systems implemented in other languages or

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033



(a) Alarms

(b) True Alarms

Fig. 15. Number of alarms and true alarms, for different threshold functions (Eclipse JDT Core).

Severity	Bugs		Predicted	
	#	%	#	%
Blocker	38	1	11	1
Critical	131	4	24	3
Major	311	8	61	7
Normal	3001	81	694	84
Minor	156	4	33	4
Trivial	60	2	4	0
Total	3697	100	827	100

(a) Eclipse JDT Core

Severity	Bugs		Predicted	
	#	%	#	%
Blocker	14	1	1	1
Critical	52	3	4	2
Major	115	6	9	5
Minor	77	4	5	3
Normal	1509	84	159	89
Trivial	31	2	1	1
Total	1798	100	179	100

(b) Eclipse PDE UI

Severity	Bugs		Predicted	
	#	%	#	%
Blocker	14	2	1	2
Critical	33	4	1	2
Major	55	7	4	7
Minor	12	2	-	-
Normal	661	84	48	89
Trivial	9	1	-	-
Total	784	100	54	100

(c) Equinox Framework

Severity	Bugs		Predicted	
	#	%	#	%
Blocker	3	1	-	-
Critical	6	2	-	-
Major	143	43	20	42
Minor	160	48	27	56
Normal	-	-	-	-
Trivial	23	7	1	2
Total	335	100	48	100

(d) Lucene

Fig. 16. Distribution of bugs by severity.

to systems from different domains, such as real-time systems and embedded systems.

Internal validity: This form of validity concerns to internal factors that may influence our observations. A possible internal validity threat concerns the procedure to identify the thresholds used by our model to trigger alarms. We rely on the average of the positive variations in the metric values to define such thresholds. We acknowledge that the use of the average in this case is not strictly recommended, because we never checked whether the positive variations follow a normal distribution. We tested other functions (median, 1st quartile, minimum, etc.) and they presented similar results than the average.

Construct validity: This form of validity concerns the relationship between theory and observation. A possible threat concerns the way we linked bugs to defects in classes. Particularly, we discarded bugs without explicit references in the textual description of the commits. However, the percentage of such bugs was not large, around 36% ((10,394–6614)/10,394) of the bugs considered in our evaluation. Moreover, this approach is commonly used in studies that involve mapping bugs to defects in classes (D'Ambros et al., 2010; Zimmermann et al., 2007; Śliwerski et al., 2005).

7. Related work

A recent systematic literature review identified 208 defect prediction studies—including some of the works presented in this section—published from January 2000 to December 2010 (Hall et al., 2012). The studies differ in terms of the software metrics used for prediction, the modeling technique, the granularity of the independent variable, and the validation technique. Typically, the independent variables can be classified into source code metrics, change metrics, bug finding tools, and code smells. The modeling techniques vary with respect to linear regression, logistic regression, naïve bayes, neural networks, etc. The granularity of the prediction can be at the method level, file/class level, or module/package level. The validation can be conducted using classification or ranking techniques. It is worth noting that neither of the 208 surveyed studies rely on causality tests as the underlying modeling technique. Usually, such studies emulate the lag concept by using larger evaluation intervals.

The defect prediction approaches we discuss in this section can be arranged in two groups: (a) source code metrics approaches and (b) process metrics approaches. Approaches based on source code metrics consider that the current design and structure of the program may influence the presence of future defects. On the other hand, approaches based on process metrics consider that information extracted from version control platforms such as code changes influence the occurrence of defects. In this section, approaches that use both source code and process metrics as independent variables appear only in one group, based on the best results they achieved. Finally, we have a third group that presents a single study on the application of the Granger test in software maintenance.

Source code metrics approaches: Basili et al. were one of the first to investigate the use of CK metrics as early predictors for fault-prone classes (Basili et al., 1996). In a study on eight medium-sized systems they report a correlation between CK metrics (with the exception of the NOC metric) and fault-prone classes. Subramanyam et al. later relied on the CK metrics to predict defect-prone components in an industrial application with subsystems implemented in C++and Java (Subramanyam and Krishnan, 2003). They concluded that the most useful metrics to predict defects may vary across these languages. For modules in C++, they report that WMC, DIT, and CBO with DIT had the most relevant impact on the number of defects. For the modules in Java, only CBO with DIT had an impact on defects.

Nagappan et al. conducted a study on five components of the Windows OS in order to investigate the relationship between complexity metrics and field defects (Nagappan et al., 2006). 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. As a consequence of this finding, the authors 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 using the project's history (Zimmermann et al., 2008).

Later, the study of Nagappan et al. was replicated by Holschuch et al. using a large ERP system (SAP R3) (Holschuh et al., 2009). However, both studies rely on linear regression models and correlation tests, which consider only an "immediate" relation between the independent and dependent variables. On the other hand, the dependency between bugs and source code metrics may not be immediate, i.e., usually exists a delay or lag in this dependency. In this paper, we presented a new approach for predicting bugs that considers this lag.

Process metrics approaches: D'Ambros et al. provided the original dataset with the historical values of the source code metrics we extended in this paper (D'Ambros et al., 2010, 2012). By making this dataset publicly available, their goal was to establish a common benchmark for comparing bug prediction approaches. They relied on this dataset to evaluate a representative set of prediction approaches reported in the literature, including approaches based on 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. Their results shows that churn and entropy of source code achieved a better adjusted R^2 and Spearman coefficient 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 their approach do not aim to trigger alarms as soon as risky changes (as captured by variations in the values of source code metrics) are applied to the classes of a target system. Instead the authors used the Spearman coefficient to evaluate the correlation between the rank of predicted defects and real defects. In other words, they do not evaluate precision and recall considering defects alarms.

Emanuel et al. proposed a defect prediction model at the method-level, using four classification methods: Random Forest (RndFor), Bayesian Network (BN), Support Vector Machine (SVM), and J48 Decision Tree (Giger et al., 2012). More specifically, the proposed model is used to identify defect-prone methods using 24 method level change and source code metrics. They performed an experiment using 21 open-source systems to assess the efficacy of the prediction models. The results indicated that the model based on RndFor reached a precision of 85% and a recall of 95%. However, they evaluate the models using a 10-fold cross-validation technique. On the other hand, cross-validation operates on a single time frame and therefore does not consider the temporal aspect. In this paper, we trained our prediction models using data from a time frame and validated them using data from future time frames.

Typically, defect prediction models are used to identify defect-prone methods, files, or packages. Kamei et al. proposed a new approach for defect prediction called "Just-In-Time Quality Assurance" that focus on identifying defect-prone software changes instead of methods, files or packages (Kamei et al., 2013). Based on logistic regression, the models they propose identify whether or not a change is defect-prone using change metrics, such as number of modified files, number of developers involved in the change, lines of code added and deleted, etc. They performed an empirical study with six open-source and five commercial systems to evaluate the performance of the models. The results showed an average precision of 34% and an average recall of 64%. Similar to the study of Emanuel et al., the models are not validated in future time frames.

Hassan and Holt's Top Ten List is an approach that highlights to managers the ten most fault-prone subsystems of a given software, based on the following heuristics: Most Frequently/Recently Modified, Most Frequently/Recently Fixed (Hassan and Holt, 2005). The goal is to provide guidance to maintainers, by suggesting they must invest their limited resources on the recommended subsystems. Similarly, our goal is to provide guidance to maintainers, but by triggering alarms when risky changes—according to Granger causality test—are applied to classes.

Application of Granger in software maintenance: Canfora et al. propose the use of the Granger test to detect change couplings, i.e., software artifacts that are frequently modified together (Canfora et al., 2010). They claim that conventional techniques to determine change couplings fail when the changes are not "immediate" but due to subsequent 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 a study 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 summary, our approach for defect prediction differs from the presented studies with respect to three aspects: (a) to the best of our knowledge, the existing defect prediction approaches do not consider the idea of causality between software metrics and defects. Differently, our approach relies on the Granger test to infer relationships between source code metrics and defects; (b) typically, most studies evaluate their models in a single time frame. In contrast, we evaluated our approach in several life stages of the considered systems; and (c) unlike common approaches for defect prediction, the models we propose do not aim to predict the number of defects of a class in a future time frame. Instead, our models trigger alarms that indicate changes to a class that have more chances to generate defects. However, we acknowledge that other defect prediction techniques can also be extended to include such alarms.

8. Conclusions

In this paper, we described and evaluated an approach for predicting defects using causality tests. In contrast with other works on defect prediction, our approach does not aim to predict the number or the existence of defects in a class in a future time frame. Alternatively, we proposed a model that predicts defects as soon as they are introduced in the source code. More specifically, we rely on input from the Granger test to trigger alarms whenever a change performed to a class reproduces similar variations in the class' source code properties that in the past caused defects. Our approach reached an average precision greater than 50% in several life stages of three out of the four systems we evaluate. Furthermore, by comparing our approach with baselines that are not based on causality tests, it achieved a better precision.

As further work, we plan to design and implement a tool supporting the defect prediction model proposed in this paper. We plan to implement this tool as a plug-in for version control platforms, like SVN and Git. Basically, this tool should trigger alarms whenever risky changes are committed to the version repository. Based on such alarms, the maintainer can for example perform software quality assurance activities (e.g., testing or code inspections) before executing the commit. In addition, we plan to extend the proposed defect prediction model to handle the cases where changes in a class cause defects in other classes of the system. Finally, another future work includes a qualitative analysis on why some defects can be predicted and others not, which certainly requires a direct participation of expert developers on the target systems.

The dataset with the time series of source code metrics and defects used in this paper is publicly available at: http://aserg.labsoft.dcc.ufmg.br/jss2013.

Acknowledgements

This work was supported by FAPEMIG, CAPES, and CNPq.

References

Araujo, J.E., Souza, S., Valente, M.T., 2011. Study on the relevance of the warnings reported by Java bug finding tools. IET Software 5 (4), 366–374.
Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22 (10), 751–761.
Canfora, G., Ceccarelli, M., Di Penta, M., Cerulo, L., 2010. Using multivariate time series and association rules to detect logical change coupling: an empirical study. In: 26th International Conference on Software Maintenance (ICSM), pp. 1–10.
Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.
Couto, C., Araujo, J.E., Silva, C., Valente, M.T., 2013. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. Software Quality Journal 21 (2), 241–257.
Couto, C., Silva, C., Valente, M.T., Bigonha, R., Anquetil, N., 2012. Uncovering causal relationships between software metrics and bugs. In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 223–232.
D'Ambros, M., Bacchelli, A., Lanza, M., 2010. On the impact of design flaws on software defects. In: 10th International Conference on Quality Software (QSIC), pp. 23–31.
D'Ambros, M., Lanza, M., Robbes, R., 2010. An extensive comparison of bug prediction approaches. In: 7th Working Conference on Mining Software Repositories (MSR), pp. 31–41.
D'Ambros, M., Lanza, M., Robbes, R., 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. Journal of Empirical Software Engineering 17 (4-5), 531–577.
Fuller, W.A., 1994. Introduction to Statistical Time Series. John Wiley & Sons, United States of America.
Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C., 2012. Method-level bug prediction. In: 6th International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 171–180.
Granger, C., 1969. Investigating causal relations by econometric models and cross-spectral methods. Econometrica 37 (3), 424–438.
Granger, C., 1981. Some properties of time series data and their use in econometric model specification. Journal of Econometrics 16 (6), 121–130.
Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Transactions on Software Engineering 38 (6), 1276–1304.
Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 31st International Conference on Software Engineering (ICSE), pp. 78–88.
Hassan, A.E., Holt, R.C., 2005. The top ten list: dynamic fault prediction. In: 21st International Conference on Software Maintenance (ICSM), pp. 263–272.
Holschuh, T., Pauser, M., Herzig, K., Zimmermann, T., Premraj, R., Zeller, A., 2009. Predicting defects in SAP Java code: an experience report. In: 31st International Conference on Software Engineering (ICSE), pp. 172–181.
Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N., 2013. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering 39 (6), 757–773.
Lanza, M., 2001. The evolution matrix: recovering software evolution using software visualization techniques. In: 4th International Workshop on Principles of Software Evolution (IWPSE), pp. 37–42.
Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. Proceedings of the IEEE 68 (9), 1060–1076.
Mukhopadhyay, N.D., Chatterjee, S., 2007. Causality and pathway search in microarray time series experiment. Bioinformatics 23 (4), 442–449.
Nagappan, N., Ball, T., 2005. Static analysis tools as early indicators of pre-release defect density. In: 27th International Conference on Software Engineering (ICSE), pp. 580–586.
Nagappan, N., Ball, T., Zeller, A., 2006. Mining metrics to predict component failures. In: 28th International Conference on Software Engineering (ICSE), pp. 452–461.
Perry, D.E., Porter, A.A., Votta, L.G., 1997. A primer on empirical studies (tutorial). In: Tutorial presented at 19th International Conference on Software Engineering (ICSE), pp. 657–658.
Schein, A.I., Popescul, A., Ungar, L.H., Pennock, D.M., 2002. Methods and metrics for cold-start recommendations. In: 25th International Conference on Research and Development in Information Retrieval (SIGIR), pp. 253–260.
Schröter, A., Zimmermann, T., Zeller, A., 2006. Predicting component failures at design time. In: 5th International Symposium on Empirical Software Engineering (ISESE), pp. 18–27.
Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? In: 2nd Working Conference on Mining Software Repositories (MSR), pp. 1–5.
IEEE Standard IEEE Standard Glossary of Software Engineering Terminology. IEEE Std. 610.12, 1990.
Subramanyam, R., Krishnan, M.S., 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Transaction on Software Engineering 29 (4), 297–310.
Zimmermann, T., Nagappan, N., Zeller, A., 2008. Predicting Bugs from History. Springer, pp. 69–88, chapter 4.
Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for eclipse. In: 3rd International Workshop on Predictor Models in Software Engineering, p. 9.

Please cite this article in press as: Couto, C., et al., Predicting software defects with causality tests. J. Syst. Software (2014), http://dx.doi.org/10.1016/j.jss.2014.01.033