Clustering Static Analysis Defect Reports to Reduce Maintenance Costs

Zachary P. Fry and Westley Weimer
University of Virginia, Charlottesville, Virginia, USA
{zpf5a,weimer}@cs.virginia.edu

***Abstract*—Static analysis tools facilitate software maintenance by automatically identifying bugs in source code. However, for large systems, these tools often produce an overwhelming number of defect reports. Many of these defect reports are conceptually similar, but addressing each report separately costs developer effort and increases the maintenance burden. We propose to automatically cluster machine-generated defect reports so that similar bugs can be triaged and potentially fixed in aggregate. Our approach leverages both syntactic and structural information available in static bug reports to accurately cluster related reports, thus expediting the maintenance process.**

We evaluate our technique using 8,948 defect reports produced by the Coverity Static Analysis and FindBugs tools in both C and Java programs totaling over 14 million lines of code. We find that humans overwhelmingly agree that clusters of defect reports produced by our tool could be handled aggregately, thus reducing developer maintenance effort. Additionally, we show that our tool is not only capable of perfectly accurate clusters, but can also significantly reduce the number of defect reports that have to be manually examined by developers. For instance, at a level of 90% accuracy, our technique can reduce the number of individually inspected defect reports by 21.33% while other multi-language tools fail to obtain more than a 2% reduction.

I. Introduction

A critical part of the software maintenance process is identifying and addressing unknown defects. While defects can be reported by end-users or found during testing, such approaches are expensive and can typically only reveal bugs exercised during execution. In response to these limitations, many automatic bug finding techniques have been developed (e.g., [1], [2], [3], [4], [5], [6]).

However, false positives, spurious defect warnings, and duplicate defect reports can negate the potential time savings of such bug finding tools [7], [8]. While existing approaches have focused on reducing false positives and spurious warnings, the problem of duplicate defect reports *produced by automated tools* has not been investigated (cf. [9], [10], [11]). Such duplicate reports are an increasing problem in industrial practice: static analysis defect finders commonly require program-specific tuning to eliminate large groups of spurious reports. Even when programming patterns are ignored to reduce false reports, such tools can still produce groups of highly-related defects. We found that over 30% of the automatically produced defects examined in this paper were duplicates, suggesting that a technique capable of producing clusters of highly-related automatically produced defect reports could save a considerable amount of developer effort.

In this paper, we explore an automatic technique that clusters duplicate or related defect reports, increasing the utility of existing bug-finding tools. In this context, "related reports" are those that developers can triage at once, and ideally that all admit a similar fix; clustering related reports can thus save maintenance effort by allowing for handling similar reports in parallel. We believe that tool-generated defect reports can be clustered automatically because duplicate reports often share syntactic and structural similarities. Although some similarities can be syntactic, critically, not all duplicate defect reports involve syntactically-identical code that is easily identifiable as such [12]. While tools for detecting both syntactic similarity in code [13], [14], [15] and conceptual similarity in natural language descriptions of defects [9], [10], [11] are established, tools for finding syntactically different, semantically similar automatically generated defect reports are not.

We therefore propose a parametric technique to cluster defect reports using similarity metrics that leverage both syntactic and structural information produced by static bug finding tools. The technique takes as input a set of defect reports produced by a static bug finder and partitions it into clusters of related reports. The produced clusters must be accurate in order for the technique to be useful, because misclassification negates some portion of the provided time savings. We thus favor a clustering that maximizes the size of clusters produced (i.e., saves time) while ensuring that the clustered defect reports are in fact similar (i.e., is accurate). Because cluster size and accuracy are incomparable goals, our algorithm can be adjusted to favor one or the other as desired, producing a Pareto-optimal frontier of options.

We evaluate our technique on 8,948 defect reports produced by two popular static analysis techniques (Coverity [6], a commercial tool, and FindBugs [2], an open source tool) spanning eleven benchmarks totaling over 14 million lines of code in multiple languages. Since we are unaware of any previous techniques specifically designed to output clusters of defect reports produced by static analyses, we use existing code clone detection techniques as baselines. We find that our technique consistently clusters defect reports accurately.

The main contributions of this paper are as follows:

- We propose a lightweight, language-independent technique for clustering defect reports produced by existing state-of-the-art static defect detectors.
- We empirically compare our technique against code clone detection tools using defect reports from both Coverity's Static Analyzer and FindBugs on large programs written in both C and Java. Our tool is capable of larger reductions in the overall set of defect reports when cluster accuracy is required.
- To ground our technique, we present a human survey in which participants overwhelmingly agree (99% of

978-1-4799-2931-3/13 © 2013 IEEE

WCRE 2013, Koblenz, Germany

the time) that clusters produced by our technique in fact contain reports that should be triaged together.

II. Motivation

Static defect reports typically contain both implicated source code lines and structural (or semantic) information related to the reported defect. The use of the term "semantic" to describe statically obtained information about potential defects here may be somewhat misleading, thus henceforth we describe such information as *structural* to avoid confusion. State-of-the-art duplicate detection techniques target human-written reports or source code respectively and do not perform well on, or are not applicable to, the information produced by static-analysis defect detectors (see Section VI for more detail). Instead, our proposed technique exploits the structure of the information present in automatically generated defect reports to identify highly related clusters. In this section, we motivate this approach by presenting examples of automatically identified defect reports that are conceptually similar but exhibit syntactic discrepancies, and show how additional structural information can help us identify which of the example reports are related.

Coverity's Static Analysis tool ("Coverity SA") is a multi-language commercial bug finder that uses semantic path information to pinpoint likely bugs, matching known faulty semantic patterns [6]. Coverity SA reports bugs by outputting the type of error suspected and the given error's location in the code. We ran Coverity SA on version 2.6.15 of the Linux kernel and it reported over 1,500 candidate defects. We show three example reports from two separate files in Figure 1. We show only the file, suspected programmatic element, and implicated source lines (highlighted) in the context of surrounding code for ease of presentation.

There are several syntactic similarities to note in the given code. For instance, the immediate context of the code implicated by all three reports contain multiple references to a variable named `lp`. Additionally, reports A and C share calls to the `printk` debugging function while reports B and C share calls to the `isdn_dc2minor` function. However, there are obvious syntactic differences between all three reports as well. For example, parameters to the shared function calls are different across all three reports. While some might be tempted to group reports A and C due to the abundance of syntactic similarities, others might group reports B and C because of the similarity in function calls in the lines directly implicated.

As the syntactic information in the three reports fails to strongly indicate the presence or absence of conceptual similarity, we leverage the structural information in each report to get a more definite measure of relatedness. For example, we note that reports B and C are not only in the same file, but implicate the same programmatic element, `isdn_dc2minor`, as the source of the error. By comparison, report A shares few structural similarities with either reports B or C.

In reality, the `isdn_dc2minor` function called in reports B and C fails to check for negative values before returning and uses `-1` as its default return value. There are no corresponding checks for negative values at either call site shown before attempting to use the result to index arrays, and thus reports B and C represent actual defects. By contrast, the code in report A cannot suffer from the same type of error because an if statement ensures `lp->ppp_slot` is in bounds. A developer or tool using only syntactic information may conclude that the three defect reports are all related because of shared function calls and variables, or completely unrelated because of an abundance of unique program identifiers and code structure. However, the structural similarities between defect reports B and C provide more conclusive evidence that B and C describe a related defect while report A describes an orthogonal problem. We aim to leverage such information in our technique to identify such relationships quickly and automatically cluster defect reports that may be conceptually related but syntactically distinct.

Inspecting reports B and C aggregately can save time. For example, a check for negative values in B could be easily adapted to work in C, or a patch might restructure negative return handling in `isdn_dc2minor` and thus address both call sites. Both candidate fixes represent potential time savings (i.e., either because the same negative value check can be inserted in two places or because one fix to the callee affects multiple callers) and thus clustering reports like these can reduce maintenance effort.

Such a clustering technique could be used during both report triage and defect fixing to expose similarities that may not be obvious from manual inspection. Using such a tool allows developers to triage and fix similar defects in parallel, thus saving maintenance effort overall.

III. Methodology

We propose a similarity model for automatically-reported defects, allowing for the use of off-the-shelf clustering algorithms. Our model considers both syntactic and structural judgments of relatedness, using information reported by static analysis tools. We first outline the structure of an automatically produced defect report and describe how to extract the relevant pieces of syntactic and structural information (Section III-A). Once we have obtained the structured information from the defect reports, we measure defect similarity by systematically comparing the sub-parts of defect reports by both adapting existing techniques and introducing novel similarity metrics (Section III-B). We then learn a descriptive model of defect report similarity using linear regression (Section III-C) and explain how to use the resulting similarity measures to then cluster related defect reports (Section III-D).

A. Modeling Static Analysis Defect Reports

In Section II we introduced the Coverity SA bug finder; in practice there are many tools that report candidate software defects (e.g., [1], [3], [4], [5]). In this paper we also focus on FindBugs [2], an open source, lightweight static bug finder that uses pattern recognition to find known faulty code sequences. Similar to Coverity SA, it reports the type and location of the suspected defect. A Coverity SA or FindBugs defect report can be viewed as 5-tuple $\langle \mathcal{D}, \mathcal{L}, \mathcal{P}, \mathcal{F}, \mathcal{M} \rangle$ where: $\mathcal{D}$ is a free-form string naming the *defect type*; $\mathcal{L}$ is a $\langle source\_file, line\_number \rangle$ pair representing the *line* directly implicated by the tool as containing the defect in question; $\mathcal{P}$ is a sequence of zero or more $\langle source\_file, line\_number \rangle$ pairs, encoding a static execution *path* of lines that may be visited when exercising the defect; $\mathcal{F}$ is a string naming the nearest enclosing *function*, class or file to $\mathcal{L}$; and $\mathcal{M}$ is a set of zero or more free-form strings holding any additional *meta-information* reported

Defect Report A:
File:
`/drivers/isdn/i4l/isdn_ppp.c`
Suspect Variable:
`lp->ppp_slot`

```
printk(KERN_DEBUG "Receive CCP
  frame from peer slot(%d)",
  lp->ppp_slot);
if (lp->ppp_slot < 0 ||
    lp->ppp_slot > ISDN_MAX) {
  printk(KERN_ERR "%s:
    lp->ppp_slot (%d) out of
    range", _FUNCTION_,
    lp->ppp_slot);
  return;
}
is = ippp_table[lp->ppp_slot];
isdn_ppp_frame_log('ccp-rcv',
  skb->data, skb->len, 32,
```

Defect Report B:
File:
`/drivers/isdn/i4l/isdn_net.c`
Suspect Variable:
`isdn_dc2minor`

```
if (!lp->master)
  qdisc_reset(lp->netdev->
    dev.qdisc);
lp->dialstate = 0;
dev->st_netdev[isdn_dc2minor(
  lp->isdn_device,
  lp->isdn_channel)] = NULL;}
isdn_free_channel(
    lp->isdn_device,
    lp->isdn_channel,
    ISDN_USAGE_NET);
lp->flags &=
  ~ISDN_NET_CONNECTED;
```

Defect Report C:
File:
`/drivers/isdn/i4l/isdn_net.c`
Suspect Variable:
`isdn_dc2minor`

```
 sidx = isdn_dc2minor(di, 1);
#ifdef ISDN_DEBUG_NET_ICALL
 printk(KERN_DEBUG ``n_fi: ch=0\n'');
#endif

 if (USG_NONE(dev->usage[sidx])){
 if (dev->usage[sidx] &
       ISDN_USAGE_EXCLUSIVE) {
   printk(KERN_DEBUG ``n_fi: 2nd
     channel is down and bound\n'');
   if ((lp->pre_device == di) &&
       (lp->pre_channel == 1)) {
```

Fig. 1. Example Linux defect reports produced by Coverity's Static Analysis. The information presented is a mix of syntactic (e.g., the implicated code) and structural information (e.g., the suspected defective execution path and programmatic source of the defect). Syntactically, there are both similarities and differences between all three reports. When considering structural information, it appears that reports B and C share commonalities while A differs from both.

by the analysis (e.g., optional defect sub-types, categorical information for given lines of code, or suspected sources of the defect). Our technique operates on reports produced by any analysis that follow this format (or a subset of it, e.g., [4], [16]), regardless of defect-finding strategy.

The defect report components provide several potential sources of both structural and syntactic information that may be used in measuring the similarity between two reports. We use certain pieces of information exactly as they appear in a defect report, and coerce others to maximize the utility of the information extracted. The following paragraphs detail the specific types of information used to measure report similarity.

Function — Taken verbatim from $\mathcal{F}$, this string represents the name of the nearest enclosing function of the line indicated to be the manifestation of the defect. When a defect is reported outside a function, we use the enclosing class or file.

Path Lines — This information is a sequence of strings representing the source code lines implicated in a static path that may be executed to reach the site of the defect ($\mathcal{P}$ in our model). We hypothesize that errors on the same or similar execution paths may be related. Beyond comparing these path sequences explicitly, we additionally sort the source lines in $\mathcal{P}$ alphabetically to help expose defects that implicate similar lines of code but in different orders.

Code Context — Given the exact line indicated in $\mathcal{L}$ as the manifestation of the bug, the code context is the sequence of strings representing the three preceding and three following lines as they appear in the original source file. This window of code is an approximation of the context of the bug. We hypothesize that defects that occur in similar contexts (e.g., inside a `try/catch` block) may be similar.

Macros — By extracting all tokens containing *only* capital letters or digits from the actual source line text referenced in both $\mathcal{L}$ and $\mathcal{P}$ above, this information approximates the set of macros referenced in any indicated code line. Finding the exact set of macros in code requires a preprocessor and is prohibitively expensive. We thus use an approximation: While some tokens that appear in all capital letters may not actually be macros, a random check of 20 such instances showed that 85% indeed were. We hypothesize that the use of the same macros may indicate similarity between defect reports.

File System Path — This information is a string representing the exact path of the indicated file (taken from $\mathcal{L}$) in the given project's file structure which attempts to link defects that are in the same module or even sub-folder. Similar to the enclosing function, we hypothesize that defect reports indicating locally-close files may exhibit similarity.

Meta-tags — When available, this is a set of strings taken directly from $\mathcal{M}$: any additional information from the static analysis tool. With respect to the defect reports presented in Section II, this information includes the suspected source of the defect. Depending on the tool being used to find defects, the type and amount of information can vary widely. We hypothesize that *any* information produced by the static analysis tools *may* be useful when measuring similarity.

B. Defect Report Similarity Metrics

We propose a set of lightweight similarity metrics for tool-reported defects $\langle \mathcal{D}, \mathcal{L}, \mathcal{P}, \mathcal{F}, \mathcal{M} \rangle$ that are collectively applicable for both syntactic and structural information. Since we are interested in relationships *between* defect reports, the basic unit over which we measure similarity is a *pair* of defect reports. We determine an overall similarity rating for two defect reports by computing a weighted sum (Section III-C) of the similarities of their individual sub-components (described in Section III-A). This similarity model allows us to *cluster* related defect reports (Section III-D).

We use metrics from information retrieval and natural language processing in addition to introducing novel lightweight similarity metrics specifically applicable to the structure of the information present in this domain to compare individual defect report sub-components. Unless otherwise specified, we tokenize raw strings by splitting on whitespace and punctuation. The metrics we consider are described in the following paragraphs.

Exact Equality — a character-wise boolean match of two strings. Intuitively, reports with exactly-matching sub-components are likely related.

Strict Pairwise Comparison — the percentage of tokens from two strings that match exactly (comparing a_i to b_i for two token sequences a and b). When comparing textual lines of code, for instance, this metric can identify similar code that differs only in a few variable names or method calls.

Levenshtein Edit Distance — adapted from the information retrieval community, this metric in an approximate string matching technique measuring the number of incremental changes necessary to transform one string into another [17]. We lift the traditional metric, which operates on strings of characters, to sequences of tokens. Working over the alphabet of all tokens in either string, we count the number of token-level changes to transform one string into the other. Levenshtein distance relaxes a strict pairwise comparison, allowing approximate alignments. Spell checkers often use a similar method for suggesting replacement words for misspellings. Conceptually, our lifted Levenshtein distance is similar: it suggests defect reports with information that may be related.

TF-IDF — a document similarity metric common in the natural language processing community. It rewards tokens unique to the two documents in question and discounts tokens that appear frequently in a global context [18]. The use of TF-IDF assumes the existence of a representative corpus from which to measure the relative global frequency of all tokens. We take as this corpus the set of all tokens from all tool-produced defect reports to be clustered for a given program. We compare two documents by inspecting the *term frequency* (tf) and *inverse document frequency* (idf) of each token individually. *Term frequency* measures the relative count of all words while *inverse document frequency* measures the "uniqueness" of terms and discounts common words like *int* or *the*, while weighting unique and thus potentially more meaningful words highly. For example, referring back to the defect reports presented in Section II, the token `isdn_dc2minor` occurs in both reports B and C, but only in 0.69% of reports overall. Sharing this rare token increases the TF-IDF measure between these two reports, exposing an inherent similarity. By contrast, the token `lp` occurs frequently in all three reports and in 4.95% of others for Linux: it thus has a lower idf value. This prevents TF-IDF from mistakenly indicating as similar all defect reports with this globally-frequent token.

Largest Common Pairwise Prefix — the number of tokens two strings have in common when comparing each from left to right (i.e., the largest i such that $\forall j.\ 0 \leq j \leq i \implies a_j = b_j$). To illustrate the utility of this metric, consider two statements that assign the results of similar function calls to the same variable. Even if the function calls' parameters differ, this metric will capture the initial similarity between the two lines. Put differently, the way programs are written sometimes corresponds loosely to English, where the subject and verb usually appear towards the beginning of a sentence. Similarly, the left-most columns in many high-level programming languages (e.g., generally, the variable being assigned to or the root object of a method call) are the most fundamental to the execution and state of a program. For these reasons, we hypothesize that checking for similarities between the prefixes of code-based information might expose related defect reports.

Punctuation Edit Distance — a lightweight metric for structural code similarity. Traditional methods, like comparing control flow graphs, are expensive and are made difficult because compilation may be not be available on all projects during the triage stage. We instead adopt a lightweight metric that approximates program structure while retaining consideration for the sequence in which programmatic events occur. We compute the Levenshtein edit distance between token sequences with all non-punctuation removed (e.g., only curly braces, parentheses, operators, etc. remain). As an example of the utility of such a metric, consider two pieces of code that share both the same method calls and similarly structured loops. A similar pattern of parenthesis, commas, curly braces, and semicolons will help make the relatedness evident. By abstracting away textual identifiers, this metric complements more language-focused notions.

These metrics operate on a variety of input types. We coerce one type of information to another when necessary. For example, any string or set of strings can be viewed as a "bag of words" (the document data structure used by TF-IDF) by splitting on punctuation and whitespace while aggregating term frequency counts. Similarly, a set of strings can be coerced into a sequence (used by Levenshtein edit distance, for instance) by sorting them in order of textual appearance or alpha-numerically.

C. Modeling Report Similarity

The textual code-based and structural programmatic features outlined in Section III-A serve as input to the similarity measurements, allowing us to compare sub-components of two automatically-generated defect reports. We apply each similarity metric to all pairs of applicable report sub-components to obtain similarities for each pair of reports. We elide combinations with little or no predictive power for simplicity.

We avoid asserting an *a priori* relationship between these measurements and whether a pair of defect reports are related. Instead, we build a classifier that examines a candidate report pair and, based on a learned linear combination of weighted feature values, determines whether the pair is "similar." Thus, the similarity judgment for a pair of defect reports is a sum of weighted features (where each f_i is similarity metric value for a pair of report sub-components):

$$c_0 + c_1 f_1 + c_2 f_2 + \ldots + c_n f_n > c' \quad (1)$$

Two defects are called "similar" if the resulting aggregate sum is greater than an experimentally chosen cutoff: c'. We use linear regression to learn values for c_0, c_1 through c_n, and c'. A training stage, detailed in Section IV, is required to learn this classifier. We choose a linear model to allow for exploration of a series of smooth cutoffs given a single model.

D. Clustering Process

A cluster of defect reports wherein each individual report is "similar" (with respect to our model) to all others is amenable to aggregate triage. Having defined similarity between defects reports, we now require a lightweight and accurate method for clustering related defects. Traditional clustering techniques (e.g. k-medoid clustering) often try to measure the similarity between single entities given a formal metric space. Specifically, k-medoid clustering assumes that all features are real-valued and weighted equally in the model. First, we do not assume that the features in our model warrant equal weighting (as evidenced by the learned coefficients in

our similarity model). Additionally, our features are not real-valued measures of an individual defect's properties, but rather relative measures of similarity. We thus adopt a well-known algorithm for measuring interconnectedness of components for the purpose of clustering.

One can view a cluster of similar defects as an undirected graph where the vertices represent defects and the edges represent the similarity relationship (that is, any connected vertices are considered "similar" using equation (1)). To prefer accurate clusters and avoid falsely clustering unrelated defects, clustering can be performed by finding maximum cliques in the induced graph [19]. Finding cliques ensures that any defect in the clique (cluster) will be similar to all other defects therein.

We propose a 2-phase recursive approach to clustering:

1) Construct an undirected graph where the vertices represent all remaining, unclustered defects and the edges signify our definition of "similarity".
2) Find the maximum clique, output all included defects as a cluster and remove them from the graph; return if no defects remain, otherwise recurse.

In the worst case, clique finding requires exponential time: the time complexity is $\mathcal{O}(n \times 2^n)$ where n is the number of vertices in the overall graph. In practice, "almost-cliques" are rare (i.e., spurious interconnecting edges between clusters are sparse when a high "similarity" cutoff is chosen) and our implementation runs sufficiently fast. For example, on a Linux kernel module with 869 defect reports, the average run time was 0.088 seconds. This approach to clustering produces distinct sets of defects that display a high degree of internal similarity, as we show in the next section.

IV. Evaluation

We seek to evaluate our technique's utility when clustering defect reports and also put it in context with relevant work. We thus address four research questions:

- *R1*: How effective is our technique at accurately clustering defect reports produced by off-the-shelf static analysis tools?
- *R2*: Does our approach outperform existing code clone detection techniques when clustering defect reports?
- *R3*: How does our technique perform across different static analysis tools and different languages?
- *R4*: Do humans agree with the clusters produced by our technique?

For the purposes of these experiments, we collected defect reports from eleven C and Java programs comprising over 14 million lines of code and yielding over 8,000 defect reports from Coverity SA and FindBugs. Further details of these benchmark programs can be found in Table I.

TABLE I. Test programs and defect reports used to evaluate our algorithm. The top group of programs are written in C while the bottom group is written in Java. Note: the KLOC totals represent the number of lines analyzed by the bug finders and might be smaller than the total number of lines in the projects.

Program	Version	KLOC	Reports	Description
Blender	2.45	996	827	3D content creation suite
GDB	6.7	1,689	827	Multi-language debugger
Linux (fs)	2.6.15	521	175	Linux OS Filesystem module
Linux (sound)	2.6.15	420	869	Linux OS Sound module
Linux (other)	2.6.15	4,263	214	All other Linux OS modules
MPlayer	1.0rc2	845	500	Media player
Perl	5.8.8	430	63	Perl language interpreter
Ruby	1.8.6-p111	194	75	Ruby language interpreter
Xine	1.1.10.1	499	292	Media player
	Totals:	**9,862**	**3842**	
Bcel	5.1	56	238	Byte Code Engineering Lib
Eclipse	3.1.2	3,618	4345	Programming IDE
JFreeChart	1.0.1	211	338	Chart toolkit
Spring	2.0.8	430	185	Java application framework
	Totals:	**4,316**	**5106**	

A. Learning a Model

First, we construct a model that, given a set of similarity measurements between the sub-components of two candidate defect reports, determines whether the two reports should be considered highly related. We use linear regression to learn the coefficients c_i and the cutoff c', such that the model declares two reports similar according to equation (1).

Linear regression requires training data consisting of the measured features annotated with the response variable (i.e., the "correct" answers). The response variable for our model is defect report similarity — a human judgment. Because such a judgment cannot be automatically measured, we hand-annotated all combinations of defect report pairs (where only defect reports of the same "type" could be potentially clustered) to serve as a ground truth when training and testing the model. Our goal is to cluster not just syntactically similar defect reports, but also those that are related semantically. When annotating the data set, we therefore deemed two defect reports "similar" if any of the following criteria were met:

1) the code contexts displayed significant syntactic similarity while implicating the same defect
2) the implicated code for both reports was semantically related such that the underlying causes of the defects were the same
3) the reported defects' code exhibited semantic similarities such that the defects would manifest in the same way

We mitigate the threat of over-fitting our model by specifically training and testing on different sets of data. We randomly selected small subsets of the annotated defect report pairs for each benchmark program for the purpose of training. We then tested the model on the remaining data. As such, we gain confidence that our model is not simply encoding specific data points, but rather learning meaningful weights for the associated sub-component comparisons as intended.

An advantage of our approach is that it does not distinguish between "true positive defect reports" (real bugs) and "false positive defect reports" (spurious reports from the static analysis tool). When clustering reports to expedite triage, effort can be saved in both cases: false positives must be identified as such, and doing so aggregately saves maintenance effort.

B. Maintenance Savings versus Cluster Accuracy

The goal of our technique is to reduce maintenance effort by clustering tool-generated defect reports, allowing developers to triage and even fix defects in aggregate. In this section,

we evaluate the potential for effort savings associated with our tool (*question R1*). Additionally, we put our tool in context by comparing it with the closest related duplicate detection techniques (*question R2*).

1) Metrics: To evaluate both research questions, we use two distinct success metrics. First, we measure the average internal accuracy of all clusters produced. That is, for each proposed cluster we measure the ratio of the size of the largest contained clique (with respect to our ground-truth annotations) to the size of the cluster as produced by our technique. For example, a cluster of size five where only four reports are perfectly interrelated would have an accuracy of 0.8. Second, we compute the percent reduction in size of the overall set of defect reports when using the resulting clustering to handle and triage clustered defect reports aggregately. We "collapse" each emitted cluster into one effective defect report, assuming (given the stated definition of *related* defect reports) that similar reports can be handled in parallel. For example, if there are 20 original reports and an approach identifies two clusters of size five each, the resulting *effective* size is 12 conceptual clusters (10 singletons and 2 of size 5), making 40% reduction in the number of reports that must be considered separately and addressed using separate reasoning. We recognize that not all defects take the same amount of time or human effort to triage and fix and thus note that approximating the reduction in human effort based on the reduction in defect reports would be strictly an estimation. Gauging the effort needed to triage and fix any one defect [20] is orthogonal to this work and, as such, we simply measure a reduction in the number of reports for our evaluation.

2) Code clone tools: To our knowledge, there are no existing fully-automatic techniques for clustering defect reports produced by static bug finding tools. Kremenek *et al.* propose a defect report ranking technique based on clustering, but it relies on repeated human feedback and thus is not directly comparable to our technique [21]. However, code clone detection is a closely related task — reports implicating similar code (e.g., from copy-and-paste development, or just from similar development logic) may likely be related, and thus we can use such techniques as a baseline for comparison. Tool-generated defect reports contain an abundance of code-based information and thus, adapting code clone detection tools for this task provides a direct means of comparison. Additionally, code clone tools rely almost exclusively on syntactic string matching techniques and as such provide an excellent baseline for comparison: any increase in accuracy or the number of clustered defect reports exhibited by our tool can be attributed to our inclusion of structural information or use of diverse similarity metrics.

There are many state-of-the-art techniques capable of performing clone detection with high accuracy; we adapt three popular tools to compare against our technique: ConQAT, PMD, and Checkstyle [13], [14], [15]. These tools typically take as input a set of source files and produce a list of all code clones. We adapt them to defect report clustering by creating a set of synthesized source files, each deriving from an individual report. For a given defect report, we construct a synthesized source file by concatenating $\mathcal{L}$ (the source line implicated) and $\mathcal{P}$ (the implicated execution path source). The set of all synthesized files (corresponding to all defect reports in question) is used as input for the given clone detection tool, and we then use the code clone tools' output as a defect report

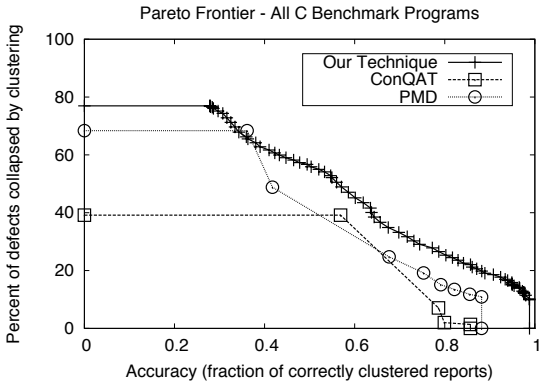

Fig. 2. Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate reduction in the number of defect reports from clustering for C benchmark programs.

similarity metric and perform clustering in the same manner as our technique (as described in Section III-D).

3) Results: Figure 2 and Figure 3 show the percent reduction of the overall set of defect reports when using each clustering approach at varying levels of cluster accuracy, split between C and Java defect reports. These results are presented in terms of Pareto-optimal frontiers to show the tradeoff between cluster accuracy and the number of distinct defect reports. Each point on a Pareto frontier represents a possible outcome for a parametric technique. Our approach admits more fine-grained adjustment than off-the-shelf code clone tools as it is parametric based on modeled similarity and not simply the size of the matches found in the code.

Our technique clusters more defects than comparable code clone detection techniques at nearly all levels of accuracy for both languages. When considering Java defect reports, our technique outperforms all code clone tools at all levels of accuracy. Additionally, our technique is capable of perfect accuracy (the bottom right portion of either graph), while the other tools are not. We note that while lower accuracies appear to yield large clusters and thus a great reduction of the overall set of defects, in practice, spurious reports in such clusters would greatly reduce the benefit of treating such defects in aggregate. We assert that higher levels of accuracy should be favored to reduce maintenance effort. We present the full spectrum of accuracy values for the sake of completeness.

Comparing the area under competing Pareto frontier curves provides a way to generalize performance across all tradeoffs. When considering all defect reports, the area under the curve for our technique is 1.4 and 2.5 times larger than ConQAT and PMD, the two multi-language code clone tools we consider, respectively (Checkstyle works only for Java programs and thus is not considered here). Code clone tools take into account mostly syntactic features, thus the increase in performance associated with our tool can likely be attributed to the inclusion of structural, semantically-related, features. We believe that the disparity in performance between the code clone tools and our techniques can be explained by clusters of conceptually similar but syntactically unique defect reports (see Section II).

C. Semantic Clustering Generality

Having demonstrated that our technique can reduce the number of distinct defect reports (e.g., to save developer

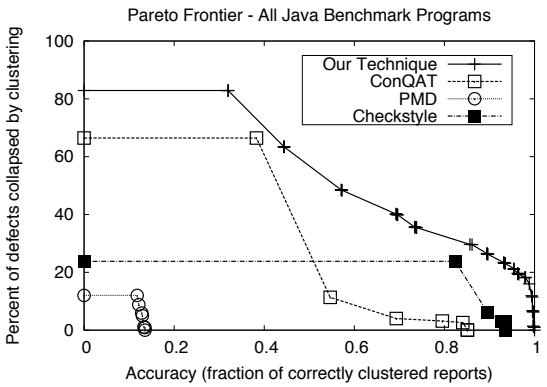

Fig. 3. Pareto frontier plotting our technique's accuracy when clustering defect reports as well as the aggregate reduction in the number of defect reports from clustering for Java benchmark programs.

maintenance effort), we investigate the differences in performance across languages and when considering defect reports produced by different static bug finding tools (*question R3*).

1) Different languages: While syntax-based code clone detectors exhibit varying levels of performance across languages, our technique generalizes with higher stability. Notably, the Pareto frontiers for our tool with respect to both C and Java defect reports share a similar shape and comparable levels of defect report set reduction at varying levels of accuracy while those of the code clone tools do not. Our model does not use language-specific features, and thus displays cross-language consistency. On average, ConQAT and PMD (the two code clone detectors that work on both C and Java code) show over 5 times more variance (in terms of under-curve area) across languages as compared to our technique.

2) Different tools: There are numerous qualitative differences between the defect reports produced by Coverity SA and FindBugs, thus it is not obvious that a given clustering technique will immediately generalize across static analyzers. Coverity SA yields semantically-rich data, typically producing non-empty $\mathcal{P}$, non-empty $\mathcal{F}$, and various additional information in $\mathcal{M}$. FindBugs, by contrast, often produces fewer suspected lines and even generalizes some types of defects to only the containing class, yielding $\mathcal{L}$ as the only line of code defining the associated defect. Similar to the cross-language comparison, our technique performs comparably on Java defect reports from both Coverity SA and FindBugs (FindBugs is not meant to run on C code). ConQAT, PMD, and Checkstyle exhibit variance comparable to that of our technique across different static bug finding tools, but our technique emits larger clusters (thus allowing for greater maintenance effort savings) at all levels of accuracy for both tools' defect reports. This further suggests that our semantically-rich tool is better suited to clustering multiple types of automatically-produced defect reports than are the most closely related code clone techniques.

3) Predictive Power of Structural Metrics: We hypothesize that the inclusion of structural features accounts for the relatively high performance of our technique. Table II presents the features used in our model along with the corresponding relative predictive power (or "quality measure") of each as measured by the ReliefF method [22], [23]. ReliefF does not assume linear independence of features and thus is appropriate given that some of our model's features may overlap as they derive from similar parts of the code or defect reports. ReliefF reports each feature's importance based on the *relative* magnitude of each feature's quality measure — larger numbers indicate more powerful features. Notably, some of the most predictive features use parts of the defect reports including the code path, the contextual window around the suspected defect, and the file system path, none of which are used by the code clone tools. This suggests that the use of structural information is beneficial when clustering duplicate automatically-generated defect reports. Macros and meta-tags proved to be weaker sources of information: we hypothesize that because not all defect reports contain this information, they may not be universally powerful predictors.

TABLE II. A LIST OF THE PREDICTIVE POWER OF THE SIMILARITY FEATURES USED BY OUR TECHNIQUE. THE "INFORMATION" COLUMN NOTES THE PART OF THE STATIC ANALYSIS OUTPUT BEING EXAMINED AND THE "MATCH TYPE" COLUMN INDICATES THE TYPE OF SIMILARITY METRIC USED. FEATURES' QUALITIES ARE MEASURED RELATIVE TO ONE ANOTHER, WHERE HIGHER VALUES INDICATE MORE PREDICTIVE POWER.

Information	Match Type	ReliefF
Path Lines	Strict Pairwise	0.0043
Code Context	Strict Pairwise	0.0042
File System Path	Common Prefix	0.0039
Code Context	Levenshtein	0.0022
Path Lines	TF-IDF	0.0021
Path Lines	Common Prefix	0.0020
Path Lines	Punctuation	0.0016
Path Lines, Sorted	Common Prefix	0.0016
Macros	TF-IDF	0.0008
Path Lines, Sorted	Levenshtein	0.0009
Path Lines	Levenshtein	0.0009
Meta-tags, Sorted	Levenshtein	0.0003

D. Cluster Quality

Clustering defect reports is advantageous only if the clusters contain reports that are, in fact, related. An incorrectly-clustered set of defect reports that is mistakenly triaged in the same way may negate some or all of the maintenance effort savings associated with clustering. We must verify that our clustering technique agrees with human maintenance judgments (*question R4*).

Our technique models a human cognitive notion of defect report similarity and thus any qualitative validation cluster accuracy should include human judgment. In Section IV-B we quantitatively show that our technique is capable of achieving high accuracy with respect to our human-annotated data set. In this subsection, we present evidence from a developer survey showing that our annotated data set is grounded in reality and thus that developers may benefit from using our approach.

Our survey goal was to evaluate our annotation technique and provide confidence that it generalizes. Focusing on clusters that should and should not be triaged together in practice, we presented 12 developers (graduate students and developers from industrial firms) from both academia and industry with 50 clusters of defect reports. We randomly selected 25 "accurate" clusters from those produced by our technique at accuracy greater than 90% (manually verified against the hand-annotated data set). We also randomly selected 25 "inaccurate" clusters selected from those produced by PMD with accuracy

less than 10%. For a given cluster, participants were provided with the type of defect being clustered and the code implicated in all related reports. Participants were shown all 50 clusters in a random order and asked to determine whether they believed the reports in a given cluster could be triaged and potentially fixed in the same way — that is, were the clustered reports likely representative of the same or highly related bugs? This high-level definition for the "similarity" of defect reports mimics the stated use case for our technique.

We hypothesize that humans would strongly agree with our annotations, thus validating the results presented in Section IV-B. We present results in terms of both raw agreement percentages and Randolph's free-marginal multirater kappa, an aggregate measure of inter-annotator agreement that does not assume a fixed distribution of categorizations for a given participant [24]. Free-marginal multirater kappa values range from -1.0 to 1.0, where a value of 1.0 represents perfect agreement and values greater than 0.8 indicate "*strong*" agreement. Participants agreed with our annotations with respect to "accurate" clusters 99% of the time (with a free-marginal multirater kappa agreement of 0.96), suggesting that our annotation process is grounded with respect to human judgments of defect report similarity. Conversely, participants showed more variability with respect to the PMD-generated clusters containing defect reports we annotated as "not accurate". Participants only agreed that clusters we annotated as "inaccurate" (or not related) were, in fact, related 44% of the time (free-marginal multirater kappa agreement of 0.28). This finding argues for a parametric technique such as ours: when high accuracy is demanded of our tool, humans show almost perfect agreement with it, but since human variability exists, some developers may prefer looser, and thus potentially larger, clusters.

We have shown that developers may prefer different levels of accuracy for defect report clustering and that our technique is capable of near-perfect accuracy. However, the increase in clustering size grows rapidly as small accuracy decreases are allowed. For instance, while the number of defects effectively removed from the overall set at 100% accuracy is 4.30%, at 95% accuracy, the savings jumps to 18.35% (an $4\times$ increase). By contrast, all three code clone tools fail to ever achieve 95% accuracy. The example cluster presented in Section II (i.e., a cluster containing defect reports B and C) was produced by our technique tuned to an accuracy level of 85%, further suggesting that perfect accuracy is not required for useful clustering.

E. Cluster Case Study

To further explore the quality of the clusters produced by our technique (*R4*), we present an example cluster of defect reports from the Eclipse project in Figure 4. The three defect reports presented are categorized as "forward null" defects, which suggests that the successful execution of a given statement necessarily indicates that the value of a specific variable in a following statement will be `null`.

Through careful inspection, we have concluded that these defects are not only false positives, but are also similar enough to be handled aggregately, saving maintenance effort. Notably, the Eclipse-specific `Assert.isTrue(...)` method called in three cases will throw an `AssertionFailedException` if the variable in question, `entry`, is ever null. This will interrupt execution, preventing the suspected defective lines from executing. Coverity's Static Analyzer is equipped with functionality to handle such system-specific idiosyncrasies, but has to be manually configured to do so. These three defect reports are additionally similar because the false positive is caused by a call to `Assert.isTrue(...)` and concerns a variable with the same name, created from the same source method call in all three cases. In practice, a developer presented with this cluster could quickly identify the commonalities and discard these three defect reports aggregately, reducing the required maintenance effort.

Additionally, these reports exemplify the utility of structural features and some of the shortcomings associated with syntactic-only models. All three defect reports exhibit the following structural similarities:

1) Spatial Locality — All three machine-generated defect reports indicate code in the CheatSheetStopWatch.java file within the UI module of the Eclipse code base.
2) Contextual Similarity — Each suspected defective code path is immediately preceded by two textually-identical lines of code (lines 2–3 in all three examples).
3) Punctuation Edit Distance — The indicated lines exhibit high similarity with respect to punctuation. For instance, lines 4 from each of the reports are identical when only punctuation is considered.

However, there are syntactic differences between these three defect reports that may make it difficult for syntax-only approaches to definitively indicate similarity:

1) While the statements spanning lines 4 through 6 in each report exhibit some similarities, the large string literals are unique in each case.
2) In all three cases, the suspected location of the respective errors (the last highlighted line in each code segment) are very different. Specifically, report X indicates an assignment statement, report Y an assertion, and report Z a conditional.

Our technique produced this cluster with perfect clustering accuracy and a 3.25% reduction of defect reports overall. Thus, a user requiring even the highest level of accuracy would be provided this cluster in practice. By contrast, only one of the code clone detection tools, ConQAT, also produced this cluster — at a level where its accuracy was 0.30. At this level of accuracy, 70% of reports clustered using ConQAT would be miscategorized, and much of the effort savings associated with clustering reports would be lost.

V. Threats to validity

While our experiments were designed to show the utility of our technique when clustering defect reports produced by different static analysis tools over large, open-source programs in several languages, our results may not generalize to industrial practice. First, our benchmark programs may not generalize to all industrial code. To mitigate this threat, we selected both large and small programs from varying domains spanning both C and Java. In addition, many of these benchmarks are used by Coverity for in-house testing, suggesting external belief in their generality.

Additionally, Coverity SA and FindBugs may not generalize to all static bug finders and thus our technique's performance may not generalize to all such tools. We attempted to

Defect Report X:

```
1  public void stop(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start != -1,
5     "start() must be called before using stop()");
6    entry.stop = System.currentTimeMillis();
7  }
```

Defect Report Y:

```
1  public long totalElapsedTime(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start != -1,
5    "start() must be called before using
6    totalElapsedTime()");
7    Assert.isTrue(entry.stop != -1, "stop() must be
8    called before using totalElapsedTime()");
9      //$NON-NLS-1$
10   return entry.stop - entry.start;
11 }
```

Defect Report Z:

```
1  public void lapTime(String key) {
2    Assert.isNotNull(key);
3    Entry entry = getEntry(key);
4    Assert.isTrue(entry == null || entry.start != -1,
5    "start() must be called before using lapTime()");
6    if(entry.currentLap == -1) {
7      entry.previousLap = entry.start;
8    } else {
9      entry.previousLap = entry.currentLap;
10   }
11   entry.currentLap = System.currentTimeMillis();
12 }
```

Fig. 4. Three defect reports from Coverity Static Analysis when run on version 3.1.2 of the Eclipse IDE. The highlighted lines are specifically implicated by Coverity SA as the suspected defective execution path while the additional lines provide context. In each case, the last highlighted line is the exact spot of the suspected defect.

mitigate this threat by designing our technique to operate on an abstract representation of defect reports and by testing our technique on two tools. Coverity SA is a commercial static bug finder that is semantically-rich and works across several languages. Comparatively, FindBugs is an open source pattern-based bug finder that is targeted at Java.

Finally, our method of manually annotating defect reports with respect to similarity may not generalize to the philosophy of all developers or systems. As noted in Section IV-A, defect report similarity is inherently a human judgment and thus different developers may have more or less strict ideas for what constitutes "similar" defect reports. We attempt to mitigate this threat in two ways. First, we designed our technique such that accuracy is adjustable parameter. Secondly, we asked multiple developers to assess on clusters produced by both our technique and a code clone tool. For clusters that we annotated as being "accurate" (thus, the defect reports are "similar"), developers agreed with our judgment 99% of the time.

VI. Related Work

Kremenek *et al.* proposed the only other cluster-based work with respect to automatically-created defect reports that we are aware of [21]. The technique exploits locality to cluster related defects with the goal of improving severity rankings and ultimately reduce false positive reports. In addition to locality, it specifically relies on iterative human feedback which is an additional cost our tool does incur. Comparatively, our technique examines a simple version of code locality while also considering many other features, showing that a combination of diverse features is effective at clustering defect reports in practice. Additionally, we focus on clustering *all* defects to reduce the overall size of the set developers have to triage and fix, while this previous work focuses specifically on reducing the number of false positives a developer would have to examine while using an automatic bug finding tool.

Recent work by Thung *et al.* categorizes fixed defects to characterize the types of bugs a system may be susceptible to in the future [25]. Their work differs from ours in goal and in input assumptions: their technique relies on features based on the actual fixes for the associated defects, which would not yet be available in our pre-triage and pre-fix usage scenario.

Duplicate manual defect reports and duplicate code have long been recognized as important issues in software engineering (e.g., [26]). While spurious manual defect reports are an obvious source of overhead, duplicated or semantically related source code also leads to higher defect densities and thus additional developer effort throughout the maintenance process [27]. There are many tools designed to find code clones for the purpose of removing or refactoring them to aid in future development [13], [14], [15]. Automatic techniques have also been developed to eliminate duplicated human-created bug reports, thus saving developers effort throughout the maintenance process [9], [10], [11]. Human-reported defects often contain a natural language description of the defect in question and optionally a stack trace or automated error output.

Automatic defect detection tools, by contrast, generally produce mostly structural and code-centric information when identifying potentially buggy statements in software. Techniques to detect manually-created duplicate reports generally focus on matching natural language information, while our technique focuses more on the structural similarities between different pieces of code. While duplicate detection has been studied comprehensively as it relates to both source code and manual defect reports, there is a notable lack of research in this area with respect to static analysis results. Additionally, in Section IV-B we show that using existing duplicate detection tools is insufficient for the problem of reducing the maintenance cost associated with automatically-generated defect reports. In fact, as few as 11% of manual defect reports contain stack traces (a code-based source of information) in practice [28]. As such, using tools designed to handle mostly natural language information for the purpose of detecting similar automatically-generated defect reports may not provide a valid means of comparison for our technique.

While our technique functions as a post-processing step to static bug finders, its success is determined by the precision and amount of information they produce. Many successful static bug finding tools have been developed in recent years [1], [2], [3], [4], [5], [6]. The task of actually finding defects in source code is orthogonal to the task of clustering said defect reports to speed up triage and bug fixing.

VII. Conclusion

We present a language-independent technique for clustering defect reports produced by static analysis-based bug finding tools. To the best of our knowledge, there are no existing tools specifically designed for such a task (e.g., tools for human-written reports focus instead on natural language), and we show that our tool is capable of clustering similar defect reports accurately to save maintenance effort. Our evaluation includes over 8,000 defect reports on over 14 million lines of code.

A quantitative evaluation shows that our tool outperforms state-of-the-art code clone tools adapted to the task of defect report clustering at nearly all levels of cluster accuracy. Additionally, our tool generalizes across defects found by both Coverity's Static Analysis tool and FindBugs in both C and Java programs. These results suggest that syntax-only approaches, like those used to find duplicate manual defect reports and code clones, are insufficient for the task of accurately clustering automatically-generated defect reports. Developers could use such a clustering technique when attempting to triage and fix defects to save maintenance effort by handling similar defect reports aggregately.

We also show that real world developers agree with our notion of an "accurate" cluster 99% of the time, thus suggesting our technique could be useful in practice. Furthermore, there is developer disagreement over "inaccurate" clusters, supporting our design decision that cluster accuracy be a tunable parameter. As bug-finding tools grow in popularity, processing their voluminous output becomes an increasing challenge: this paper presents, to our knowledge, the first technique for clustering tool-generated defect reports and argues that it is effective.

Acknowledgments

The authors are sincerely indebted to Andy Chou of Coverity for initial ideas, guidance, and technical support. We are also grateful to Claire Le Goues for insightful discussions on an earlier draft of this work. We acknowledge the partial support of NSF (CCF 0954024, CCF 0905373) AFOSR (FA9550-07-1-0532, FA9550-10-1-0277), and DARPA (P-1070-113237).

References

[1] Ballou, M.C.: Improving software quality to drive business agility. White paper, International Data Corporation (June 2008)

[2] Hovemeyer, D., Pugh, W.: Finding bugs is easy. In: Companion to the conference on Object-oriented programming systems, languages, and applications. (2004) 132–136

[3] Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Programming Language Design and Implementation. (2007) 89–100

[4] Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Principles of Programming Languages. (2002) 1–3

[5] Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Programming Language Design and Implementation. (2003) 141–154

[6] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.R.: A few billion lines of code later: Using static analysis to find bugs in the real world. Communications of the ACM **53**(2) (2010) 66–75

[7] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: Program Analysis for Software Tools and Engineering. (2007) 1–8

[8] Vetro, A., Torchiano, M., Morisio, M.: Assessing the precision of findbugs by mining java projects developed at a university. In: Mining Software Repositories. (2010) 110–113

[9] Jalbert, N., Weimer, W.: Automated duplicate detection for bug tracking systems. In: International Conference on Dependable Systems and Networks. (2008) 52–61

[10] Wang, X., Zhang, L., Xie, T., Anvik, J., Sun, J.: An approach to detecting duplicate bug reports using natural language and execution information. In: International Conference on Software Engineering. (2008) 461–470

[11] Sun, C., Lo, D., Wang, X., Jiang, J., Khoo, S.C.: A discriminative model approach for accurate duplicate bug report retrieval. In: International Conference on Software Engineering, ACM (2010) 45–54

[12] Engler, D.R., Chen, D.Y., Chou, A.: Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In: Symposium on Operating Systems Principles. (2001)

[13] ConQAT: Conqat. https://www.conqat.org/ (2011)

[14] PMD: Pmd. http://pmd.sourceforge.net/pmd-5.0.0/ (2012)

[15] Checkstyle: Checkstyle. http://checkstyle.sourceforge.net/ (2011)

[16] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. International Journal on Software Tools for Technology Transfer **9**(5) (October 2007) 505–525

[17] Levenshtein, V.: Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady **10** (1966) 707

[18] Jones, K.S.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation **28** (1972) 11–21

[19] Sipser, M.: Introduction to the Theory of Computation. Second edition. (1997)

[20] Weiß, C., Premraj, R., Zimmermann, T., Zeller, A.: How long will it take to fix this bug? In: Workshop on Mining Software Repositories. (May 2007)

[21] Kremenek, T., Ashcraft, K., Yang, J., Engler, D.: Correlation exploitation in error ranking. In: Foundations of Software Engineering. (2004) 83–93

[22] Kononenko, I.: Estimating attributes: Analysis and extensions of relief. In Bergadano, F., Raedt, L.D., eds.: European Conference on Machine Learning, Springer (1994) 171–182

[23] Robnik-Sikonja, M., Kononenko, I.: An adaptation of relief for attribute estimation in regression. In Fisher, D.H., ed.: Fourteenth International Conference on Machine Learning, Morgan Kaufmann (1997) 296–304

[24] Randolph, J.J.: Free-marginal multirater kappa (multirater κ free): an alternative to Fleiss' fixed-marginal multirater kappa. In: Joensuu Learning and Instruction Symposium. (2005)

[25] Thung, F., Lo, D., Jiang, L.: Automatic defect categorization. In: Working Conference on Reverse Engineering. (2012) 205–214

[26] Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S.: Duplicate bug reports considered harmful...really? In: International Conference on Software Maintenance. (2008) 337 –345

[27] Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: International Conference on Software Engineering. (2009) 485 –495

[28] Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: International Conference on Software Engineering. (2006) 361–370