An Extensive Comparison of Bug Prediction Approaches

Marco D'Ambros, Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano, Switzerland

Romain Robbes
Computer Science Department (DCC)
University of Chile, Santiago, Chile

Abstract—**Reliably predicting software defects is one of software engineering's holy grails. Researchers have devised and implemented a plethora of bug prediction approaches varying in terms of accuracy, complexity and the input data they require. However, the absence of an established benchmark makes it hard, if not impossible, to compare approaches.**

We present a benchmark for defect prediction, in the form of a publicly available data set consisting of several software systems, and provide an extensive comparison of the explanative and predictive power of well-known bug prediction approaches, together with novel approaches we devised.

Based on the results, we discuss the performance and stability of the approaches with respect to our benchmark and deduce a number of insights on bug prediction models.

I. Introduction

Defect prediction has generated widespread interest for a considerable period of time. The driving scenario is resource allocation: Time and manpower being finite resources, it makes sense to assign personnel and/or resources to areas of a software system with a higher probable quantity of bugs.

A variety of approaches have been proposed to tackle the problem, relying on diverse information, such as code metrics [1]–[8] (lines of code, complexity), process metrics [9]–[12] (number of changes, recent activity) or previous defects [13]–[15]. The jury is still out on the relative performance of these approaches. Most of them have been evaluated in isolation, or were compared to only few other approaches. Moreover, a significant portion of the evaluations cannot be reproduced since the data used by them came from commercial systems and is not available for public consumption. As a consequence, articles reached opposite conclusions: For example, in the case of size metrics, Gyimothy *et al.* reported good results [6] unlike Fenton *et al.* [16].

What is missing is a baseline against which the approaches can be compared. We provide such a baseline by gathering an extensive dataset composed of several open-source systems. Our dataset contains the information required to evaluate several approaches across the bug prediction spectrum on a number of systems large enough to have confidence in the results. The contributions of this paper are:

- A public benchmark for defect prediction, containing enough data to evaluate several approaches. For five open-source software systems, we provide, over a five-year period, the following data: (1) process metrics on all the files of each system, (2) system metrics on bi-weekly versions of each system, (3) defect information related to each system file, and (4) bi-weekly models of each system version if new metrics need to be computed.
- The evaluation of a representative selection of defect prediction approaches from the literature.
- Two novel bug prediction approaches based on bi-weekly samples of the source code. The first measures code churn as deltas of source code metrics instead of line-based code churn. The second extends Hassan's concept of entropy of changes [10] to source code metrics. These techniques provide the best and most stable prediction results in our comparison.

Structure of the paper: In Section II we present an overview of related work in defect prediction. We describe our benchmark and evaluation procedure in Section III. In Section IV, we detail the approaches that we reproduce and the ones that we introduce. We report on their performance in Section V. In Section VI, we discuss possible threats to the validity of our findings, and we conclude in Section VII.

II. Defect Prediction

We describe several approaches to defect prediction, the kind of data they require and the various data sets on which they were validated. All approaches require a defect archive to be validated, but do not necessarily require it to actually perform their analysis. When they do, we indicate it.

Change Log Approaches use information extracted from the versioning system, assuming that recently or frequently changed files are the most probable source of future bugs.

Nagappan and Ball performed a study on the influence of code churn (*i.e.,* the amount of change to the system) on the defect density in Windows Server 2003. They found that relative code churn was a better predictor than absolute churn [9]. Hassan introduced the entropy of changes, a measure of the complexity of code changes [10]. Entropy was compared to amount of changes and the amount of previous bugs, and was found to be often better. The entropy metric was evaluated on six open-source systems: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. Moser *et al.* used metrics (including code churn, past bugs and refactorings, number of authors, file size and age, *etc.*), to predict the presence/absence of bugs in files of Eclipse [11].

The mentioned techniques do not make use of the defect archives to predict bugs, while the following ones do.

Hassan and Holt's top ten list approach validates heuristics about the defect-proneness of the most and most recently changed and bug-fixed files, using the defect repository data [15]. The approach was validated on six open-source case studies: FreeBSD, NetBSD, OpenBSD, KDE, KOffice, and PostgreSQL. They found that recently modified and fixed entities were the most defect-prone. Ostrand *et al.* predict faults on two industrial systems, using change and defect data [14]. The bug cache approach by Kim *et al.* uses the same properties of recent changes and defects as the top ten list approach, but further assumes that faults occur in bursts [13]. The bug-introducing changes are identified from the SCM logs. Seven open-source systems were used to validate the findings (Apache, PostgreSQL, Subversion, Mozilla, JEdit, Columba, and Eclipse). Bernstein *et al.* use bug and change information in non-linear prediction models [12]. Six eclipse plugins were used to validate the approach.

Single-version approaches assume that the current design and behavior of the program influences the presence of future defects. These approaches do not require the history of the system, but analyze its current state in more detail, using a variety of metrics. One standard set of metrics used is the Chidamber and Kemerer (CK) metrics suite [17].

Basili *et al.* used the CK metrics on eight medium-sized information management systems based on the same requirements [1]. Ohlsson *et al.* used several graph metrics including McCabe's cyclomatic complexity on an Ericsson telecom system [2]. El Emam *et al.* used the CK metrics in conjunction with Briand's coupling metrics [3] to predict faults on a commercial Java system [4]. Subramanyam *et al.* used CK metrics on a commercial C++/Java system [5]; Gyimothy *et al.* performed a similar analysis on Mozilla [6]. Nagappan and Ball estimated the pre-release defect density of Windows Server 2003 with a static analysis tool [7]. Nagappan *et al.* used a catalog of source code metrics to predict post release defects at the module level on five Microsoft systems, and found that it was possible to build predictors for one individual project, but that no predictor would perform well on all the projects [8]. Zimmermann *et al.* applied a number of code metrics on Eclipse [18].

Other Approaches. Zimmermann and Nagappan used dependencies between binaries in Windows server 2003 to predict defect [19]. Marcus *et al.* used a cohesion measurement based on LSI for defect prediction on several C++ systems, including Mozilla [20]. Neuhaus *et al.* used a variety of features of Mozilla (past bugs, package imports, call structure) to detect vulnerabilities [21].

Observations We observe that both *case studies* and the *granularity of approaches* vary. Varying case studies make a comparative evaluation of the results difficult. Validations performed on industrial systems are not reproducible, because it is not possible to obtain the data that was used. There is also some variation among open-source case studies, as some approaches have more restrictive requirements than others. With respect to the granularity of the approaches, some of them predict defects at the class level, others consider files, while others consider modules or directories (subsystems), or even binaries. While some approaches predict the presence or absence of bugs for each component, others predict the amount of bugs affecting each component in the future, producing a ranked list of components.

These observations explain the lack of comparison between approaches and the occasional diverging results when comparisons are performed. In the following, we present a benchmark to establish a common ground for comparison.

III. Experiments

We compare different bug prediction approaches in the following way: *Given a release* x *of a software system* s*, released at date* d*, the task is to predict, for each class of* x*, the number of post release defects, i.e., the number of defects reported from* d *to six months later.* We chose the last release of the system in the release period and perform class-level defect prediction, and not package- or subsystem-level defect prediction, for the following reasons:

- Predictions at the package-level are less helpful since packages are significantly larger. The review of a defect-prone package requires more work than a class.
- Classes are the building blocks of object-oriented systems, and are self-contained elements from the point of view of design and implementation.
- Package-level information can be derived from class-level information, while the opposite is not true.

We predict the number of bugs in each class –not the presence/absence of bugs– as this better fits the resource allocation scenario, where we want an ordered list of classes. We use post-release defects for validation (*i.e.,* not all defects in the history) to emulate a real-life scenario. As in [18] we use a six months time interval for post-release defects.

A. Benchmark Dataset

Our dataset is composed of the change, bug and version information of the five systems detailed in Figure 1.

We provide, for each system: The data extracted from the change log, including reconstructed transaction and links from transactions to model classes; The defects extracted from the defect repository, linked to the transactions and the system classes referencing them; Biweekly versions of the systems parsed into object-oriented models; Values of all the metrics used as predictors, for each version of each class of the system; And post-release defect counts for each class. All systems are written in Java to ensure that all the code metrics are defined identically for each system. By using the same parser, we can avoid issues due to behavior differences in parsing, a known issue for reverse engineering tools [22].

System	Prediction release	Time period	#Classes	#Versions	#Transactions	#Post-rel. defects
Eclipse JDT Core www.eclipse.org/jdt/core/	3.4	1.1.2005 - 6.17.2008	997	91	9,135	463
Eclipse PDE UI www.eclipse.org/pde/pde-ui/	3.4.1	1.1.2005 - 9.11.2008	1,562	97	5,026	401
Equinox framework www.eclipse.org/equinox/	3.4	1.1.2005 - 6.25.2008	439	91	1,616	279
Mylyn www.eclipse.org/mylyn/	3.1	1.17.2005 - 3.17.2009	2,196	98	9,189	677
Apache Lucene lucene.apache.org	2.4.0	1.1.2005 - 10.8.2008	691	99	1,715	103

Figure 1. Systems in the benchmark.

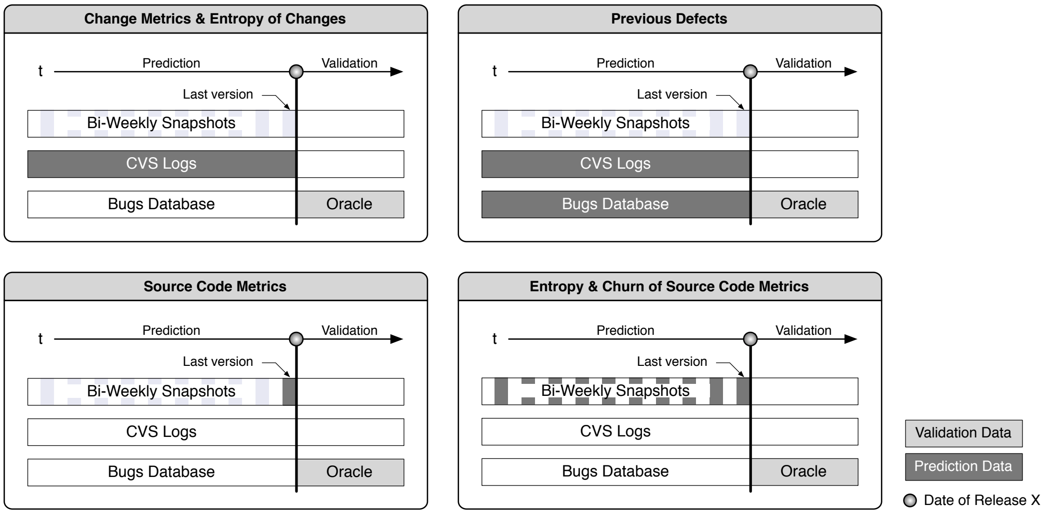

Figure 2. The types of data used by different bug prediction approaches.

Data Collection. Figure 2 shows the types of information needed by the compared bug prediction approaches.

We need the following information: (1) change log information to extract process metrics; (2) source code version information to compute source code metrics; and (3) defect information linked to classes for both the prediction and validation. Figure 3 shows how we gather this information, given an SCM system (CVS/Subversion) and a defect tracking system (Bugzilla/Jira).

Creating a History Model. To compute the various process metrics, we model how the system changed during its lifetime by parsing the versioning system log files. We create a model of the history of the system using the transactions extracted from the system's SCM repository. A transaction (or commit) is a set of files which were modified and committed to the repository, together with the timestamp, the author and the comment. SVN marks co-changing files at commit time as belonging to the same transaction while for CVS we infer transactions from each file's modification time, commit comment, and author.

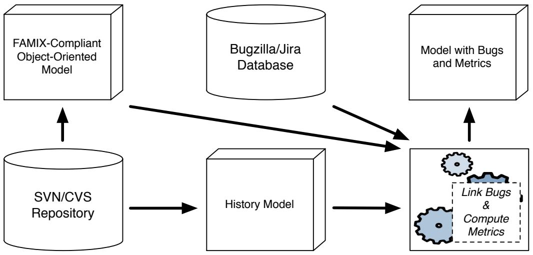
FAMIX-Compliant Object-Oriented Model
Bugzilla/Jira Database
Model with Bugs and Metrics
SVN/CVS Repository
History Model
Link Bugs & Compute Metrics

Figure 3. Model with bug, change and history.

Creating a Source Code Model. We retrieve the source code from the SCM repository and we extract an object-oriented model of it according to FAMIX, a language independent meta-model of object oriented code [23]. Since we need several versions of the system, we repeat this process at bi-weekly intervals over the history period we consider.

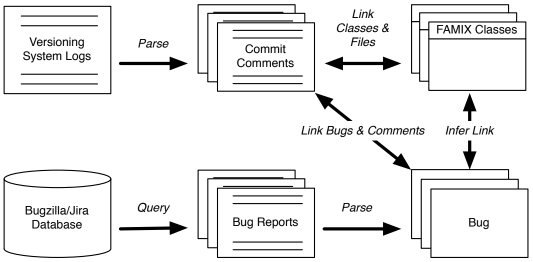

Figure 4. Linking bugs, SCM files and classes.

Linking Classes with Bugs. To reason about the presence of bugs affecting parts of the software system, we first map each problem report to the components of the system that it affects. We link FAMIX classes with versioning system files and bugs retrieved from Bugzilla and Jira repositories, as shown in Figure 4. A file version in the versioning system contains a developer comment written at commit time, which often includes a reference to a problem report (*e.g.,* "fixed bug 123"). Such references allow us to link problem reports with files in the versioning system (and thus with classes). However, the link between a CVS/SVN file and a Bugzilla/Jira problem report is not formally defined: We use pattern matching to extract a list of bug id candidates [18], [24]. Then, for each bug id, we check whether a bug with such an id exists in the bug database and, if so, we retrieve it. Finally we verify the consistency of timestamps, *i.e.,* we reject any bug whose report date is after the commit date.

Due to the file-based nature of SVN and CVS and to the fact that Java inner classes are defined in the same file as their containing class, several classes might point to the same CVS/SVN file, *i.e.,* a bug linking to a file version might be linking to more than one class. We are not aware of a workaround for this problem, which in fact is a shortcoming of the versioning system. For this reason, we do *not* consider inner classes. We also filter out test classes from our dataset.

Computing Metrics. At this point we have a model including source code information over several versions, change history, and defects data. The last step is to enrich it with the metrics we want to evaluate. We describe the metrics as they are introduced with each approach.

Tools. To create our dataset, we use the following tools:

- *inFusion* (developed by the company intooitus in Java and available at http://www.intooitus.com/) to convert Java source code to FAMIX models.
- *Moose* (developed in Smalltalk and available at http://www.moosetechnology.org) to read FAMIX models and to compute a number of source code metrics.
- *Churrasco* (developed in Smalltalk and available at http://churrasco.inf.usi.ch) to create the history model, to extract bug data and to link classes, versioning system files and bugs.

B. Evaluating the Approaches

To compare bug prediction approaches we apply them on the same software systems and, for each system, on the same data set. We consider the last major releases of the software systems and compute the predictors up to the releases dates.

We base our predictions on generalized linear regression models built from the metrics we computed. The independent variables (used in the prediction) are the set of metrics under study for each class, while the dependent variable (the predicted one) is the number of post-release defects. Following the method proposed by Nagappan *et al.* [8], we perform principal component analysis, build regression models, and evaluate explanative and predictive power.

Principal Component Analysis (PCA) [25] avoids the problem of multicollinearity among the independent variables. This problem comes from intercorrelations amongst these variables and can lead to an inflated variance in the estimation of the dependent variable. We do not build the regression models using the actual variables (*e.g.,* metrics) as independent variables, but instead we use sets of principal components (PC), which are independent and therefore do not suffer from multicollinearity, while at the same time they account for as much sample variance as possible. We select PCs that account for at least 95% of the variance.

Building Regression Models. We do cross validation, *i.e.,* we use 90% of the dataset (90% of the classes – training set) to build the prediction model, and the remaining 10% (validation set) to evaluate the accuracy of the model. For each model we perform 50 folds, *i.e.,* we create 50 random 90%-10% splits of the data.

Evaluating Explanative Power. We use the adjusted R^2 coefficient. The (non-adjusted) R^2 is the ratio of the regression sum of squares to the total sum of squares. It ranges from 0 to 1, and quantifies the variability in the data explained by the model. The adjusted R^2, accounts for degrees of freedom of the independent variables and the sample population; it is consistenly lower than R^2. When reporting results, we only mention the adjusted R^2. We test the statistical significance of the regression models using the F-test (99% significance, $p < 0.01$).

Evaluating Predictive Power. We compute the Spearman correlation between the predicted number of defects and the actual number. The Spearman correlation is computed on two lists (classes ordered by actual number of bugs and classes ordered by number of predicted bugs) and is an indicator of the similarity of their order. We decided to measure the correlation with the Spearman coefficient (instead of, for example, the Pearson coefficient), as it is recommended with skewed data (which is the case here, as most classes have no bugs).We compute the Spearman on the validation set, which is 10% of the original dataset. Since we perform 50 folds cross validation, the final values of the Spearman and adjusted R^2 are averages over 50 folds.

IV. Bug Prediction Approaches

Table I summarizes the bug prediction approaches that we compare. In the following we detail each approach.

Type	Rationale	Used by
Change metrics	Bugs are caused by changes.	Moser [11]
Previous defects	Past defects predict future defects.	Kim [13]
Source code metrics	Complex components are harder to change, and hence error-prone.	Basili [1]
Entropy of changes	Complex changes are more error-prone than simpler ones.	Hassan [10]
Churn (source code metrics)	Source code metrics are a better approximation of code churn.	Novel
Entropy (source code metrics)	Source code metrics better describe the entropy of changes.	Novel

Table I
Categories of bug prediction approaches.

A. Change Metrics

We selected the approach of Moser *et al.* as a representative, and describe three additional variants.

MOSER. We use the catalog of file-level change metrics introduced by Moser *et al.* [11] listed in Table II. The metric NFIX represents the number of bug fixes as extracted from the versioning system, not the defect archive. It uses a heuristic based on pattern matching on the comments of every commit. To be recognized as a bug fix, the comment must match the string "%fix%" and not match the strings "%prefix%" and "%postfix%". The bug repository is not needed, because all the metrics are extracted from the CVS/SVN logs, thus simplifying data extraction. For systems versioned using SVN (such as Lucene) we perform some additional data extraction, since the SVN logs do not contain information about lines added and removed.

NR	Number of revisions
NREF	Number of times file has been refactored
NFIX	Number of times file was involved in bug-fixing
NAUTH	Number of authors who committed the file
LINES	Lines added and removed (sum, max, average)
CHURN	Codechurn (sum, maximum and average)
CHGSET	Change set size (maximum and average)
AGE	Age and weighted age

Table II
Change metrics used by Moser *et al.*

NFIX: Zimmermann *et al.* showed that the number of past defects has the highest correlation with number of future defects [18]. We inspect the accuracy of the bug fix approximation in isolation.

NR: In the same fashion, since Graves *et al.* showed that the best generalized linear models for defect prediction are based on number of changes [26], we isolate the number of revisions as a predictive variable.

NFIX+NR: We combine the previous two approaches.

B. Previous Defects

This approach relies on a single metric to perform its prediction. We also describe a more fine-grained variant exploiting the categories present in defect archives.

BUGFIXES. The bug prediction approach based on previous defects, proposed by Zimmermann *et al.* [18], states that the number of past bug fixes extracted from the repository is correlated with the number of future fixes. They then use this metric in the set of metrics with which they predict future defects. This measure is different from the metric used in **NFIX-ONLY** and **NFIX+NR**: For **NFIX**, we perform pattern matching on the commit comments. For **BUGFIXES**, we also perform the pattern matching, which in this case produces a list of potential defects. Using the defect id, we check whether the bug exists in the bug database, we retrieve it and we verify the consistency of timestamps (*i.e.,* if the bug was reported before being fixed).

Variant: BUG-CATEGORIES. We also use a variant in which as predictors we use the number of bugs belonging to five categories, according to severity and priority. The categories are: All bugs, non trivial bugs (severity>trivial), major bugs (severity>major), critical bugs (critical or blocker severity) and high priority bugs (priority>default).

C. Source Code Metrics

Many approaches in the literature use the CK metrics. We compare them with additional object-oriented metrics, and LOC. Table III lists all source code metrics we use.

Type	Metric	
CK	WMC	Weighted Method Count
CK	DIT	Depth of Inheritance Tree
CK	RFC	Response For Class
CK	NOC	Number Of Children
CK	CBO	Coupling Between Objects
CK	LCOM	Lack of Cohesion in Methods
OO	FanIn	Number of other classes that reference the class
OO	FanOut	Number of other classes referenced by the class
OO	NOA	Number of attributes
OO	NOPA	Number of public attributes
OO	NOPRA	Number of private attributes
OO	NOAI	Number of attributes inherited
OO	LOC	Number of lines of code
OO	NOM	Number of methods
OO	NOPM	Number of public methods
OO	NOPRM	Number of private methods
OO	NOMI	Number of methods inherited

Table III
Class level source code metrics.

CK. Many bug prediction approaches are based on metrics, in particular the Chidamber & Kemerer suite [17].

OO. An additional set of object-oriented metrics.

CK+OO. The combination of the two sets of metrics.

LOC. Gyimothy *et al.* showed that lines of code (**LOC**) is one of the best metrics for fault prediction [6]. We treat it as a separate predictor.

D. Entropy of Changes

Hassan predicts defects using the entropy (or complexity) of code changes [10]. The idea consists in measuring, over a time interval, how distributed changes are in a system. The more spread, the higher is the complexity. The intuition is that one change affecting one file only is simpler than one affecting many different files, as the developer who has to perform the change has to keep track of all of them. Hassan proposed to use the Shannon Entropy defined as

$$H_n(P) = -\sum_{k=1}^{n} p_k * \log_2 p_k \qquad (1)$$

where p_k is the probability that the file k changes during the considered time interval. Figure 5 shows an example with three files and three time intervals.

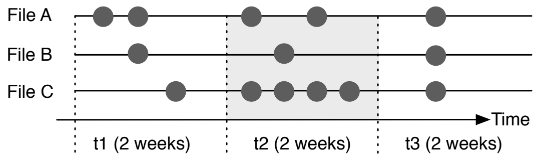

Figure 5. An example of entropy of code changes.

In the fist time interval $t1$, we have a total of four changes, and the change frequencies of the files (*i.e.,* their probability of change) are $p_A = \frac{2}{4}, p_B = \frac{1}{4}, p_C = \frac{1}{4}$. The entropy in $t1$ is therefore $H = -(0.5 * \log_2 0.5 + 0.25 * \log_2 0.25 + 0.25 * \log_2 0.25) = 1$. In $t2$, the entropy is higher: $H = -(\frac{2}{7} * \log_2 \frac{2}{7} + \frac{1}{7} * \log_2 \frac{1}{7} + \frac{4}{7} * \log_2 \frac{4}{7}) = 1.378$. As in [10], to compute the probability that a file changes, instead of simply using the number of changes, we take into account the amount of change by measuring the number of modified lines (lines added plus deleted) during the time interval. Hassan defined the Adaptive Sizing Entropy as:

$$H' = -\sum_{k=1}^{n} p_k * \log_{\bar{n}} p_k \qquad (2)$$

where n is the number of files in the system and $\bar{n}$ is the number of recently modified files. To compute the set of recently modified files we use previous periods (*e.g.,* modified in the last six time intervals). To use the entropy of code change as a bug predictor, Hassan defined the History of Complexity Metric (HCM) of a file j as

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} HCPF_i(j) \qquad (3)$$

where $\{a,..,b\}$ is a set of evolution periods and $HCPF$ is:

$$HCPF_i(j) = \begin{cases} c_{ij} * H'_i, & j \in F_i \\ 0, & \text{otherwise} \end{cases} \qquad (4)$$

where i is a period with entropy H'_i, F_i is the set of files modified in the period i and j is a file belonging to F_i. According to the definition of c_{ij}, we test two metrics:

- **HCM:** $c_{ij} = 1$, every file modified in the considered period i gets the entropy of the system in the considered time interval.
- **WHCM:** $c_{ij} = p_j$, each modified file gets the entropy of the system weighted with the probability of the file being modified.

Concerning the periods used for computing the History of Complexity Metric, we use two weeks time intervals.

Variants. We define three further variants based on **HCM**, with an additional weight for periods in the past. In **EDHCM** (Exponentially Decayed HCM, introduced by Hassan), entropies for earlier periods of time, *i.e.,* earlier modifications, have their contribution exponentially reduced over time, modelling an exponential decay model. Similarly, **LDHCM** (Linearly Decayed) and **LGDHCM** (LoGarithmically decayed), have their contributions reduced over time in a respectively linear and logarithmic fashion. Both are novel. The definition of the variants follows (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{e^{\phi_1 * (|\{a,..,b\}| - i)}} \qquad (5)$$

$$LDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{\phi_2 * (|\{a,..,b\}| + 1 - i)} \qquad (6)$$

$$LGDHCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} \frac{HCPF_i(j)}{\phi_3 * \ln(|\{a,..,b\}| + 1.01 - i)} \qquad (7)$$

E. Churn of Source Code Metrics

Using churn of source code metrics to predict post release defects is novel. The intuition is that higher-level metrics may better model code churn than simple metrics like addition and deletion of lines of code. We sample the history of the source code every two weeks and compute the deltas of source code metrics for each consecutive pair of samples.

For each source code metric, we create a matrix where the rows are the classes, the columns are the sampled versions, and each cell is the value of the metric for the given class at the given version. If a class does not exist in a version, we indicate that by using a default value of -1. We only consider the classes which exist at release x for the prediction.

We generate a matrix of deltas, where each cell is the absolute value of the difference between the values of a metric –for a class– in two subsequent versions. If the class does not exist in one or both of the versions (at least one value is -1), then the delta is also -1.

Figure 6 shows an example of deltas matrix computation for three classes. The numbers in the squares are metrics; the numbers in circles, deltas. After computing the deltas

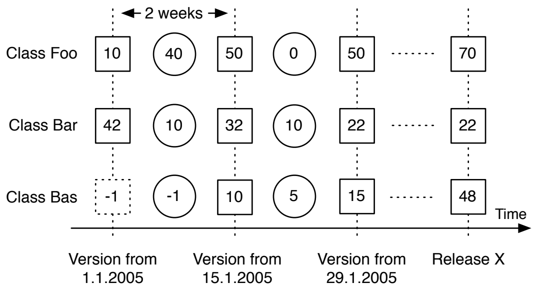

Figure 6. Computing metrics deltas from sampled versions of a system.

matrices for each source code metric, we compute churn as:

$$CHU(i) = \sum_{j=1}^{C} \begin{cases} 0, & deltas(i,j) = -1 \\ PCHU(i,j), & \text{otherwise} \end{cases} \tag{8}$$

$$PCHU(i,j) = deltas(i,j) \tag{9}$$

where i is the index of a row in the deltas matrix (corresponding to a class), C is the number of columns of the matrix (corresponding to the number of samples considered), $deltas(i,j)$ is the value of the matrix at position (i,j) and $PCHU$ stands for partial churn. For each class, we sum all the cells over the columns –excluding the ones with the default value of -1. In this fashion we obtain a set of churns of source code metrics at the class level, which we use as predictors of post release defects.

Variants. We define several variants of the partial churn of source code metrics (PCHU): The first one weights more the frequency of change (*i.e.,* delta > 0) than the actual change (the delta value). We call it **WCHU** (weighted churn), using the following partial churn:

$$WPCHU(i,j) = 1 + \alpha * deltas(i,j) \tag{10}$$

where α is the weight factor, set to 0.01 in our experiments. This avoids that a delta of 10 in a metric has the same impact on the churn as ten deltas of 1. We consider many small changes more relevant than few big changes. Other variants are based on weighted churn (**WCHU**) and take into account the decay of deltas over time, respectively in an exponential (**EDCHU**), linear (**LDCHU**) and logarithmic manner (**LGDCHU**), with these partial churns (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDPCHU(i,j) = \frac{1+\alpha*deltas(i,j)}{e^{\phi_1*(C-j)}} \tag{11}$$

$$LDPCHU(i,j) = \frac{1+\alpha*deltas(i,j)}{\phi_2*(C+1-j)} \tag{12}$$

$$LGDPCHU(i,j) = \frac{1+\alpha*deltas(i,j)}{\phi_3*\ln(C+1.01-j)} \tag{13}$$

F. Entropy of Source Code Metrics

In the last bug prediction approach we extend the concept of code change entropy [10] to the source code metrics listed in Table III. The idea is to measure the complexity of the variants of a metric over subsequent sample versions. The more distributed over multiple classes the variants of the metric is, the higher the complexity. For example, if in the system the WMC changed by 100, and only one class is involved, the entropy is minimum, whereas if 10 classes are involved with a local change of 10 WMC, then the entropy is higher. To compute the entropy of source code metrics, we start from the matrices of deltas computed as for the churn metrics. We define the entropy, for instance for WMC, for the column j of the deltas matrix, *i.e.,* the entropy between two subsequent sampled versions of the system, as:

$$H'_{WMC}(j) = -\sum_{i=1}^{R} \begin{cases} 0, & deltas(i,j) = -1 \\ p(i,j) * \log_{\bar{R}_j} p(i,j), & \text{otherwise} \end{cases} \tag{14}$$

where R is the number of rows of the matrix, $\bar{R}_j$ is the number of cells of the column j greater than 0 and $p(i,j)$ is a measure of the frequency of change (viewing frequency as a measure of probability, similarly to Hassan) of the class i, for the given source code metric. We define it as:

$$p(i,j) = \frac{deltas(i,j)}{\sum_{k=1}^{R} \begin{cases} 0, & deltas(k,j) = -1 \\ deltas(k,j), & \text{otherwise} \end{cases}} \tag{15}$$

Equation 14 defines an adaptive sizing entropy, because we use $\bar{R}_j$ for the logarithm, instead of R (number of cells greater than 0 instead of number of cells). In the example in Figure 6 the entropy for the first column is $-(\frac{40}{50} * \log_2 \frac{40}{50} + \frac{10}{50} * \log_2 \frac{10}{50}) = 0.722$, while for the second column it is $-(\frac{10}{15} * \log_2 \frac{10}{15} + \frac{5}{15} * \log_2 \frac{5}{15}) = 0.918$.

Given a metric, for example WMC, and a class corresponding to a row i in the deltas matrix, we define the history of entropy as:

$$HH_{WMC}(i) = \sum_{j=1}^{C} \begin{cases} 0, & deltas(k,j) = -1 \\ PHH_{WMC}(i,j), & \text{otherwise} \end{cases} \tag{16}$$

$$PHH_{WMC}(i,j) = H'_{WMC}(j) \tag{17}$$

where PHH stands for partial historical entropy.

Compared to the entropy of changes, the entropy of source code metrics has the advantage that it is defined for every considered source code metric. If we consider "lines of code", the two metrics are very similar: HCM has the benefit that it is not sampled, *i.e.,* it captures all changes recorded in the versioning system, whereas HH_{LOC}, being sampled, might lose precision. On the other hand, HH_{LOC} is more precise, as it measures the real number of lines of code (by parsing the source code), while HCM measures it from the change log, including comments and whitespace.

Variants. In Equation 17 each class that changes between two version (delta greater than 0) gets the entire system entropy. To take into account also how much the class changed, we define the history of weighted entropy HWH, by redefining PHH as:

$$HWH(i,j) = p(i,j) * H'(j) \tag{18}$$

We also define three other variants by considering the decay of the entropy over time, as for the churn metrics, in an exponential ($EDHH$), linear ($LDHH$), and logarithmic ($LGDHH$) fashion. We define their partial historical entropy as (ϕ_1, ϕ_2 and ϕ_3 are the decay factors):

$$EDHH(i,j) = \frac{H'(j)}{e^{\phi_1 * (C-j)}} \tag{19}$$

$$LDHH(i,j) = \frac{H'(j)}{\phi_2 * (C+1-j)} \tag{20}$$

$$LGDHH(i,j) = \frac{H'(j)}{\phi_3 * \ln(C+1.01-j)} \tag{21}$$

From these definitions, we define several prediction models using several object-oriented metrics: **HH**, **HWH**, **EDHHK**, **LDHH** and **LGDHH**.

V. Results

In Table IV, we report the results of each approach on each case study, in terms of explanative power (adjusted R^2), and predictive power (Spearman's correlation).

We also compute an overall score in the following way: For each case study, add three to the score if the R^2 or Spearman is within 90% of the best value, 1 if it is between 75–90%, and subtract one when it is less than 50%. We use this score, rather than an average of the values, to promote consistency: An approach performing very well on a case study, but bad on others will be penalized. We use the same criteria to highlight the results in Table IV: R^2 and Spearman within 90% of the best value are bolded, the ones within 75% have a dark gray background, while values less than 50% of the best have a light gray background. Scores of 10 or more denote good overall performance; they are underlined.

A general observation is the discrepancy between the R^2 score and the Spearman score for entropy approaches (HCM–LGDHCM): This is because HCM and its variations are based on a single metric, the number of changes, hence it explains a comparatively smaller portion of the variance, despite performing well. Based on the results in the table, we answer several questions.

What is the overall best performing approach? If we do not consider the amount of data needed to compute the metrics and instead compare absolute predictive power, we can infer the following: The best classes of metrics on all the data sets are the churn and the entropy of source code, with **WCHU** and **LDHH** in particular scoring most of the times in the top 90% in prediction, and **WCHU** having also a good and stable explanative power. Then the previous defects approaches, **BUGFIXES** and **BUGCAT** follow. Next comes the single-version code metrics **CK+OO**, followed by the entropy of changes (**WHCM**) and change metrics (**MOSER**).

Approaches based on churn and entropy of source code metrics have good and stable explanative and predictive power, better than all the other applied approaches.

What is the best approach, data-wise? If we take into account the amount of data and computational power needed, one might argue that downloading and parsing several versions of the source code is a costly process. It took several days to download, parse and extract the metrics for about ninety versions of each software system. Two more lightweight approaches, which work well in most of the cases, are based on previous defects (**BUGFIXES**) and source code metrics extracted from a single version (**CK+OO**). However, approaches based on bug or multiple versions data have limited usability, as the history of the system is needed, which might be inaccessible or, for newly developed systems, not even existent. This problem does not hold for the source code metrics **CK+OO**, as only the last version of the system is necessary to extract them.

Using the source code metrics, **CK+OO** *to predict bugs has several advantages: They are lightweight to compute, have good explanative and predictive power and do not require historical information.*

What are the best source code metrics? The **CK** and **OO** metrics fare comparably in predictive power (with the exception of Mylyn), whereas the **OO** metrics have the edge in explanative power. However, the combination of the two metric sets **CK+OO** is a considerable improvement over them separated, as the performance is more homogeneous across all case studies. In comparison, using lines of code (**LOC**) only, even if it is simple, yields a poor predictor, as its behavior is unstable among systems.

Using the **CK** *and the* **OO** *metric sets together is preferable to using them in isolation, as the performances are more stable across case studies.*

Is there an approach based on a single metric with good and stable performances? We have just seen that **LOC** is a predictor of variable accuracy. All approaches based on a single metric, *i.e.,***NR**, **BUGFIXES**, **NFIXONLY** and **HCM** (and variants) have the same issues: The results are not stable for all the case studies. However, among them **BUGFIXES** is the best one.

Bug prediction approaches based on a single metric are not stable over the case studies.

What is the best weighting for past metrics? In multi-version approaches and entropy of changes, weighting has an impact on explanative and predictive power. Our results show that the best weighting is linear, as models with linear decay have better predictive power and better or comparable explanative power than models with exponential or logarithmic decay (for entropy of changes, churn and entropy of source code metrics).

The best weighting for past metrics is the linear one.

	Adjusted R^2 - Explanative power						Spearman correlation - Predictive power					
Predictor	Eclipse	Mylyn	Equinox	PDE	Lucene	Score	Eclipse	Mylyn	Equinox	PDE	Lucene	Score
Change metrics (Section IV-A)												
MOSER	0.454	**0.206**	0.596	0.517	**0.57**	9	0.323	**0.284**	**0.534**	0.165	0.238	6
NFIX-ONLY	0.143	0.043	0.421	0.138	0.398	-3	0.288	0.148	0.429	0.113	0.284	-1
NR	0.38	0.128	0.52	0.365	0.487	2	0.364	0.099	**0.548**	0.245	0.296	5
NFIX+NR	0.383	0.129	0.521	0.365	0.459	2	0.381	0.091	**0.567**	0.255	0.277	4
Previous defects (Section IV-B)												
BF (short for BUGFIXES)	0.487	0.161	0.503	0.539	**0.559**	5	**0.41**	0.159	0.492	**0.279**	**0.377**	10
BUG-CAT	0.455	0.131	0.469	0.539	**0.559**	5	**0.434**	0.131	0.513	**0.284**	**0.353**	9
Source code metrics (Section IV-C)												
CK+OO	0.419	0.195	**0.673**	**0.634**	0.379	8	0.39	**0.299**	0.453	**0.284**	0.214	8
CK	0.382	0.115	0.557	0.058	0.368	0	0.377	0.226	0.484	0.256	0.216	4
OO	0.406	0.17	**0.619**	**0.618**	0.209	6	0.395	**0.297**	0.49	0.263	0.214	6
LOC	0.348	0.039	0.408	0.04	0.077	-3	0.38	0.222	0.475	0.25	0.172	2
Entropy of changes (Section IV-D)												
HCM	0.366	0.024	0.495	0.13	0.308	-2	**0.416**	-0.001	0.526	0.244	0.308	5
WHCM	0.373	0.038	0.34	0.165	0.49	-1	**0.401**	0.076	**0.533**	0.273	0.288	7
EDHCM	0.209	0.026	0.345	0.253	0.22	-4	0.371	0.07	0.495	0.258	0.306	3
LDHCM	0.161	0.011	0.463	0.267	0.216	-4	0.377	0.064	**0.581**	**0.28**	0.275	6
LGDHCM	0.054	0	0.508	0.209	0.141	-3	0.364	0.03	**0.562**	0.263	0.33	5
Churn of source code metrics (Section IV-E)												
CHU	0.445	0.169	**0.645**	**0.628**	0.456	8	0.371	0.226	0.51	0.251	0.292	5
WCHU	**0.512**	0.191	**0.645**	**0.608**	0.478	11	**0.419**	**0.279**	**0.56**	**0.278**	0.285	13
LDCHU	**0.557**	**0.214**	0.581	**0.616**	0.458	11	0.395	**0.275**	**0.563**	**0.307**	0.293	11
EDCHU	**0.509**	**0.227**	0.525	**0.598**	0.467	11	0.362	0.259	0.464	**0.294**	0.28	6
LGDCHU	0.473	0.095	**0.642**	0.486	0.493	5	**0.442**	0.188	**0.566**	0.189	0.29	7
Entropy of source code metrics (Section IV-F)												
HH	0.484	0.199	**0.667**	0.514	0.433	7	**0.405**	**0.277**	0.484	0.266	0.318	9
HWH	0.473	0.146	**0.621**	**0.641**	0.484	8	**0.425**	0.212	0.48	0.266	0.263	5
LDHH	**0.531**	**0.209**	0.596	0.522	0.343	8	**0.408**	**0.272**	**0.53**	**0.296**	0.333	13
EDHH	0.485	**0.226**	0.469	0.515	0.359	5	0.366	**0.273**	**0.586**	**0.304**	0.337	11
LGDHH	0.479	0.13	**0.66**	0.447	0.419	4	**0.421**	0.185	0.492	0.236	**0.347**	8
Combined approaches												
BF+CK+OO	0.492	**0.213**	**0.707**	**0.649**	**0.586**	13	**0.439**	**0.277**	**0.547**	**0.282**	**0.362**	15
BF+WCHU	**0.536**	0.193	**0.645**	**0.627**	**0.594**	13	**0.448**	0.265	**0.533**	**0.282**	0.31	11
BF+LDHH	**0.561**	**0.217**	**0.615**	**0.601**	**0.592**	15	**0.422**	0.221	**0.533**	**0.305**	**0.352**	12
BF+CK+OO+WCHU	**0.559**	**0.25**	**0.734**	**0.661**	**0.61**	15	**0.425**	**0.306**	0.524	**0.31**	0.298	11
BF+CK+OO+LDHH	**0.587**	**0.262**	**0.73**	**0.68**	**0.618**	15	**0.44**	**0.291**	**0.571**	**0.312**	**0.377**	15
BF+CK+OO+WCHU+LDHH	**0.62**	**0.277**	**0.754**	**0.691**	**0.65**	15	**0.408**	**0.326**	**0.592**	**0.289**	**0.341**	15

Table IV
EXPLANATIVE AND PREDICTIVE POWER FOR ALL THE APPROACHES.

Are bug fixes extracted from the versioning system a good approximation of actual bugs? If we compare the performance of **NFIX-ONLY** with respect to **BUGFIXES** and **BUG-CAT**, we see that the heuristic searching bugs from commit comments is a poor approximation of actual past defects. On the other hand, there is no improvement in categorizing bugs.

Using string matching on versioning system comments, without validating it on the bug database, decreases the accuracy of bug prediction.

Can we go further? One can argue that bug information is anyways needed to train the model. We investigated whether adding this metric to our best performing approaches would yield improvements at a moderate cost. We tried various combinations of **BUGFIXES**, **CK+OO**, **WCHU** and **LDHH**. We display the results in the lower part of Table IV, and see that this yields an improvement, as the **BUGFIXES+CK+OO** approach scores a 15 (instead of a 10 or an 8), despite being lightweight. The combinations involving **WCHU**, exhibit a gain in explanative but not in predictive power: The Spearman correlation score is worse for the combinations (11) than for **WCHU** alone (13). One combination involving **LDHH**, **BF+CK+OO+LDHH**, yields a gain both in explanative and predictive power (15 for both). The same holds for the combination of all the approaches (**BF+CK+OO+WCHU+LDHH**).

Combining bugs and OO metrics improves predictive power. Adding this data to **WCHU** *improves explanation, but degrades prediction, while adding it to* **LDHH** *improves both explanation and prediction.*

VI. THREATS TO VALIDITY

Threats to Construct Validity regard the relationship between theory and observation, *i.e.,* the measured variables may not actually measure the conceptual variable. A first threat concerns the way we link bugs with versioning system files and subsequently with classes. In fact, all the links that do not have a bug reference in a commit comment cannot be found with our approach. Bird *et al.* studied this problem in bug databases [27]: They observed that the set of bugs which are linked to commit comments is not a fair representation of the full population of bugs. Their analysis of several software projects showed that there is a systematic bias which threatens the effectiveness of bug prediction models. However, this technique represents the state of the art in linking bugs to versioning system files [18], [24].

Another threat is the noise affecting Bugzilla repositories. In [28] Antoniol *et al.* showed that a considerable fraction of problem reports marked as bugs in Bugzilla (according to their severity) are indeed "non bugs", *i.e.,* problems not related to corrective maintenance. We manually inspected a statistically significant sample (107) of the Eclipse JDT Core bugs we linked to CVS files, and found that more than 97% of them were real bugs[1]. Therefore, the impact of this threat on our experiments is limited.

Threats to Statistical Conclusion Validity concern the relationship between the treatment and the outcome. In our experiments we used the Spearman correlation coefficient to evaluate the performances of the predictors. All the correlations are significant at the 0.01 level.

Threats to External Validity concern the generalization of the findings. We have applied the prediction techniques to open-source software systems only. There are certainly differences between open-source and industrial development, and in particular because some industrial settings enforce standards of code quality. We minimized this threat by using parts of Eclipse in our benchmark, a system that while being open-source has a strong industrial background. A second threat concerns the language: All considered software systems are written in Java. Adding non-Java systems to the benchmark would increase its value, but would introduce problems since the systems would need to be processed by different parsers, producing variable results.

The bias between the set of bugs linked to commit comments and the entire population of bugs, that we discussed above, threatens also the external validity of our approach, as results obtained on a biased dataset are less generalizable.

To decrease the impact of a specific technology/tool, in our dataset we included systems developed using different versioning systems (CVS and SVN) and different bug tracking systems (Bugzilla and Jira). Moreover, the software systems in our benchmark are developed by independent development teams and emerged from the context of two unrelated communities (Eclipse and Apache).

VII. Conclusion

Bug prediction concerns the resource allocation problem: Having an accurate estimate of the distribution of bugs across components helps project managers to optimize the available resources by focusing on the problematic system parts. Different approaches have been proposed to predict future defects in software systems, which vary in the data sources they use and in the systems they were validated on, *i.e.,* no baseline to compare such approaches exists.

We have introduced a benchmark to allow for common comparison, which provides all the data needed to apply several prediction techniques proposed in the literature. Our dataset, publicly available at http://bug.inf.usi.ch, allows the reproduction of the experiments reported in this paper and their comparison with novel defect prediction approaches.

We evaluated a selection of representative approaches from the literature, some novel approaches we introduced, and a number of variants. Our results showed that the best performing techniques are **WCHU** (Weighted Churn of source code metrics) and **LDHH** (Linearly Decayed Entropy of source code metrics), two novel approaches that we proposed. They gave consistently good results –often in the top 90% of the approaches– across all five systems. As WCHU and LDHH require a large amount of data and computation, past defects and source code metrics are lightweight alternatives with overall good performance. Our results provide evidence that prediction techniques based on a single metric do not work consistently well across all systems.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063) and the European Smalltalk User Group (http://www.esug.org).

References

[1] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751–761, 1996.

[2] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Trans. Software Eng.*, vol. 22, no. 12, pp. 886–894, 1996.

[3] L. C. Briand, J. W. Daly, and J. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Trans. Software Eng.*, vol. 25, no. 1, pp. 91–121, 1999.

[4] K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[5] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Software Eng.*, vol. 29, no. 4, pp. 297–310, 2003.

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

[7] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *Proceedings of ICSE 2005*. ACM, 2005, pp. 580–586.

[8] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of ICSE 2006*. ACM, 2006, pp. 452–461.

[1] This is not in contradiction with [28]: Bugs mentioned as fixes in CVS comments are intuitively more likely to be real bugs, as they got fixed.

[9] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of ICSE 2005*. ACM, 2005, pp. 284–292.

[10] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of ICSE 2009*, 2009, pp. 78–88.

[11] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of ICSE 2008*, 2008, pp. 181–190.

[12] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Proceedings of IWPSE 2007*, 2007, pp. 11–18.

[13] S. Kim, T. Zimmermann, J. Whitehead, and A. Zeller, "Predicting faults from cached history," in *Proceedings of ICSE 2007*. IEEE CS, 2007, pp. 489–498.

[14] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Software Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[15] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of ICSM 2005*, 2005, pp. 263–272.

[16] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797–814, 2000.

[17] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[18] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of PROMISE 2007*. IEEE CS, 2007, p. 76.

[19] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of ICSE 2008*, 2008.

[20] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 287–300, 2008.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of CCS 2007*. ACM, 2007, pp. 529–540.

[22] R. Kollmann, P. Selonen, and E. Stroulia, "A study on the current state of the art in toolsupported UML-based static reverse engineering," in *Proceedings of WCRE 2002*, 2002, pp. 22–32.

[23] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1 — The FAMOOS Information Exchange Model," University of Bern, Tech. Rep., 2001.

[24] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of ICSM 2003*. IEEE CS, 2003, pp. 23–32.

[25] E. J. Jackson, *A Users Guide to Principal Components*. John Wiley & Sons Inc., 2003.

[26] T. L.Graves, A. F.Karr, J.S.Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Trans. Software Eng.*, vol. 26, no. 07, pp. 653–661, 2000.

[27] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of ESEC/FSE 2009*. New York, NY, USA: ACM, 2009, pp. 121–130.

[28] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of CASCON 2008*. ACM, 2008, pp. 304–318.

APPENDIX
BUG PREDICTION DATASET

To make the experiments presented in this paper reproducible, we created a website[2] where we share our bug prediction dataset. The dataset is a collection of models and metrics of five software systems and their histories. The goal of such a dataset is to allow researchers to compare different defect prediction approaches and to evaluate whether a new technique is an improvement over existing ones.

In particular, the dataset contains the data needed to run a defect prediction technique, and compute its performance by comparing the prediction with an oracle set, *i.e.,* the number of post release defects as reported in the bug tracking system.

We designed the dataset to perform defect prediction at the class level. However, package or subsystem information can be derived by aggregating class data, since per each class the dataset specifies the package that contains it.

Table V summarizes the contents of our dataset.

System models as FAMIX MSE files
91 versions of Eclipse JDT Core
97 versions of Eclipse PDE UI
91 versions of Equinox Framework
98 versions of Mylyn
99 versions of Lucene

For each class in each system version:
6 CK metrics
11 object oriented metrics
15 change metrics
Categorized (with severity and priority) past defect counts
Categorized (with severity and priority) post-release defect counts

For each class history (over all the versions):
Churn measures for all CK and object oriented metrics
Entropy measures for all CK and object oriented metrics
Complexity of code change measures
Weighted, linear, exponential and logarithmic variants of churn, entropy and complexity of code change

Table V
CONTENTS OF THE BUG PREDICTION DATASET

On the website the data is available as either CSV files or MSE[3] files (for FAMIX models).

Our bug prediction dataset is not the only one publicly available. Other datasets exist (for example http://promisedata.org), but none of them provides all the pieces of information that ours includes: Process measures extracted from versioning system logs, defect information and source code metrics for hundreds of system versions. The extensive set of metrics we provide makes it possible to compute the churn and entropy of source code metrics, and to compare a wider set of defect prediction techniques.

[2] Available at http://bug.inf.usi.ch

[3] Specs available at http://www.moosetechnology.org/docs/mse