Predicting Risk of Pre-Release Code Changes with CheckinMentor

Alexander Tarvo*
Brown University
Providence, RI
alexta@cs.brown.edu

Nachiappan Nagappan, Thomas Zimmermann,
Thirumalesh Bhat, Jacek Czerwonka
Microsoft Corporation
Redmond, WA
{nachin, tzimmer, thirub, jacekcz }@microsoft.com

Abstract—**Code defects introduced during the development of the software system can result in failures after its release. Such post-release failures are costly to fix and have negative impact on the reputation of the released software. In this paper we propose a methodology for early detection of faulty code changes. We describe code changes with metrics and then use a statistical model that discriminates between faulty and non-faulty changes. The predictions are done not at a file or binary level but at the change level thereby assessing the impact of each change. We also study the impact of code branches on collecting code metrics and on the accuracy of the model.**

The model has shown high accuracy and was developed into a tool called CheckinMentor. CheckinMentor was deployed to predict risk for the Windows Phone software. However, our methodology is versatile and can be used to predict risk in a variety of large complex software systems.

Keywords — code change, risk, software metrics, code branch

I. Introduction

Faulty changes, made during the development of the software system, must be detected and fixed before the system is released. Otherwise they can cause post-release failures.

Fixing a post-release failure is costly because it requires not just implementing a fix, but also to test the change and to issue an update for the software system. As a result, post-release failures remain a severe problem in all kinds of software systems. But one area which suffers the most is, probably, the mobile software. The size and complexity of mobile software approaches those of traditional desktop software. At the same time, high competitiveness of mobile market forces developers to reduce time between releases of mobile platforms, which leaves less time for extensive testing and quality control. Most importantly, post-release failures result in user dissatisfaction and can reduce sales of an otherwise successful product.

The key method to either avoid or at least decrease the number of post-release failures is thorough testing of all the changes made before the system's release (*pre-release changes*). However, extensive testing is an extremely time-consuming and costly enterprise by itself. Thus some method of test prioritization is necessary. The most widely used approach is to assess the risk of code changes. The change is *risky* if it has a high probability of introducing a failure. Such changes could be subject to additional testing.

Often, the risk of a change is estimated by an expert or a group of experts (e.g. code review). However, the manual estimation of the change risk is subjective as it relies on skills and experience of experts. Furthermore, it is hardly applicable to large systems, where hundreds and even thousands of code changes can be made every day. This necessitates development of an objective and automated solution for risk prediction.

Existing approaches [3][9][13] predict risk of code changes made after the system's release. But usually these post-release changes are smaller and much less frequent. As a result, existing methods for risk prediction do not scale well when applied to larger and more numerous pre-release code changes.

We present a general-purpose approach to predict risk of pre-release code changes in a large modern software system. To the best of our knowledge this is the first paper dedicated to predicting risk of pre-release code changes. We do not predict risk of individual functions, files or binaries. We rather predict risk of individual changes (a single change can affect multiple source files, binaries, or functions) to make it more actionable for the user to identify the impact of these changes.

We develop and validate our methodology using the data on a Windows Phone 7 mobile OS, but it can be easily applied to any large software system. We describe each code change with metrics and use a statistical model to predict its risk. We evaluate various metrics, such as the size of a change, properties of the changed components, and historic code churn. To ensure that our approach is applicable to a wide variety of software systems, we use only data sources that are common for virtually all the modern software projects.

Training a statistical model requires knowledge of which code changes are risky and which are not. This information is normally not available for the pre-release changes, and to quantify risk of existing changes we use our modification of the SZZ algorithm. Another novel aspect of our work is taking into account branching information in the software system (here the term "branch" denotes a separate copy of the source code in the version control system). In particular, we show how multiple code branches can affect collection of change metrics and, ultimately, the accuracy of the prediction model. Our model and the corresponding data collection process were

* Alexander Tarvo was a research intern in the Microsoft Research when this work was carried out.

implemented as a web application named CheckinMentor. CheckinMentor has been used on the daily basis by the Windows Phone team.

The rest of the paper is organized as following. In the Section 2 we provide a survey of related work. In the Section 3 we discuss data collection. In the Section 4 we explain our definition of the change risk. In the Section 5 we describe the set of metrics we use to describe changes. In the Section 6 we describe our experiments. The Section 7 presents threats to validity of our work and the Section 8 concludes the paper.

II. Related work

Predicting risk in software systems is a well-studied area. Most of the publications are dedicated to predicting fault proneness of individual components of the software system [2][10][11][12][14][15][16][20]. In a fault proneness model the system is divided into a number of components, and each component is described by metrics (numeric properties). A statistical model is normally used to discriminate between faulty and non-faulty components (Kim et al [21] use a cache-based algorithm instead). A notable extension of fault proneness models are the effort-based models [23] that also predict the amount of effort required to test the component.

Various metrics have been used to predict fault proneness: Menzies et al [17] used code metrics such as component size and complexity; Basili et al [18] relied on object-oriented metrics; Zimmermann and Nagappan [11] used dependencies between the components. Nagappan and Ball used code churn to predict defect density in components [10]. Hassan [19] used entropy-based change complexity measures to predict the number of faults. Metrics mined from the e-mail archives [22], organization structure [2], and socio-technical networks [12] proved to be good predictors of faulty components.

Fault proneness models point to components that are likely to fail, but they cannot predict risk of failure for individual code changes. Nevertheless, a similar approach can be used to predict risk of code changes. Code changes are described with metrics and then a statistical method is used to discriminate between "risky" and "non-risky" changes. However, predicting risk for the code changes appears to be a less explored area. Authors of this paper are aware of a few publications on the subject, all dealing with post-release changes only.

Mockus and Weiss [9] predicted risk of code changes in the telecommunication system using logistic regression. The metrics include: the size of the change, the number of changed components, and the experience of the developer. Authors report the percentage of Type I and Type II errors in (20%; 40%). Kim et al [13] collected information on about 12,000 post-release changes in 12 open-source programs. They used a support vector machine to discriminate between risky and non-risky changes. The precision of the model ranges in (0.44; 0.85) and recall is in (0.43; 0.86). Shihab et al predicted subjective risk for changes, as it is perceived by developers, in the mobile-phone software [24]. Tarvo [3] predicted risk for post-release changes in the Windows OS. Metrics used by the author include change size, properties of changed components and dependencies between those, and developer experience. The resulting classifier had the area under ROC curve of 0.77.

One reason for the lack of research on the topic is the difficulty of collecting a dataset for training a statistical model, which involves measuring actual risk of changes and labeling them as "risky" or "not risky". Some authors rely on subjective estimation of risk by programmers, which may not reflect the actual probability of introducing a failure [24]. In other cases [3] such labeling can be mined from software repositories. But usually it must be inferred. Unfortunately, existing algorithms for automated labeling of code changes such as SZZ [1] are computationally intensive and cannot be readily applied to infer risk of pre-release changes in a large software system.

Our work makes a number of important contributions to the state of the art:

- to the best of our knowledge, this is the first paper on predicting risk for pre-release code changes;
- we propose a novel algorithm to label risky code changes in the training set;
- we study the effect of code branches on data collection techniques and on the accuracy of the resulting model.

III. Data collection

Before predicting risk for new code changes the statistical model must be trained using data on existing changes. The training dataset must include two aspects of information on changes: change metrics and change risk. *Change metrics* are the numeric characteristics of the change that form predictor (independent) variables in the model. The *risk of the change* defines a target (dependent) variable.

Training dataset can be collected using data on the previous release of the system, or from a similar system. Change metrics are collected using the data on pre-release code changes in that system. Change risk is determined using data on bug fixes that occurred after its release. Thus collecting metrics and determining risk of changes become the most important aspects of our data collection stage.

A. Source of information on the software system

Our goal is developing a general-purpose approach that can predict risk for **various large** software systems. This imposes specific requirements for change metrics, and, correspondingly, to sources of information about changes:

1. *Expressiveness.* Metrics should capture important characteristics of the code changes and be effective predictors of their risk;
2. *Portability.* Metrics must be easily computed for different products and development teams regardless of the specifics of engineering process and tools they use;
3. *Performance.* Predicting risk for code changes must be done on a regular basis. As a consequence, computation of change metrics should not take a long time;

Because of these requirements change metrics may differ from metrics used in fault proneness models. In particular, highly expressive metrics can be non-portable and have low performance. One example of expressive but non-portable metrics is organizational metrics. These metrics require knowledge of organizational hierarchy at the specific date when the change was made, which might not be available for

`

certain teams. Another example of expressive but low-performance metrics is dependency metrics. The structure of the system can change frequently during the pre-release timeframe. As a result, dependency metrics must be recomputed for each pre-release code change, which usually takes a long time.

Despite such strict requirements, we have developed a set of metrics which allow for high prediction accuracy, but still can be quickly computed using only sources of information used by the vast majority of development teams – a version control system and a bug-tracking database.

The *version control system (VCS)* stores a complete history of changes in the source code of the program. The representation of the history in the VCS is two-fold. On the one hand, each change is represented as a *checkin* – a record that contains detailed information on a change, including the date and time of the change, its textual description, the list of the changed files, and the name of the developer who made the change. From the point of risk prediction, a **checkin is the one single entity that most naturally corresponds to the notion of the "code change".** On the other hand, the VCS also tracks the history of changes in the source code of the system. For each source file f the VCS stores a set {f[1],...,f[m]} of its previous *revisions*. For each revision f[i] we can retrieve the corresponding checkin and the textual diff between f[i] and the previous revision f[i-1].

The VCS provides comprehensive information about code changes themselves, but it does not provide rationale why those changes were made. This data is stored in the *Bug Tracking Database (BTD)* in form of bug records. Each bug record is identified by a unique identifier and contains information such as the type of the work item (is it a bug, a new feature, or another type of the change), a list of people who worked on the issue, and other useful data.

The checkin and the bug record can be related to each other. For example, a bug fix in the system should be reflected both as the change in the source code (and, correspondingly, as a checkin in the VCS) and as the bug record in the BTD. Unfortunately, discovering such relations is not easy. Although the ID of the corresponding bug can be entered as a part of the checkin description field, there is no single format respected by the engineering teams. Similarly, the checkin ID can be entered into the bug record, but it is not required. Furthermore, the bug-to-checkin relation is not defined for all the changes: for example, not all the changes made during the development of the system will be reflected in the BTD. Nevertheless, a relation between checkins and bugs is important; we use it to build a training set for the risk prediction model.

B. Branches and their impact on data collection

A *code branch* is a separate copy of the system's source code with its own history of changes. Branches are used to distribute and isolate development of a program across multiple teams of engineers. Nowadays practically all large software systems are developed using multiple code branches.

Normally, a team maintains their own private copy of the program's code in a separate *feature branch*. When a developer changes system's code, she commits the change (we call such changes *"plain" checkins* or *edits*) into the corresponding feature branch.

Once a certain milestone is reached, development teams submit their changes into the *main branch* (also called *trunk*) during the process called *reverse integration*. To improve the stability of the system, integrations can traverse one or more *interim branches* (see Figure 1) before reaching the trunk. We call resulting changes in the trunk and in interim branches *integration checkins*.

After all the teams copy their changes into the trunk, the system is being tested. As the system reaches an acceptable level of quality, all the code from the trunk is copied back into the feature branches – this process is called *forward integration*. Forward integration ensures that developers will use the latest version of the system's code while working on the next milestone.

Branches facilitate engineering process because changes made by each team on the ongoing basis will not affect other teams working on different parts of the system. But as a result of branching various types of code changes (integrations and plain checkins) are scattered across multiple code branches. This implication is not that important for fault proneness models, which concentrate on overall changes in the components of the system. But it is crucial for our model that concentrates on individual checkins. Here branching poses an important question: for what kind of checkins we should predict risk: plain checkins, integration checkins, or both?

In the context of our work, **we predict risk for plain checkins only**. There are important reasons for this.

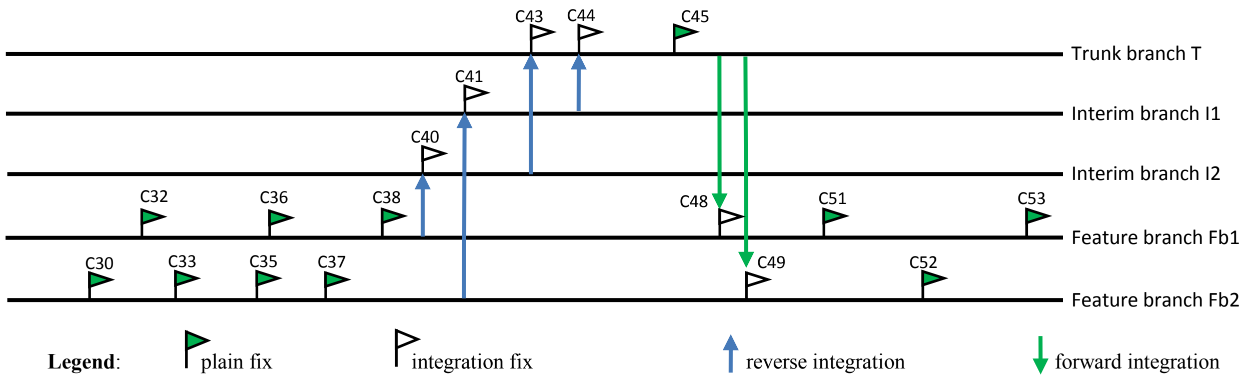


Figure 1: Code branches in a software system

`

First, plain checkins are the actual changes made by the developers, and these are the checkins for which management wants to know their risk.

Second, a single integration checkin may contain changes from multiple plain checkins. It would be impossible to determine how individual plain checkins contribute to the risk of the integration checkin.

Third, integration checkins contain the same *overall* set of code changes as plain checkins. They do not add any new information about code changes, but rather cause loss of important pieces of information about plain checkins:

- **code churn**. Some files can be changed multiple times before being integrated into the trunk. However, in the trunk only a single change will be recorded;
- **developer who made the change**. Edits and integrations are usually performed by different people. Analysis of integration checkins will not yield information about the developer who made the change;
- **bugs**. Integration checkins usually do not contain information about associated bugs. If present, this data is available only in individual edits.

As a result, mining multiple branches is necessary to collect history of changes in source files. Complications associated with mining historic code churn from multiple branches are discussed in the Section V.

IV. DEFINING RISK OF THE CHECKIN

To train our risk model we must mark which pre-release checkins in the training set are risky. We state that **a pre-release checkin is risky if it causes a bug fix after the system's release**. Such checkins are also called as "bug-introducing" checkins.

Different methods for identifying bug-introducing checkins have been proposed. In [3] the author relies on the special field in the bug record, which points to the bug-introducing checkin. In [24] authors used developers' subjective estimation of the checkin risk. These approaches proved to be successful for building risk models for post-release checkins, where bug-introducing checkins can be identified manually. However, this information is generally not available for the pre-release checkins because of the large number of such checkins.

To address this problem the SZZ algorithm [1] can be used. SZZ relies on a combination of annotation graphs and heuristics to identify bug-introducing checkins. The key assumptions of the SZZ are 1) if the bug is found it will be fixed and 2) the fix for the bug affects the same area of the code (function, line) of the system. The SZZ is considered to be accurate; however, it requires computing annotation graphs for each source line affected by the change. This is acceptable for relative small projects the SZZ was tested upon (~100 KLOC, 500-1000 checkins), but can be prohibitively expensive for large systems with tens of millions lines of code and hundreds of thousands of checkins. In order to make the SZZ applicable for large software systems we modified it in the following way:

`

Decreasing the granularity. We identify bug-introducing checkins at the level of a source file. The change to the source file is considered to be *bug-introducing* if the file was changed after the release and if that post-release change was a bug fix.

Introducing the Change Risk Index. We define the *change risk index* of the checkin c denoted as CRI(c) as the number of bug-introducing file changes in c.

Figure 2 shows an example of risk calculation for 4 pre-release checkins that affect 5 source files. Checkin C30 affects files f4 and f5; CRI(C30)=1 because the file f4 has a post-release bug fix. Checkin C33 affects only the file f5; CRI(C33)=0 because there are no post-release bug fixes in f5. CRI(C36)=2 since 2 out of 4 files affected by C36 have post-release bug fixes; and CRI(C38)=0 because the only affected file f1 has no post-release changes at all.

The purpose of the CRI is two-fold. First, CRI serves as a proxy metric to the amount of work required to fix bugs caused by c. The high values of CRI for the checkin mean that there are either multiple small bugs affecting its files or there are few serious issues which require large fixes.

Second, the CRI is called to improve the accuracy of our algorithm. Decreasing the granularity makes our algorithm faster than the original SZZ, but allows for a larger number of false positives. If the same file was modified twice in the pre-release timeframe, it is not possible to precisely pinpoint the actual bug-introducing checkin without thorough analysis of its code. However, if the bug-introducing checkin changes multiple source files, the bug fix for it normally affects some percentage of these files too. In this case the CRI for the bug-introducing checkin will be higher than the CRI of a correct check-in that coincidentally involves one of the affected files. For example the bug-introducing checkin C36 has CRI(C36)= 2, while for the correct checkin C30 CRI(C30)=1. To reduce the amount of false positives we introduce a lower threshold on the CRI values for checkins.

Similar to SZZ our algorithm requires discriminating between types of the checkins (bug fixes vs. new features). We use the BTD to obtain this information. We retrieve all the post-release checkins and try matching them with bug records. The checkin is considered to be a bug fix if the corresponding bug record exists and is marked as a "bug fix".

We consider the bug to be related to the checkin if the bug

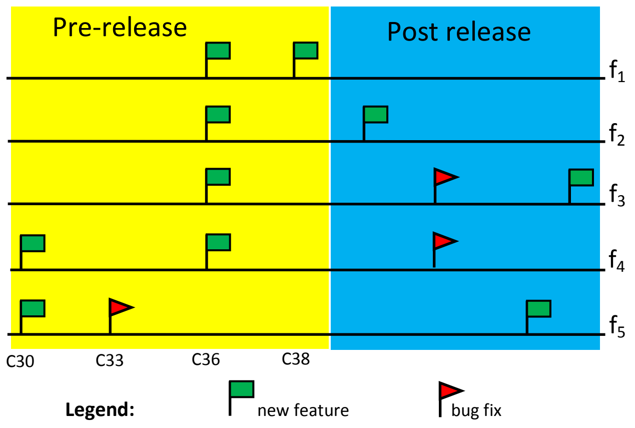


Figure 2: Risk calculation example

record has a reference to the checkin or if the checkin description contains an ID of the corresponding bug. Since the checkin description can have numbers other than the bug ID, we perform additional data cleanup by using heuristics. The relation is considered to be valid if:

- the bug with such bug ID is present in the BTD and
- the bug was resolved as "Fixed" and
- the checkin was made after the bug record was opened, but before it was resolved.

V. CHECKIN METRICS

Normally, version control systems collect change data for the individual source files in the software system. But in our work we have to define metrics for the individual checkins, which usually affect multiple files. This necessitates for some way to aggregate file-level metrics into a checkin level. Formally, if the checkin c affects n source files $f_1,\ldots,f_n$ then the metric M for the c will be an aggregation of values of the metric M for each individual source files:

$$\mathrm{M}(c) = aggr(\mathrm{M}(\mathrm{f}_i)) \qquad (1)$$

In our study we used min(), max(), and avg() aggregation functions.

The checkin metrics we collect can be divided into following groups: code metrics, change size, historical code churn, and organization metrics.

A. Code metrics

Code metrics characterize properties of the components of the software system that are modified by the checkin. The underlying intuition is that changes in large and complex components are more risky. We collect following code metrics that represent the size of the changed components:

- LOC(c): the total number of code lines in all the files changed by the checkin c;
- CLASS(c): the total number of classes in all the files changed by the checkin c;
- FUNC(c): the total number of functions in all the files changed by the checkin c;

There are other code metrics such as cyclomatic complexity or dependency metrics. However, those metrics are hard to compute using only VCS data and are usually highly correlated with size metrics, thus we exclude them from our study.

B. Change size

The amount of changes introduced by a checkin is a strong predictor of risk for post-release checkins [3][24]. We apply this technique to predict risk of pre-release checkins as well.

We denote the amount of changes introduced by the checkin c (its change size) as Δ(c). Depending on the granularity of changed items (source line, code chunk, source file) and the type of the changed (is the item added, changed, or deleted) we have defined total 9 different change metrics. Our change metrics are compact, expressive, and can be easily computed solely by using data from the VCS.

Δ_LOC_chg(c), Δ_LOC_add(c), Δ_LOC_del(c) metrics denote the number of lines of code changed, added, or deleted by the checkin c correspondingly. Δ_chunk_chg(c), Δ_chunk_add(c), Δ_chunk_del(c) refer to the number of changed, added, or deleted code chunks (here *code chunk* is a fragment of the program's code consisting of a few adjacent lines). Δ_file_chg(c), Δ_file_add(c), Δ_file_del(c) change metrics correspond to the number of source files added, changed or deleted by the checkin. Δ_file_int(c) is the number of files in c changed due to code integrations.

C. Historic code churn

It is known [10] that components with large amount of pre-release changes are prone to post-release failures. We rely on this observation to define historic code churn metrics.

Conceptually, *historic code churn* is a summary of change size metrics collected across previous revisions of source files affected by the checkin. To compute historic code churn for the checkin we employ the concept of a "sliding window". Let us consider the checkin c that was made on the date t and affects files $f_1,\ldots,f_n$. We want to compute the historic code churn for c that has occurred within past w days (a sliding window of a size w), which we denote as $\Delta^w(c)$. We do it in two stages.

First, we compute the historic code churn $\Delta^w(\mathrm{f_i})$ for each file $f_i \in \{f_1,\ldots,f_n\}$ affected by c. We build a set $F=\{f_i[k],\ldots,f_i[l]\}$ that contains revisions k,…,l of the file f_i that happened within the time interval [t-w;t). For each revision $\mathrm{f_i}[j] \in F$ we compute the code churn $\Delta(\mathrm{f_i}[j])$ between the revisions $\mathrm{f_i}[j]$ and $\mathrm{f}_i[j-1]$. Then we summarize the amount of changes occurred in those revisions:

$$\Delta^w(\mathrm{f}_i) = \sum_{j=k}^{l} \Delta(\mathrm{f}_i[j]) \qquad (2)$$

Second, we compute historic churn for the checkin c by summing historic churn metrics for all the affected files $f_1,\ldots,f_n$ according to (1):

$$\Delta^w(c) = \sum_{i=1}^{n} \Delta^w(\mathrm{f}_i) \qquad (3)$$

Similarly to the change size, we collect historic churn at the different levels of granularity (source line, chunk, source file) and for the different change types (added, changed, and deleted). We also collect the metric Δ^w_chekins(c), which denotes the number of checkins that affected files $f_1,\ldots,f_n$. Here we count only unique checkins; namely, if the same checkin affected two different files f_i and f_j, we count this as a single change towards computing Δ^w_checkins(c).

D. Organization metrics

Organization metrics proved to be good predictors of the fault proneness [2]. They represent information about the developer who made a checkin. However, computing most of these metrics requires knowledge of the organization hierarchy, which cannot be directly obtained from the VCS. Thus we define only a limited set of organizational metrics:

- EXP(c): the experience of the developer d who made the checkin c. Calculated as the total number of checkins the d made within the past w days;

`

- NDEV(c): the total number of developers who changed files $f_1,\dots,f_n$ within previous w days. Similarly, we count only unique developers. If some developer changed two or more files $f_1,\dots,f_n$ we count him as a single developer towards NDEV(c).

E. Collecting checkin metrics in multi-branch environments

Calculating historic churn and organization metrics for the checkin c requires collecting a history of previous changes for each file f_i changed by c. So far for the sake of simplicity we assumed there is only one copy of each source file that contains a complete history of all its changes. However, if the program was developed using multiple code branches, the previous changes to the source file can be scattered across some of these branches.

We investigated three basic methodologies for collecting the history of code changes in multi-branch environments, evaluated their predictive power and implementation complexity. These methodologies differ in the way how we build the set F of previous revisions of the source file f_i: collecting F from all the branches in the system, collecting F only from the trunk branch, or collecting F only from the branch where the checkin c was made.

1. Collecting historic information from all the branches.

The most comprehensive set of changes is a set **$F= F_{plain}^{all}$** of plain checkins that were made across all the branches within the given time window. Below we describe the approach we use to build the set F_{plain}^{all}.

Suppose we have a $f^{fbY}[p]$ – the p-th revision of the file f created on the date d in the branch fbY (the superscript denotes affiliation of the file to the branch). To obtain a set F_{plain}^{all} of previous revisions that occurred across all the branches in the timeframe [t-w; t) we define two sets F_{plain}^{trunk} and F_{plain}^{fb}:

F_{plain}^{trunk} ={f^T[i],...,f^T[j]} contains revisions i,...,j of the file f in the trunk such that:

- {f^T[i],...,f^T[j]} are plain checkins;
- {f^T[i],...,f^T[j]} occurred within the time window [t-w;t).

F_{plain}^{fb} = {f^{fb1}[k],...,f^{fbX}[l]} contains all revisions of the f in all the feature branches fb1,...,fbX such that:

- {f^{fb1}[k],...,f^{fbX}[l]} were integrated into the trunk;
- {f^{fb1}[k],...,f^{fbX}[l]} are plain checkins;
- {f^{fb1}[k],...,f^{fbX}[l]} occurred within the time window [t-w;t).

Finally, we define the set $F_{plain}^{all} = F_{plain}^{trunk} \cup F_{plain}^{fb}$.

For an example, consider a situation depicted at the Figure 1. Suppose we would like to build a set F_{plain}^{all} for the change associated with the checkin C51 (in this example we specify file revision by the number of the checkin). Then the set F_{plain}^{trunk} = {f^T[C45]} consists from a single file revision. Correspondingly, the set F_{plain}^{fb} = {f^{fb1}[C32], f^{fb1}[C36], f^{fb1}[C38], f^{fb2}[C30], f^{fb2}[C33], f^{fb2}[C35], f^{fb2}[C37]}. Here the number of checkins in feature branches $|F_{plain}^{fb}|$ is much greater than $|F_{plain}^{trunk}|$ because the number of plain checkins in the trunk is normally small in comparison to feature branches.

The approach described above allows collecting the most comprehensive set of historic changes for a file, but its practical implementation might be complex. In particular, it requires finding integration sources for the checkins in the trunk (this information can be either provided by the VCS itself or can be mined using the algorithm [4]).

2. Collecting historic information from the trunk.

A simpler solution is collecting all the checkins in the trunk branch, which, essentially, represent a cumulative set of changes in feature branches. Namely, we can define the set **$F=F_{all}^{trunk}$**={f^T[i],...,f^T[j]}, where {f^T[i],...,f^T[j]} are *all* checkins (including code integrations) that occurred in the trunk during the time window [d-w;d). In our example the set F_{all}^{trunk} for the checkin C51 is F_{all}^{trunk} ={f^T[C43], f^T[C44], f^T[C45]}.

Collecting history of checkins from the trunk is simpler and faster as it requires analyzing only a single branch. Unfortunately, a plain checkin and its integration into the trunk are usually performed by different developers. Collecting history of changes from the trunk will result in loss of information about the developer who made the actual checkin. As a result, organization metrics will become less informative.

3. Collecting historic information from the feature branch where the checkin was made.

Another solution would be collecting a set **$F=F_{plain}^{fb}$** = {f^{fb}[k],...,f^{fb}[l]} of plain checkins only across the feature branch where the checkin c was made. We collect only plain checkins since some of the file changes may be forward integrations from the trunk.

The set F_{plain}^{fb} does not include changes made to the file f in other branches. However, it may contain information which is more relevant to the checkin c because F_{plain}^{fb} was collected in the same branch as the checkin itself. In our example for the checkin C51 the set F_{plain}^{fb} ={f^{fb1}[C32], f^{fb1}[C36], f^{fb1}[C38]}.

Considering three different approaches to defining the set F of historic changes, an important research question arises: which approach is the best one? Namely, which of the sets F_{plain}^{all}, F_{all}^{trunk}, F_{plain}^{fb} will produce metrics that allow predicting of risky checkins with the highest accuracy? We answer this question while experimenting with our risk model.

VI. Model building

As a part of our work on the risk prediction model we had to answer following research questions:

Q1: Which metrics are the best predictors of risk for the pre-release checkins?

Q2: How historic data must be collected across multiple code branches? In particular, should we collect this data across all the branches, in the trunk branch only, or in the branch where the checkin was made?

A. Model definition

Risk prediction can be seen as a binary classification problem, where each checkin must be classified as non-risky or risky. More formally, for each checkin the model accepts the vector $\vec{x}$ of metrics (predictor variables) and produces the

`

response y – a scalar variable which can be translated into a corresponding class label ("risky" or "non-risky").

Binary classifiers are usually built using some statistical modeling technique, e.g. a decision tree, a logistic regression, or a Support Vector Machine. Such statistical model must be trained using the existing data – a training set, which contains a number of $<\vec{x},y>$ tuples. For the purposes of training, the response variable y should take values from the set {0,1}, which corresponds to "non-risky" vs. "risky" class labels.

Once the model is trained, it can predict risk for checkins. However, as any non-ideal model can make prediction errors, there are four possible outcomes of classification:

- True Positive (TP): the checkin is risky and was classified as risky;
- False Positive (FP): the checkin is not risky, but was classified as risky;
- True Negative (TN): the checkin is not risky and was classified as not risky;
- False Negative (FN): the checkin is risky, but was classified as not risky.

Based on these outcomes, a number of metrics to measure classification performance have been developed [6], including:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = True\ Positive\ Rate\ (TPR) = \frac{TP}{TP + FN}$$

$$False\ Positive\ Rate\ (FPR) = \frac{FP}{FP + TN}$$

A good classifier must have a high precision and recall, and low false positive rate.

Measuring prediction accuracy is simple if the classifier outputs class labels directly. However, most classifiers output response y as a scalar number, which represents a degree of belief that the checkin is risky. In order to translate y into the binary class label, some threshold must be used. Namely, the checkin is considered risky if the output of the classifier is higher than the threshold and as non-risky otherwise.

Obviously, the outcome of the classification will depend on the value of the threshold. To measure the change of the classifier's performance depending on the threshold, a number of techniques have been developed, including Precision-Recall graphs and Receiver Operating Characteristic (ROC).

A *ROC graph* [6] is a two-dimensional graph, where TPR (recall) is plotted on the Y axis and FPR is plotted on the X axis. The ROC curve for an ideal classifier is a straight line from (0,1) to (1,1), while the line from (0,0) to (1,1) implies a worst possible classifier that is equal to a random guessing. The a*rea under ROC curve (AUC)* can serve as a single number to measure classifier's performance. It can vary from 0.5 for the worst possible classifier to 1.0 for the ideal one.

B. Model building

To train the model and verify its accuracy we used data from the previous version of Microsoft Windows Phone 7 (WP7) system – a mobile OS developed by Microsoft. WP7 is a large system composed of tens of millions lines of code. Development of WP7 was parallelized across several branches.

We scanned all the WP7 code branches and collected data on all the plain pre-release checkins made between April 1, 2009 and September 1, 2010. This constitute last 17 months of Windows Phone 7 development. These pre-release checkins formed the training set for our model. For each of these checkins we collected code metrics, which formed the vector $\vec{x}$, and calculated the risk index (CRI). To calculate the CRI we used data on bug fixes made between September 9, 2010 and January 1, 2012. For the purposes of the model training, we considered checkins with the CRI>2 as risky (y =1). Collecting training data using a simplified modification of SZZ took 4-6 hours on the computer equipped with 2.4 GHz quad-core Xeon CPU and required about 5 GB RAM.

We built the model using two different statistical methods: a logistic regression [7] and a C4.5 classification tree [6]. To measure the accuracy of these models, we relied on the 10-fold cross-validation [6]. For each fold, we have built the model and calculated its area under the ROC curve (AUC). Then results were averaged across all the folds, so the mean $\mu(\mathbf{AUC})$ and the standard deviation $\sigma(\mathbf{AUC})$ of the AUC could be computed. We used this approach to answer questions Q1-Q2:

Q1: Which metrics are the best predictors of risk for pre-release checkins?

To answer this question we have built a number of models. Each model was built using different groups of checkin metrics described in the Section V. These metrics were collected across all the branches in the system. To avoid overfitting, we used a stepwise procedure [7] to select metrics into each model. The results are presented in the 0Corresponding ROC graphs are shown at Figure 3.

Experimental results clearly show that ***our model predicts risk for code changes with high degree of the accuracy***. In fact, the *full model* (one that includes all the metric groups) has the AUC = 0.934 (logistic regression, see Figure 4c) and 0.952 (C4.5 tree). This accuracy is very close to the accuracy of the ideal classifier (AUC=1.0), which ***proves the validity of our approach.***

To ensure that differences between various models are statistically significant, we performed t-tests between the AUC values of models built with different metric groups (we compared AUCs of models built using same statistical methods to avoid bias introduced by different modeling techniques). The null hypothesis was that the mean AUC obtained with a certain group of metrics is higher than the mean AUC obtained

TABLE I. COMPARISON OF DIFFERENT GROUPS OF METRICS

Metrics group	**Logistic regression**		**C4.5 tree**	
	μ(AUC)	*σ(AUC)*	*μ(AUC)*	*σ(AUC)*
Change size	0.914	0.0094	0.931	0.0058
Historic churn (across all branches)	0.880	0.0049	0.861	0.0167
Code	0.862	0.0062	0.833	0.0277
Organization	0.764	0.0105	0.717	0.0102
Full model (all metrics)	**0.934**	**0.0050**	**0.952**	**0.0045**

`

with another group of metrics.

Tests have shown that the AUCs for various groups of metrics are different at the 0.05 confidence level (p=0.007 for historic code churn vs. code metrics for C4.5 tree and p<0.001 for all other metric pairs).

While comparing different groups of metrics we can clearly see that *change size metrics (see the Section B) outperform all other metric groups*. In fact, the accuracy of the model built using only change size metrics is just 4% worse than the accuracy of the full model. This is an important observation, since change size metrics are also easiest to collect.

The historic churn metrics, which represent the amount of previous changes across all the source files affected by the checkin (see the Section C), are the second most significant predictors of the checkin risk. The code metrics (see the Section A) are slightly less accurate.

Our subset of the organization metrics (see the Section D) proved to be far less effective predictor; its AUC varies from 0.717 (the C4.5 tree) to 0.764 (logistic). However, we have used only a limited set of organization metrics that could be mined directly from the VCS.

Q2: how historic data should be collected across multiple code branches?

We propose three different methodologies to collect historic data for a source file: collecting historic data across all the branches, collecting historic data in the trunk, and collecting historic data in the particular branch where the checkin was made (see the Section E).

Selecting an appropriate data collection methodology is an important question. These methodologies result in different values of historic churn and organization metrics, which are built using data on historic changes. Correspondingly, these differences can have an impact on the accuracy of the model. Furthermore, these methodologies vary in their implementation complexity. In particular, collecting historic data across all the branches requires tracking all the integrations, which is a complex and computation-intense operation.

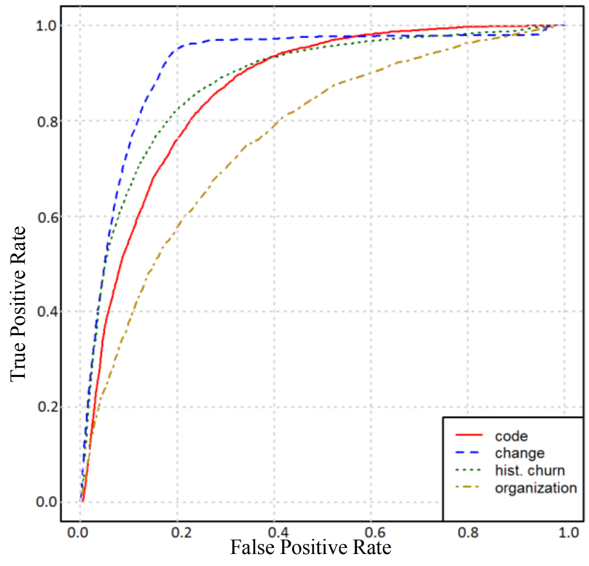


Figure 3: ROC graphs for different groups of metrics

To answer the question Q2 we have built models that use only historic churn and organization metrics to predict risk of the checkin. We have built three different groups of models (see Figure 4):

1. Historic metrics are collected from all the code branches;
2. Historic metrics are collected from the trunk branch;
3. Historic metrics are collected from the branch where the checkin was made.

TABLE II. compares accuracy of these models. We have found that the historic churn collected in the trunk branch allows for a higher accuracy than churn collected in the branch where the checkin was made (p<0.001 for both logistic regression and C4.5 tree). This behavior is expected since trunk contains a cumulative set of all the code changes, while the feature branch contains only some of them. However, historic churn collected across all the branches did not show a clear superiority over the churn collected in the trunk. The AUC of corresponding classifiers did not differ significantly (p=0.047 for the logistic regression and p=0.522 for the C4.5 tree). However, the ROC curves demonstrate that the classifier built using historic churn from the trunk predicts high-risk checkins slightly more accurately, while classifier built using churn from all the branches does better job predicting low-risk checkins. This observation might allow developing a classifier ensemble that will combine advantages of both classifiers.

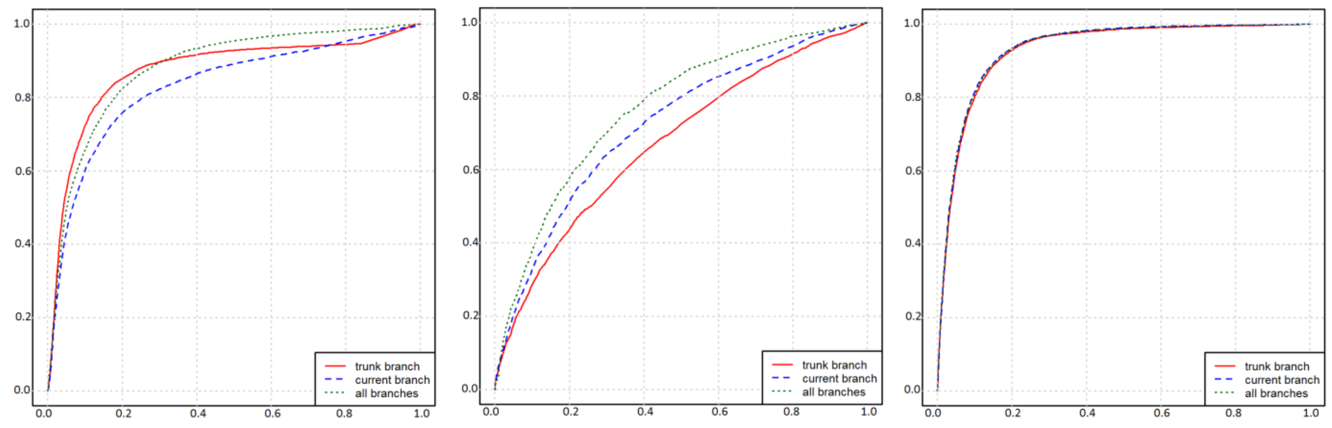
On the contrary, the model where organization metrics were collected across all the branches proved to be more accurate than the model where organization metrics were collected in the branch where the checkin was made (p<0.001 for both logistic regression and C4.5 tree). In turn, the latter model proved to be superior to the model where organization metrics were collected in the trunk branch (p<0.001 for both logistic regression and C4.5 tree). This can be explained by the fact that trunk branch does not provide the information about the actual developer who implemented the checkin.

As a result, from the point of prediction accuracy *historic code churn should be collected over the trunk branch or across all the branches, while organization metrics should be*

TABLE II. COMPARISON OF METHODS FOR COLLECTING HISTORIC DATA

Metrics group	Logistic regression		C4.5 tree	
	μ(AUC)	*σ(AUC)*	*μ(AUC)*	*σ(AUC)*
Change history collected across all the branches				
Historic churn	0.880	0.0049	0.861	0.0167
Organization	0.764	0.0105	0.717	0.0102
Full model	**0.934**	**0.0050**	**0.952**	**0.0045**
Change history collected across the trunk branch				
Historic churn	0.873	0.0106	0.862	0.0169
Organization	0.672	0.0130	0.637	0.0124
Full model	**0.934**	**0.0065**	**0.946**	**0.0074**
Change history collected across the branch where the checkin was made				
Historic churn	0.830	0.0105	0.811	0.0101
Organization	0.721	0.0128	0.679	0.0122
Full model	**0.931**	**0.0069**	**0.952**	**0.0053**

`



a) Historic churn metrics b) organization metrics c) all metrics (the full model)

b) Figure 4: Comparison of method to collect history on the accuracy of the model

collected across all the branches. However, differences in data collection methodology do not have a strong influence on the accuracy of the full model. In all cases except one there is no statistically significant impact of the historic data collection method on the accuracy of full models (p>0.05).

C. Model deployment

We have deployed a simplified version of our risk model to predict the risk of Windows Phone 8 code changes. To build the final model we used LogitBoost regression [8] – a boosted version of the regular logistic regression. Aside from the generally higher classification accuracy inherent to the boosted statistical methods, the advantage of the LogitBoost is an ability to select the most important predictors automatically.

To reduce the amount of time necessary for metrics collection, the deployed model incorporates only change size and code metrics, which do not require collecting history of code changes:

- Δ_file_chg(c): the number of files changed by c;
- Δ_file_add(c): the number of files added by c;
- Δ_file_int(c): the number of files integrated by c;
- FUNC(c): the total number of functions in all the files affected by c;

Due to the insufficient time span between the release of WP8 and writing this paper we could not verify the accuracy of the model using WP8 data. 10-fold cross-validation using WP7 data has shown that our simplified model has AUC=0.923, which approaches the accuracy of the full model.

We have implemented the model as tool called CheckinMentor. CheckinMentor is as a web application that monitors and analyzes every checkin coming into the Windows Phone code base and computes its risk. When the user opens a CheckinMentor page, he or she can choose a set of checkins to be displayed using a set of filters. In particular, checkins can be filtered by their date of change, name of the branch, the checkin ID, and name of files in the checkins. Once the user has set up a filter, the CheckinMentor displays a list of checkins that match that filter along with their metrics and the predicted risk. The user can sort the resulting list based the date and time of the checkin, name of the engineer who made the change, file name, and predicted risk.

If the user clicks on a particular checkin, the CheckinMentor displays the checkin description and the list of affected files. For each file it shows the type of the change and the number of added, deleted, and changed code lines.

VII. Threats to Validity

We consider the change as “risky” if it introduces a failure. Although this is an intrinsically correct approach followed by all the similar models [3][9][13][24], its side effect is that a large commit might introduce faulty changes and touch failure-prone files just by the sheer size only.

This observation, supported by the fact that the size of the change is the strongest predictor of its risk, gives an impression of simplicity and inefficacy of the risk model. However, this impression is wrong. The model also takes into account other metrics, which increase its accuracy. And most importantly, the model *allows for objective quantification of the risk*, which makes its predictions more actionable and accurate then risk estimation done by humans.

Nevertheless, this observation points for directions for improving the model. In particular, the model should not just quantify the risk of a change, but also identify parts of the change that are most likely to introduce the failure. Another improvement might be to account for the testing effort, which may depend on the size of the change.

To label fixes we used a simplified implementation of the SZZ algorithm that works on the granularity of a source file. This dramatically improves the performance of the data collection, but can potentially increase the number of false positive cases (a valid checkin is considered to be a bug-introducing one). This is especially likely if the source file affected by the change is large.

To account for this we consider checkins, whose CRI is lower than an empirically chosen threshold, as non-risky. However, choosing an overly low threshold can increase the number of false positives, while setting overly high threshold will lead to increase in false negatives (missing some bug-introducing changes). A more systematic approach would be

`

executing the original SZZ along with our modification on the subset of data. The output of SZZ will be considered as a ground truth and used both to calibrate the threshold imposed on the CRI and to estimate accuracy of our approach.

Our conclusions regarding the predictive power of individual groups of checkin metrics are based on the study which involved only Windows Phone 7 as a test subject. WP7 is a mature product with an established engineering process, so relative importance of the metrics might be different for other types of software systems, e.g. open-source projects. This particularly applies to metrics that take into account branches in the software system.

VIII. Conclusion

In our work we have developed a methodology for identifying those pre-release code changes that can cause post-release failures. We paid special attention to ensure a wide applicability of our approach, so it can be used to predict risk for the wide range of systems and applications.

We characterized each checkin with a set of metrics and used a statistical model to predict risk of checkins. To collect data for building the model we modified SZZ algorithm which detects bug-introduced changes. Our version of SZZ has higher performance and could be applied for large systems consisting of millions LOC and tens of thousands of checkins.

We have demonstrated how effects of code branches can affect building the risk model. We have shown that risk should be predicted only for certain types of changes. We demonstrated how code flow across different branches can affect data collection and influence the values of certain checkin metrics, such as historic code churn and organization metrics. However, our experiments have shown that although code branching have a significant impact on the predictive power of these metrics, the change size and properties of the affected components are the best predictors of the checkin risk.

We verified our approach on Microsoft Windows Phone 7 – a mobile operating system from Microsoft. Our models can identify risky changes with high degree of the accuracy (area under ROC curve = 0.93-0.95). The model was deployed as a part of the Windows Phone 8 engineering process. We will verify the accuracy of the deployed model once data on Windows Phone 8 changes will become available.

Our next steps would be applying our technique to develop risk models for a wider variety of systems, such as Windows, Office, Bing, and other Microsoft products. Most importantly, we want to verify if trends we have discovered in Windows Phone 7 hold for a wide variety of the systems. In particular we would like to know if the importance of different metrics groups will change in other software systems. To do this we will replicate our study across a wider range of software systems, including open-source projects.

References

[1] S. Kim, T. Zimmermann, K. Pan, J. Whitehead, "Automatic Identification of Bug-Introducing Changes", *Proc. ASE'06*, pp. 81-90, 2006

[2] N. Nagappan, B. Murphy, V. Basili, "The influence of organizational structure on software quality", *Proc. ICSE'08*, pp. 521-530, 2010

[3] A. Tarvo, "Using Statistical Models to Predict Software Regressions", *Proc. ISSRE'08*, pp. 259-264, 2008

[4] A. Tarvo, T. Zimmermann, J. Czerwonka, "An Integration Resolution Algorithm for Mining Multiple Branches in Version Control Systems", *Proc. ICSM'11*, pp. 402-411, 2011

[5] T. Fawcett, "An Introduction to ROC analysis", *Pattern Recognition Letters*, 26, 2006, pp. 861–874

[6] Hand D.J., Mannila H., Smyth P., "*Principles of Data Mining*", The MIT Press, 2001

[7] Larose D. T., "*Data Mining Methods and Models*", Wiley-Interscience, Hoboken, NJ , 2006

[8] M. Sumner, E. Frank, M. Hall, "Speeding up Logistic Model Tree Induction", *Proc. ECML-PKDD'05*, pp. 675-683, 2005.

[9] A. Mockus, D. Weiss, "Predicting risk of software changes", *Bell Labs Tech Journal*, Vol. 5 no. 2, 2000, pp. 169-180

[10] N. Nagappan, T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density", *Proc. ICSE'05*, pp. 284-292, 2005

[11] T. Zimmermann, N. Nagappan, "Predicting Defects Using Network Analysis on Dependency Graphs", *Proc. ICSE'08*, pp. 531-540, 2008

[12] C. Bird, N. Nagappan, P. Devanbu, H. Gall, B. Murphy, "Putting It All Together: Using Socio-technical Networks to Predict Failures", *Proc. ISSRE'09*, pp. 109-119, 2009

[13] S. Kim, E. J. Whitehead, Y. Zhang, "Classifying Software Changes: Clean or Buggy", IEEE Transactions on Software Engineering, Vol. 34 no. 2, 2008, pp. 181-196

[14] N. Nagappan, T. Ball, A. Zeller, "Mining Metrics to Predict Component Failures", *Proc. ICSE'06*, pp. 452-461, 2006

[15] E. Arisholm, L. C. Briand , "Predicting Fault-prone Components in a Java Legacy System", *Proc. ISESE'06,* pp.8-17,2006

[16] J. Munson, T. Khoshgoftaar, "The Detection of Fault-Prone Programs", IEEE Transactions on Software Engineering, vol. 18 no. 5, pp. 423-433, 1992

[17] T. Menzies, J. Greenwald, A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors", IEEE Transactions on Software Engineering, vol. 32 no 11, 2007

[18] V. Basili, L. Briand, W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators", IEEE Transactions on Software Engineering, vol. 32 no 11, pp. 751-761, 2007

[19] A. Hassan, "Predicting Faults Using Complexity of Code Changes", *Proc. ICSE'09*, pp. 78-88, 2009

[20] E. Weyuker, T. Ostrand, R. Bell, "Comparing the effectiveness of several modeling methods for fault prediction", Empirical Software Engineering vol. 15 no. 3, pp. 277-295, 2010

[21] S. Kim, T. Zimmermann, E. Whitehead, A. Zeller, "Predicting Faults from Cacned History", *Proc. ICSE'07*, pp. 489-498, 2007

[22] A. Mockus, R. Fielding, J. Herbsleb, "A Case Study of Open Source Software Development: the Apache Server", *Proc. ICSE'00*, pp. 263-272, 2000

[23] T. Mende, R. Koschke, "Effort-Aware Defect Prediction Models", *Proc. CSMR'10*, pp. 107-116, 2010

[24] E. Shihab, A. Hassan, B. Adams, Z.M. Jiang, "An Industrial Study on the Risk of Software Changes", *Proc. FSE'2012*, pp. 1-11, 2012

`