An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes

Ripon K. Saha* Lingming Zhang† Sarfraz Khurshid* Dewayne E. Perry*
*Electrical and Computer Engineering, The University of Texas at Austin, USA 78712
Email: ripon@utexas.edu, khurshid@ece.utexas.edu, perry@ece.utexas.edu
† Department of Computer Science, The University of Texas at Dallas, USA 75080
Email: lingming.zhang@utdallas.edu

***Abstract*—Regression testing is widely used in practice for validating program changes. However, running large regression suites can be costly. Researchers have developed several techniques for *prioritizing* tests such that the higher-priority tests have a higher likelihood of finding bugs. A vast majority of these techniques are based on *dynamic* analysis, which can be precise but can also have significant overhead (e.g., for program instrumentation and test-coverage collection). We introduce a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard Information Retrieval problem such that the differences between two program versions form the *query* and the tests constitute the *document collection*. REPiR does not require any dynamic profiling or static program analysis. As an enabling technology we leverage the open-source IR toolkit Indri. An empirical evaluation using eight open-source Java projects shows that REPiR is computationally efficient and performs better than existing (dynamic or static) techniques for the majority of subject systems.**

***Index Terms*—Regression Testing, Test Prioritization, Information Retrieval**

I. Introduction

Programs commonly evolve due to feature enhancements, program improvements, or bug fixes. Regression testing is a widely used methodology for validating program changes. However, regression testing can be time consuming and expensive [5], [29]. Executing a single regression suite can take weeks [46]. Regression testing is even more challenging in continuous or short-term delivery processes, which are now common practices in industry [21]. Therefore, early detection of regression faults is highly desirable.

Regression test prioritization (RTP) is a widely studied technique that ranks the tests based on their likelihood in revealing faults and defines a test execution order based on this ranking so that tests that are more likely to find (new, unknown) faults are run earlier [15], [39], [58], [61], [66]. Existing RTP techniques are largely based on dynamic code coverage where the coverage from the previous program version is used to order, i.e., *rank*, the tests for running against the next version [15], [26], [58], [61]. A few recent techniques utilize static program analysis in lieu of dynamic code coverage [39], [66]. RTP techniques (whether dynamic or static) are broadly classified into two categories, *total* or *additional*, depending on how they calculate the rank [46]. *Total* techniques do not change values of test cases during the prioritization process, whereas *additional* techniques adjust values of the remaining test cases taking into account the influence of already prioritized test cases.

Although a number of RTP techniques (specifically coverage-based ones) have been widely used, they have two key limitations [39]. First, coverage profiling overhead (in terms of time and space) can be significant. Second, in the context of certain program changes (which modify behavior significantly) the coverage information from the previous version can be imprecise to guide test prioritization for the current version. Although the static techniques [39], [66] address the coverage profiling overhead, they simulate the coverage information via static analysis, and thus can be also imprecise.

This paper presents REPiR, an information retrieval (IR) approach for regression test prioritization based on program changes. Traditional IR techniques [35] focus on the analysis of natural language in an effort to find the most relevant *documents* in a collection based on a given *query*. Even though the original focus of IR techniques was on documents written in natural language, recent years have seen a growing number of applications of IR to effectively solve software engineering problems by extracting useful information from source code and other software artifacts [9], [31], [33], [38], [42]. The effectiveness of these solutions relies on the use of meaningful terms (e.g., identifiers and comments) in software artifacts, and such use is common in most real world software projects. In the context of testing and debugging, the application of IR has been primarily focused on bug localization [42], [48], [67].

Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships by reducing the RTP problem to a standard IR problem such that program changes constitute the query and the test cases form the document collection. Our tool REPiR embodies our insight. We build REPiR on top of the state-of-the-art Indri [53] toolkit, which provides an open-source, highly optimized platform for building solutions based on IR principles.

We compare REPiR against ten traditional RTP strategies [14], [16], [39] using a dataset consisting of eight open-source software projects. The experimental results show that for the majority of subjects REPiR outperforms all program-analysis-based and coverage-based strategies at both test-method and test-class levels. Thus, REPiR provides an ef-

fective alternative approach to addressing the RTP problem without requiring any dynamic coverage or static analysis information. Furthermore, unlike traditional techniques, REPiR can be made oblivious to the program languages at the expense of only 2% of accuracy, and may be directly applied to various programs written in different languages. For reproducibility and verification, our experimental data is available online.[1] This paper makes the following contributions:

- **REPiR**. We introduce a new approach for regression test prioritization (RTP) based on program changes. We define a reduction from the regression test prioritization problem to a standard information retrieval problem and present our approach, REPiR, based on this reduction.
- **Tool**. We embody our approach in a prototype tool that leverages the off-the-shelf, state-of-the-art Indri toolkit for information retrieval.
- **Evaluation**. We present a rigorous empirical evaluation using version history of eight open-source Java projects and compare REPiR with 10 RTP strategies. We also present different variants of REPiR and provide detailed results how REPiR can be used more effectively depending on test or program differencing granularities.

II. Background

This section briefly discusses the basic concepts of regression test prioritization, working procedure of an IR system, and its applications in software engineering.

A. Regression Test Prioritization

A test prioritization technique or tool reorders the actual execution sequences of test cases in such a way that it meets certain objectives. The nature of objectives could be diverse, including (but not limited to) increasing the fault-detection or code-coverage rate. Rothermel et al. [46] formally defined the test case prioritization problem as finding $T' \in PT$, such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$. In the definition, PT denotes the set of all possible permutations of a given test suite T, and f denotes a function from any member of PT to a real number such that a larger number indicates better prioritization. In this paper, we focus on increasing the likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.

B. IR Techniques in Software Engineering

Recent years have seen many IR applications applied to solve software engineering problems. IR is concerned with search: given a massive collection of documents, IR seeks to find the most relevant documents based on a user-query. Generally an IR system works in three major steps: text preprocessing, indexing, and retrieval. Preprocessing involves text normalization, stop-word removal, and stemming. A text normalizer removes punctuation, performs case-folding, tokenizes terms, etc. In the stop-word removal phase, an IR system filters out the frequently used terms such as prepositions, articles, etc. in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same term (e.g., go, going, gone) to improve term matching between query and document. Then the documents are indexed for fast retrieval. Once indexed, queries are submitted to the search engine, which returns a ranked-list of documents in response. Finally, the search engine (or "model") is evaluated by measuring the quality of its output ranked-list relative to each user's input query. For a broad overview of IR, please refer to [35].

IR techniques have been applied to over two dozen different software engineering problems, many of which are highlighted in two surveys on the application of IR to SE problems [7], [8]. There are two predominant tasks today. The first task is feature (or concept) location, which consists of locating features described in maintenance requests, such as enhancements or faults [9], [20], [31], [33], [38], [42]. The second task is traceability, which links or recovers links between software engineering artifacts [32], [37]. Another closely related task is software reuse, where IR is used to identify the reusable software artifacts [18]. There are also a diverse set of other tasks such as quality assessment [27], change impact analysis in source code [10], restructuring and refactoring [4], defect prediction [6], clone detection [36] and duplicate bug detection [54].

III. REPiR Approach

Regression test prioritization (RTP) and Information Retrieval (IR) both deal with a *ranking* problem, albeit in different domains (Section II). While RTP is concerned with test cases written in a programming language, IR is concerned with documents written in natural language. However, many human-centric software engineering documents are text-based, including source-code, test scripts, and test documents. Furthermore, in real world software projects, developers often use meaningful identifier names and write comments, which allow solving a number of software engineering problems using information retrieval. Our key insight is that in addition to writing good identifier names and comments in the code, developers use very similar terms for test cases, and we can utilize these textual relationships using IR to develop effective and efficient RTP techniques.

A. Problem Formulation

We reduce RTP to a standard IR problem where the program difference between two software revisions or versions is the *query* and the test cases or test-classes are the *document collection*. Therefore, for a given test suite TS, the prioritized test suite TS' is defined by ranking tests in TS based on the similarity score between the program differences and test cases. Reducing the RTP technique to a standard IR task enables us to exploit a wealth of prior theoretical and empirical IR methodology, providing a robust foundation for tackling RTP. This work primarily focuses on projects with JUnit tests.

B. Construction of Document Collection

The process of constructing documents from test cases varies depending on the information granularity and the choice

[1] https://utexas.box.com/repir

of information retrieval techniques. Generally a test suite is a collection of source code files, where each source code file consists of one or more test-methods/functions. For example, JUnit has two levels of test cases: test-classes and test-methods. Prior research [61], [66] on RTP focused on prioritizing both test-methods and test-classes. In this section, we describe the construction of three types of document collections. Hereafter, we use test-class to denote test-class/file and test-method to denote test-method/function.

1) At Test-Class Level: To make a document collection at test-class level, we first build the abstract syntax tree (AST) of each source code file using Eclipse Java Development Tools (JDT), and traverse the AST to extract all the identifier names (class names, attribute names, method names, and variable names) and comments. The identifier names and comments are particularly important from the information retrieval point of view, since these are the places where developers can use their own natural language terms. Alternatively, a document collection at the test-class level can be also constructed without any knowledge of underlying programming language. In this case, we do not construct any AST for test-classes. Rather, we read each term in the test class, remove all mathematical operators using simple text processing, and tokenize them to construct the document-collection.

2) At Test-Method Level: For constructing a document collection at the test-method level, we extract all the methods from the AST using JDT and store all the identifiers and comments related to a given method as a text document.

3) Structured Document: For structured information retrieval, it is important to store documents in such a way that they retain the program structure. In our study, we distinguish four kinds of terms based on program constructs: i) class names, ii) method names, iii) all other identifier names such as attribute and variable names, API names, and iv) comments. To construct structured documents, we traverse the AST for either each test-class or test-method to extract aforementioned terms and store them in an XML document.

C. Query Construction

As we defined in the problem formulation, in an IR-based RTP, differences between two program versions comprise the query. How to utilize the best query representation (e.g., succinct vs. descriptive) is a very well-known problem in traditional IR [28]. While the succinct representation often provides the most important keywords, it may lack other terms useful for matching. In contrast, although the more verbose descriptions may contain many other useful terms to match, it may also contain a variety of distracting terms. In our work, we experiment with three representations of program differences that can affect the overall results of RTP.

1) Low-Level Differences: By low level differences, we mean the program differences between two versions at the line level. We compute the low level differences by applying UNIX *diff* recursively while ignoring spaces and blank lines. It can also be directly obtained from version-control systems (e.g. cvs, svn, git) without additional computation. We denote this representation of query as `LDiff` and quantify it in terms of number of lines.

TABLE I
HIGH LEVEL CHANGE TYPES

No.	Type	Change Description
1	CM	Change in Method
2	AM	Addition of Method
3	DM	Deletion of Method
4	AF	Addition of Field
5	DF	Deletion of Field
6	CFI	Change in Instance of Field Intializer
7	CSFI	Change in Static Field Initializer
8	LC_m	Look-up Change due to Method Changes
9	LC_f	Look-up Change due to Field Changes

2) High-Level Differences: Since LDiff is expected to be noisy (e.g., sensitive to changes in formatting, or local changes), our goal is to summarize LDiff by abstracting local changes and ignoring formatting differences. To this end, we consider nine types of atomic changes (Table I) that have been used in various studies for change impact analysis [19], [62], [64], [65]. We use FaultTracer [63], a change impact analysis tool, to extract these changes. We denote this high-level query as `HDiff` and quantify it in terms of the number of atomic changes.

3) Compact LDiff or HDiff: Since the difference between two program versions is often too long (e.g. thousands of lines), it is highly likely that they would have many duplicated terms. In this representation, we construct a compact version of LDiff and HDiff by removing all the duplicated words from them. We denote these compact forms of LDiff and HDiff as `LDiff.Distinct` and `HDiff.Distinct` respectively.

D. Tokenization

Since we are dealing with program source code rather than natural language written in English, similar to other IR systems for software engineering, our tokenization is different than that of standard IR task. Generally identifier names are a concatenation of words. Dit et al. [8] compared simple camel case splitting to the more sophisticated Samurai [10] system and found that both performed comparably in concept location. Therefore, in addition to splitting terms based on period, comma, white space, we also split identifier names based on the camel case heuristic.

E. Retrieval Model

Researchers in software engineering have experimented with a number of different retrieval models developed by IR including latent semantic indexing (LSI) [11], the vector space model (VSM) [35], and Latent Dirichlet Allocation (LDA) [57]. However, recent research shows that TF.IDF term weighted VSM (briefly TF.IDF model) works better than others [48], [67]. Another study shows that although there is a widespread debate on which of three (TF.IDF [49], BM25 (Okapi) [43], or language modeling [41]) traditionally-dominant IR paradigms was best, all three approaches utilize the same underlying textual features, and empirically perform comparably when well-tuned [17]. Therefore, we chose the

TF.IDF model for our study. We elaborate on this TF.IDF formulation below.

Let us assume that test cases (documents) and a program difference (query) are represented by a weighted term frequency vector $\vec{d}$ and $\vec{q}$ respectively of length n (the size of the vocabulary, i.e., the total number of terms).

$$\vec{d} = (x_1, x_2, ..., x_n) \quad (1)$$

$$\vec{q} = (y_1, y_2, ..., y_n) \quad (2)$$

Each element x_i in $\vec{d}$ represents the frequency of term t_i in document d (similarly, y_i in query $\vec{q}$). However, the terms that occur very frequently in most of the documents are less useful for search. Therefore, in a vector space model, generally query and document terms are weighted by a heuristic TF.IDF weighting formula instead of only their raw frequencies. Inverse document frequency (IDF) diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. Thus, weighted vectors for $\vec{d}$ and $\vec{q}$ are:

$$\vec{d_w} = (tf_d(x_1)idf(t_1), tf_d(x_2)idf(t_2), ..., tf_d(x_n)idf(t_n)) \quad (3)$$

$$\vec{q_w} = (tf_q(y_1)idf(t_1), tf_q(y_2)idf(t_2), ..., tf_q(y_n)idf(t_n)) \quad (4)$$

The basic formulation of IDF for term t_i is $idf(t_i) = log\frac{N}{n_{t_i}}$, where N is the total number of documents in C and n_{t_i} is the number of documents with term t_i. Therefore, in the simplest TF.IDF model, we would simply multiply this value by the term's frequency in document d to compute the TF.IDF score for (t, d). However, actual TF.IDF models used in practice differ greatly from this to improve accuracy [43], [51]. To date IR researchers have proposed a number of variants of TF.IDF model. We adopt Indri's built-in TF.IDF formulation, based upon the well-established BM25 model [43], [60]. This TF.IDF variant has been actively used in IR community over a decade and rigorously evaluated in shared task evaluations at the Text REtrieval Conference (TREC). In this variant, the *document*'s tf function is computed by Okapi:

$$tf_d(x) = \frac{k_1 x}{x + k_1(1 - b + b\frac{l_d}{l_C})} \quad (5)$$

where k_1 is a tuning parameter (≥ 0) that calibrates document term frequency scaling. The *term frequency* value quickly saturates (i.e., dampens or diminishes the effect of frequency) for a small value of k_1, whereas, a large value corresponds to using raw term frequency. b is another tuning parameter between 0 and 1, which is the document scaling factor. When the value of b is 1, the term weight is fully scaled by the document length. For a zero value of b, no length normalization is applied. l_d and l_C represents the document length and average document length for the collection respectively. The IDF value is smoothed as $log\frac{N+1}{n_t+0.5}$ to avoid division by zero for the special case when a particular term appears in all documents.

The term frequency function of query, tf_q is defined similarly as tf_d. However, since the query is fixed across documents, normalization of query length is unnecessary. Therefore, b is simply set to zero.

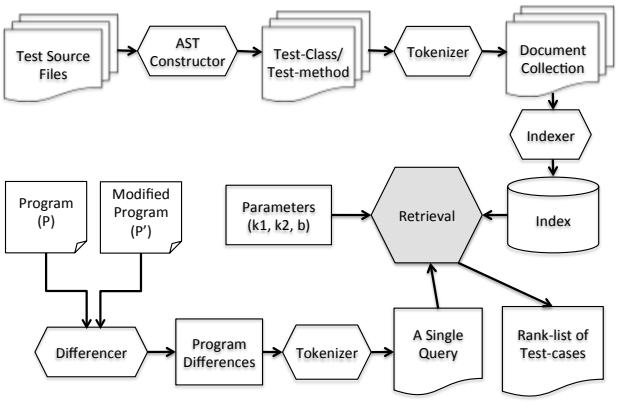

Fig. 1. REPiR architecture

$$tf_q(y) = \frac{k_2 y}{y + k_2} \quad (6)$$

Now the similarity score of document $\vec{d}$ against query $\vec{q}$ is given by Equation 7.

$$s(\vec{d}, \vec{q}) = \sum_{i=1}^{n} tf_d(x_i) tf_q(y_i) idf(t_i)^2 \quad (7)$$

F. Structured Information Retrieval

The TF.IDF model presented in Equation 7 does not consider source code structure (program constructs)–i.e., each term in a source code file is considered having the same relevance with respect to the given query. In our recent work on IR-based bug localization [48], we found that incorporating structural information into IR model, known as structured retrieval, can help improve the results considerably. In this paper, we adapt the structured IR for RTP.

As we described in Section III-B3, we distinguish four kinds of program constructs: *classes, methods, variables, and comments*. Therefore, to exploit all of this structural information, we perform a separate search for each type of terms. Since TF.IDF model assumes that there is no statistical dependence between terms, we simply sum all the field retrieval scores to calculate the final similarity score.

$$s'(\vec{d}, \vec{q}) = \sum_{f \in D} s(d_f, q) \quad (8)$$

where f is a particular document field. The benefit of this model is that terms appearing in multiple document fields are implicitly assigned greater weight, since the contribution from each term is summed over all fields in which it appears.

G. Architecture

Figure 1 shows the overall architecture of our IR-based RTP prototype, REPiR (REgression test Prioritization using information Retrieval). First, REPiR takes the source code files of tests as input that we would like to prioritize. Next, it extracts information from test cases, tokenizes the terms, and constructs a document collection for a given level of granularity as described in Section III-B. REPiR also extracts program changes (LDiff, HDiff, or compact) and tokenizes terms in the same way as tokenizing document terms to construct the query.

We adopt the Indri toolkit [53], a highly efficient open-source library, for indexing and developing our retrieval model. After documents and queries are created above, they are handed off to Indri for stopword removal, stemming, and indexing. Note that we use the default stopword list provided with Indri and the Krovetz stemmer for stemming. We also set the values of k_1, k_2, and b to 1.0, 1000, and 0.3 respectively, which have been found to perform well empirically [48].

IV. Empirical Evaluation

To investigate the effectiveness of REPiR, we performed an empirical evaluation in terms of five research questions.

- **RQ1:** Is REPiR more effective than untreated or random test orders?
  This research question aims to understand whether REPiR reveals regression faults earlier than when there is no RTP or when test cases are ordered at random.
- **RQ2:** Do the high-level program differences help improve the performance of REPiR?
  Low-level program differences are expected to produce noisy results since changes to even a single character of a line would be interpreted as deletion of a line followed by an addition of line. Therefore, in this research question, we investigate whether the high-level program differences based on AST help improve the accuracy of REPiR.
- **RQ3:** Is structured retrieval more effective for RTP?
  In our recent work, we showed that incorporating the structural information program into traditional information retrieval model, which is known as structured information retrieval, can improve the bug localization results considerably [48]. In this research question, we investigate whether the same is true for IR-based RTP.
- **RQ4:** How does REPiR perform compared to the existing RTP techniques?
  To date researchers have focused on various program analysis (either static or dynamic) based techniques to propose or improve RTP. In this research question, we are interested in investigating how well REPiR performs compared to those existing techniques.
- **RQ5:** How does REPiR perform when it is oblivious to language-level information?
  Since REPiR utilizes only textual information, identification of specific programing language constructs is not needed. Lightweight language-specific parsing is used only for identifier-name extraction for constructing document collection at the method-level. However, if we use LDiff to prioritize test-classes, REPiR can be made completely oblivious of the underlying programming language. In this research question, we investigate how REPiR performs for this configuration.

A. Subject Systems

We studied eight open source software systems for our evaluation. These systems are from diverse application domain and have been widely used in software testing research [13], [39], [50]. We obtained Xml-Security and Apache Ant from the well-known Software-artifact Infrastructure Repository (SIR) [12] and extract other subject systems from their host website. The sizes of these systems vary from 5.7K LOC (Time and Money, 2.7K LOC source code and 3K LOC test code) to 96.9K LOC (Apache Ant, 80.4K LOC source code and 16.5K LOC test code).

For each subject system, we first extract all the major releases with their test cases and consider each consecutive versions as a version-pair. For each pair, we run the old regression test suite on new version to find possible regression test failures. Then, we treat the changes causing those test failures as the regression faults. In this way, we were able to identify 24 version-pairs with regression faults, which are all used in our study. Table II provides all the details regarding each version-pair, including the statistics for test methods, test classes, fault-revealing test methods, program edits (in terms of both LDiff and HDiff), and failure-inducing edits (in terms of HDiff). It also represents the textual properties of the first version in each version-pair, such as the number of distinct tokens extracted from source code and test cases, as well as the number and proportion of test-case tokens that also appear in source code. The high proportion (i.e., $>$48.8% for all subjects) of test-case tokens in source code confirms REPiR's motivation that developers use similar terms for tests and source code.

B. Independent Variable

Since we are interested in investigating the performance of REPiR for different granularities of test cases, different representations of program differences, and how it works compared to other RTP strategies, we have mainly three independent variables:

- **IV1**: Test-case Granularity
- **IV2**: Program Differences, and
- **IV3**: Prioritization Strategy

In section III, we discussed different test cases granularities and program differences. Now we briefly describe 10 test prioritization strategies that we considered for comparison.

Untreated test prioritization keeps the original sequence of test cases as provided by developers. In our discussion, we denote the untreated test case prioritization as `UT`. We consider this to be the *control* treatment.

Random test prioritization rearranges test cases randomly. Since the results of random strategy may vary a lot for each run, we applied random test prioritization 1000 times for each subject according to Arcuri et al.'s guidelines to evaluate randomized algorithms [2]. In our discussion, we denote the random test prioritization technique as `RT`.

Dynamic coverage-based test prioritization varies depending on the types of coverage information (e.g., the method or statement coverage) and prioritization strategies (e.g., the *total* or *additional* strategy). We used the four most-widely used variants of coverage-based RTP: `CMA`, `CMT`, `CSA`, and `CST`. For example, `CMA` denotes test prioritization based on the *Method* coverage using the *Additional* strategy, and `CST` denotes test prioritization based on the *Statement* coverage using the *Total* strategy.

TABLE II
DESCRIPTION OF THE DATASET

No.	Project	Version Pair	#TMeth	#TClass	#FTMeth	#LDiff	#HDiff	#FEdits	#SrcToken	#TstToken	# ComToken
P_1	Time and Money	3.0-4.0	143	15	1	1,200	215	1	276	303	183 (60.4%)
P_2	Time and Money	4.0-5.0	159	16	1	658	246	1	303	318	207 (65.1%)
P_3	Mime4J	0.50-0.60	120	24	8	4,377	2,862	3	494	293	194 (66.2%)
P_4	Mime4J	0.61-0.68	348	57	3	2,967	3,160	4	535	472	301 (63.8%)
P_5	Jaxen	1.0b7-1.0b9	24	12	2	2,788	204	3	368	117	93 (79.5%)
P_6	Jaxen	1.1b6-1.1b7	243	41	2	5,020	92	1	458	295	207 (70.2%)
P_7	Jaxen	1.0b9-1.0b11	645	69	1	1020	92	1	476	361	235 (65.1%)
P_8	Xml-Security	1.0-1.1	91	15	5	6,025	329	2	905	396	317 (80.1%)
P_9	XStream	1.20-1.21	637	115	1	833	209	1	698	928	470 (50.6%)
P_{10}	XStream	1.21-1.22	698	124	2	1,079	222	2	712	967	492 (50.9%)
P_{11}	XStream	1.22-1.30	768	133	11	5,920	540	11	740	1,029	512 (49.8%)
P_{12}	XStream	1.30-1.31	885	150	3	2,630	416	3	780	1,122	547 (48.8%)
P_{13}	XStream	1.31-1.40	924	140	7	6,744	1,225	7	796	1,146	559 (48.8%)
P_{14}	XStream	1.41-1.42	1200	157	5	828	136	5	835	1,206	595 (49.3%)
P_{15}	Commons-Lang	3.02-3.03	1698	83	1	1,757	221	1	906	925	603 (65.2%)
P_{16}	Commons-Lang	3.03-3.04	1703	83	2	3,003	172	2	913	924	603 (65.3%)
P_{17}	Joda-Time	0.90-0.95	219	10	2	8,653	5,976	2	450	236	130 (55.1%)
P_{18}	Joda-Time	0.98-0.99	1932	71	2	13,735	1,254	2	573	401	289 (72.1%)
P_{19}	Joda-Time	1.10-1.20	2420	90	1	1,348	793	1	606	501	358 (71.5%)
P_{20}	Joda-Time	1.20-1.30	2516	93	3	1,979	571	3	618	516	369 (71.5%)
P_{21}	Apache Ant	0.0-1.0	112	28	5	7,766	2,071	3	1,169	170	144 (84.7%)
P_{22}	Apache Ant	3.0-4.0	219	52	3	40,102	5,752	3	1,895	327	287 (87.8%)
P_{23}	Apache Ant	4.0-5.0	520	101	2	7,760	586	3	2,419	646	522 (80.8%)
P_{24}	Apache Ant	6.0-7.0	558	105	12	36,749	5,019	7	2,458	671	545 (81.2%)

JUPTA [39], [66] is a static-analysis-based test prioritization approach that ranks tests based on test ability (TA). TA is determined by the number of program elements relevant to a given test case (T), which is computed from the static call graph of T to simulate coverage information. TA can be calculated based on two levels of granularity: fine granularity and coarse granularity. TA at the fine-granularity level is calculated based on the number of statements contained by the methods transitively called by each test, whereas TA at the coarse-granularity level is calculated based on the number of methods transitively called by each test. Similar with coverage-based prioritization techniques, we also used four variants of JUPTA: JMA, JMT, JSA, and JST.

Note that we implemented all the static and dynamic RTP techniques using byte-code analysis. More specifically, we used the *ASM byte-code manipulation framework*[2] to extract all the static and coverage information for test prioritization.

C. Dependent Variable

We use the Average Percentage Faults Detected (APFD) [46], a widely used metric in evaluating regression test prioritization techniques, as the dependent variable. This metric measures prioritization effectiveness in terms of the rate of fault detection of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2 \times n} \quad (9)$$

where n denotes the total number of test cases, m denotes the total number of faults, and TF_i denotes the smallest number of test cases in sequence that need to be run in order to expose the i^{th} fault. The value of APFD can vary from 0 to 1. Since n and m are fixed for any given test suite, a higher APFD value indicates a higher fault-detection rate.

[2]http://asm.ow2.org/

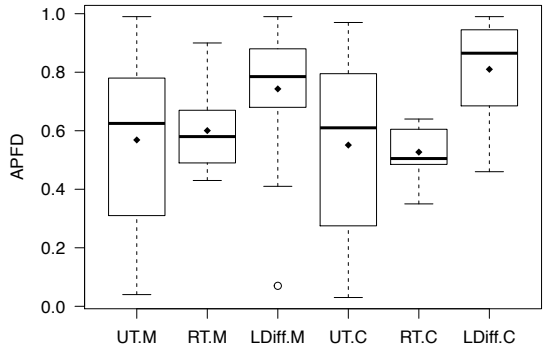

Fig. 2. Accuracy of REPiR (LDiff) at test-method and test-class levels

D. Study Results

In this section, we present the experimental results which answer our research questions.

1) REPiR vs. UT and RT: First, to understand the performance of REPiR compared to UT and RT at the test-method level, REPiR is set to construct the document collection at test-method level and use LDiff as a query. We select the TF.IDF retrieval model and run REPiR on all version pairs (P_1-P_{24}). We also run RT and UT for the same dataset. Each RTP technique provides a ranked-list of test-methods for each version-pair, which we use to calculate APFDs. We also perform the same experiment for test-class level.

Figure 2 presents the results for all version-pairs in the form of boxplot. In each plot, the X-axis shows the strategies that we compared and the Y-axis shows the APFD values. To name RTP techniques, we used M to denote method-level and C to denote class-level test-cases. Each boxplot shows the average (dot in the box), median (line in the box), upper/lower quartile,

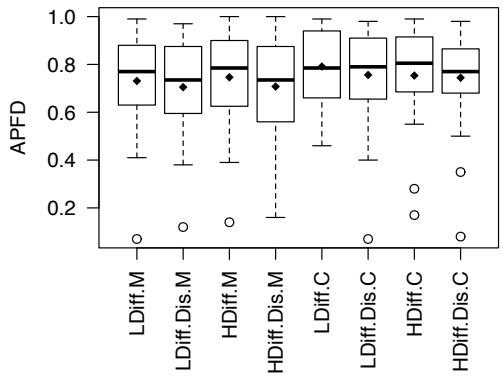

Fig. 3. Impact of program differences at test-method (M) and test-class (C) levels (Dis=Distinct)

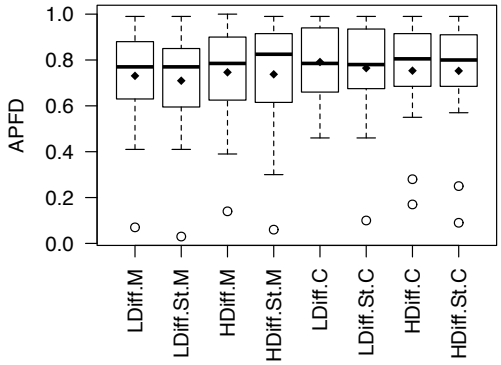

Fig. 4. Impact of structured retrieval at test-method (M) and test-class (C) levels (St = Structured)

and 90th/10th percentile APFD values achieved by a strategy. From the figure, we see that the mean, median, first and third quartiles APFD of (UT, RT, REPiR) at test-method level are (0.52, 0.6, **0.73**), (0.49, 0.55, **0.77**), (0.25, 0.5, **0.64**), and (0.72, 0.67, **0.87**) respectively, which clearly indicates that REPiR overall performs better than UT and RT.

We also investigate whether the accuracy of REPiR varies with the length of program differences or the number of test-methods, since these are two main inputs for REPiR. We compute the Spearman correlation between the size of LDiff (quantified by number of changed lines) and APFD, and between the number of test-methods and APFD. The low correlation values for both cases (0.23 and 0.2) indicate that the accuracy of REPiR is fairly independent of the length of program differences and the number of test-methods.

Similarly for the test-class level, we see that the mean, median, first and third quartiles APFD of REPiR (**0.79**, **0.79**, **0.66**, **0.94**) is higher than that of UT (0.52, 0.55, 0.3, 0.66) and RT (0.52, 0.5, 0.49, 0.53). These results show that REPiR performs much better than UT and RT at test-class level. The low Spearman correlations between the number of test-classes and APFD (0.24) and between the length of program differences and APFD (-0.49) indicate that the accuracy of REPiR is not dependent either on the the number of test-classes or the size of program differences.

2) Impact of Program Differences: To answer RQ2, we run REPiR with four forms of program differences (LDiff, LDiff.Distinct, HDiff, HDiff.Distinct) separately for both at test-method level and at test-class level. Figures 3 presents the summary of APFD values for test-methods and test-classes respectively. Results show that at method-level, HDiff works better than LDiff in terms of both mean (HDiff: 0.75 vs. LDiff: 0.73) and median (HDiff: 0.79 vs. LDiff: 0.77). When we take a closer look at our data for individual program versions, we find that HDiff improves the APFD values for 14 version-pairs, while decreases it for 7 version-pairs. However, if we further condense the query by removing duplicate terms, the accuracy decreases for both HDiff and LDiff, and the decrease rate is larger for HDiff than that of LDiff. We believe that since HDiff is already condensed, it is affected more due to the removal of duplicated terms than LDiff.

On the other hand, we see that LDiff works slightly better than HDiff, on average, at test-class level. The mean value of APFDs for HDiff are 0.75, whereas it is 0.79 for LDiff, although interestingly median APFD for HDiff is 0.01 higher than that of LDiff. We think the reason for getting better accuracies with HDiff at test-method level is that since test-methods are typically small, they are more affected by the noises of LDiff. From the results of LDiff.Distinct and HDiff.Distinct, we see that like test-method level, the removal of duplicated terms also slightly hurts the results. However, it should be noted that REPiR is computationally more efficient when it uses the compact representation of the query and thus may be suitable when the program change is large.

3) Impact of Structured Retrieval: To understand whether the structured retrieval leads to better prioritization, we constructed the structured version of document collection at both test-method and test-class level as described in Section III-B3. Then we used the structured retrieval techniques as described in Section III-F. From Figure 4, we see that structured retrieval works basically the same as unstructured retrieval at both test-method and test-class levels for HDiff. However, it reduces the accuracy slightly for LDiff. Therefore, our results suggest that although structured retrieval has been found to improve the accuracy of bug localization results considerably [48], it does not help much for RTP.

4) REPiR Vs. JUPTA or Coverage-based RTP: To answer RQ4, first we ran all the eight techniques (four variants of JUPTA based on call graphs and four variants of coverage-based technique) described in Section IV-B on each program-pair in our dataset. Figures 5 and 6 show the summary of APFD values for each strategy at method-level and class-level respectively. Results show that for both static and dynamic techniques, *additional* strategies are overall more effective than *total* strategies. This is consistent with prior studies [24], [61].

Now we compare all the mean APFDs of REPiR (from Figure 3) with that of JUPTA and coverage-based techniques (from Figures 5 and 6) as summarized in Table III. From the results, we see that REPiR equipped with either LDiff or HDiff overall outperforms all the JUPTA and coverage-based approaches (*total* or *additional*) regardless of test-case granularities (test-method/test-class). At test-method level the mean APFD achieved by REPiR are 0.73 using LDiff and

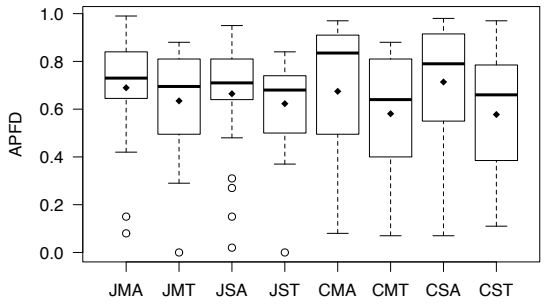

Fig. 5. Accuracy of JUPTA and coverage-based RTP at test-method level

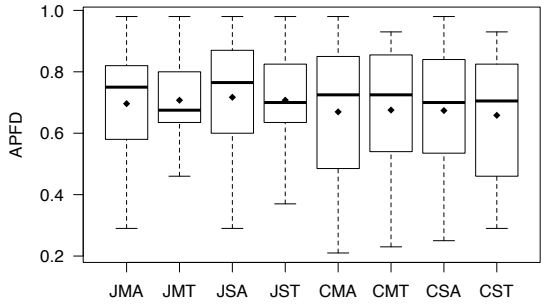

Fig. 6. Accuracy of JUPTA and coverage-based RTP at test-class level

TABLE III
COMPARISON OF MEAN APFDS ACHIEVED BY DIFFERENT STRATEGIES (TM=TEST-METHOD,TC=TEST-CLASS)

	TM	TC		TM	TC		TM	TC
LDiff	0.73	0.79	JMA	0.69	0.70	CMA	0.67	0.67
LDiff.Dis	0.71	0.76	JMT	0.63	0.71	CMT	0.58	0.68
HDiff	0.75	0.75	JSA	0.66	0.72	CSA	0.71	0.67
HDiff.Dis	0.71	0.74	JST	0.62	0.71	CST	0.58	0.66

0.75 using HDiff, whereas the best variants of JUPTA (JMA) and coverage-based technique (CSA) achieve 0.69 and 0.71, respectively. At test-class level, REPiR achieves the mean APFD of 0.79 using LDiff and 0.75 using HDiff, whereas the best variants of JUPTA (JSA) and coverage-based technique (CMT) achieve 0.72 and 0.68, respectively. Even with the compact representation of queries (LDiff.Dis and HDiff.Dis), REPiR performs better than all *total* strategies, and performs equally well or better than *additional* strategies.

We further investigate how REPiR performs for each subject system. For conciseness, we present the *average* APFD values for each system achieved by REPiR using HDiff, JMA (JUPTA based on method coverage using additional strategy), which is the best among all the JUPTA strategies, and CSA (based on statement coverage using additional strategy), which is the best among all the coverage-based strategies at method level. Similarly, for test-class level, we present the average APFD values for each subject system achieved by REPiR using LDiff, JSA, and CMT, which are the best in IR, JUPTA, and coverage-based approach respectively. Table IV shows that REPiR achieved the best APFD for six/five out of eight subjects at test-method/test-class level.

5) Accuracy when REPiR is oblivious of Programming Language: For this experimental setting, REPiR does not build any AST for source code or test classes while constructing document collections and queries. Documents are made at

TABLE IV
COMPARISON BY SUBJECTS

	Test Method			Test Class		
Subject	**HDiff**	**JMA**	**CSA**	**LDiff**	**JSA**	**CMT**
Time and Money	0.50	0.47	0.19	0.82	0.91	0.41
Mime4J	0.68	0.68	0.59	0.89	0.79	0.65
Jaxen	0.67	0.67	0.94	0.61	0.57	0.82
XML-Security	0.80	0.42	0.69	0.90	0.77	0.37
XStream	0.84	0.68	0.79	0.87	0.83	0.76
Commons-Lang	0.95	0.79	0.86	0.96	0.62	0.83
joda	0.75	0.87	0.78	0.63	0.66	0.72
Apache Ant	0.7	0.54	0.65	0.7	0.51	0.54

test-class level by simply removing mathematical operators and tokenizing any text that are in the test-classes. So the documents are expected to be noisy because of programming language keywords. We used LDiff as query, which is also program language independent. Then we run REPiR for all version-pairs and calculate the APFD values. Results show that the mean APFD across all version-pairs is 0.77, while it was 0.79 when we used only identifiers and comments as document terms. Interestingly, it turns out that the median APFD of the language oblivious approach is 0.01 higher: 0.8 for language-oblivious configuration vs. 0.79 when we used only identifiers and comments of test classes. Therefore, our results show that REPiR, even in its simplest form when it is oblivious of the programming language, does not lose any significant accuracy and outperforms all the JUPTA and coverage-based approach (highest mean is 0.72 achieved by JSA).

E. Qualitative Analysis

Our quantitative results already show that developers tend to use very similar terms in source code and corresponding test cases, which is one of our main motivations for developing an IR-based RTP. In this section, we illustrate a concrete example to show the usefulness of this information.

When Commons-Lang evolved from version 3.02 to 3.03, test-method `FastDateFormatTest.testLang538` failed since the developer incorrectly removed a conditional block for updating the time zone in method `FastDateFormat.format()` (shown in Figure 7, highlighted in red). If we extract the program differences from this change, LDiff produces the following terms: time, zone, forced, calendar, get etc., while HDiff produces `CM:FastDateFormat.format`. It should be noted there were also many other (non-faulty) changes in the query. Now let us take a look at the test-method that reveals this fault in Figure 8. Interestingly, we see many of the terms from faulty edits in the test method (highlighted in bold). Furthermore, the source code class (`FastDateFormat`) and

```
public StringBuffer format(Calendar calendar,
        StringBuffer buf) {
-       if (mTimeZoneForced){
-               calendar.getTimeInMillis();
-               calendar = (Calendar) calendar.clone();
-               calendar.setTimeZone(mTimeZone);
-       }
}
```

Fig. 7. A failure-inducing edit in Commons-Lang 3.02

```
public void testLang538() {
    final String dateTime = "2009-10-16T16:42:16.000Z";
    // more commonly constructed with: cal = new GregorianCalendar(2009, 9, 16, 8, 42, 16)
    // for the unit test to work in any time zone, constructing with GMT-8 rather than default locale time zone
    GregorianCalendar cal = new GregorianCalendar(TimeZone.getTimeZone("GMT-8"));
    cal.clear();
    cal.set(2009, 9, 16, 8, 42, 16);
    FastDateFormat format = FastDateFormat.getInstance("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", TimeZone.getTimeZone("GMT"));
    assertEquals("dateTime", dateTime, format.format(cal));}
```

Fig. 8. A fault-revealing test method for Commons-Lang 3.02

the corresponding test-class (`FastDateFormatTest`) have similar names. As a result, REPiR with HDiff ranked this method at 7th and LDiff ranked at 17th position among 1,698 test-methods. On the other hand, the best variants of JUPTA and coverage-based technique, JMA and CSA ranked it at 367th and 370th position respectively.

In spite of such good results, there was one occasion, where REPiR performed unsatisfactorily–when Time&Money evolved from 4.0 to 5.0. We found that the fault revealing test method was `MoneyTest.testPrint` where there was apparently no information of use to IR, since the only line in the test method is `assertEquals("USD 15.00",d15.toString())`. However, our overall results show that the number of such occasions is very small (1 out of 24 cases in our study).

F. Time and Space Overhead

The running time of REPiR depends on three parameters: the size of vocabulary, the number of test cases, and the length of program differences. REPiR works most efficiently when we use a compact representation of the query. It takes only a fraction of a second for each version-pair to prioritize its test cases. For example, REPiR took only 0.18 second to prioritize all the test-methods of Joda-Time 1.20, which has the largest number of test-methods in our study. The preprocessing and indexing took only three seconds. When we used the full representation of query, REPiR took 20 seconds. On the other hand, the *additional* strategy based on statement level coverage information took 40 seconds, only for test prioritization (*excluding instrumentation and coverage collection*). The time complexity of REPiR and total strategy grows *linearly* when test suite size increases, while the additional strategy grows *quadratically* [39]. Thus, REPiR is a more cost-effective approach. Furthermore, note that coverage-based approach is not useful if the coverage is not available from the old version because developers can simply run all the tests instead of spending time for recollecting the coverage.

The space overhead of REPiR is determined by the requirement of indexing test cases for IR. For most of the subjects, index-size is around 1MB. The index-size of Joda-Time 1.20 having the largest number of test-methods is 3MB. On the contrary, the data required by the traditional techniques for the same system was 11.6MB for method coverage matrix and 31MB for statement coverage matrix. The time and space overhead of JUPTA is very similar to coverage-based approaches since JUPTA tries to simulate code coverage. All the experiments were performed on a MacBook Pro running OS X 10.8 with Intel Core i7 CPU (2.8GHz) and 4GB RAM.

G. Threats to Validity

This section discusses the validity and generalizability of our findings.

Construct Validity: We used two artifacts of a software repository: program source code and test cases, which are generally well understood. Our evaluation uses subject systems with both real and seeded (for the projects from SIR) regression faults. Also we applied all the prioritization techniques on the same dataset, enabling fair comparison and reproducible findings. To evaluate the quality of prioritization, we chose APFD, which has been extensively used in the field of regression test prioritization, and is straightforward to compute. APFD expresses the quality of prioritized test cases based on how early the faulty test cases are positioned in the prioritized suite. However, APFD does not consider either the execution time of individual test cases or the severity of faults. Therefore, it may not accurately estimate how much we are gaining from the prioritized test suite in terms of cost and benefits.

Internal Validity: The success of REPiR vastly depends on the usage of meaningful and similar terms in source code and corresponding test cases, which is consistent with programming best practices. Another threat to internal validity is the potential faults in our implementations as well as the used libraries and frameworks. To reduce it, we used mature libraries and frameworks that have been widely used in various software engineering and information retrieval applications (e.g., FaultTracer [63] and Indri [53]).

External Validity: Our experimental results are based on 24 versions of programs from eight software projects, all of them are open source projects written in Java with JUnit tests. Although they are popular projects and widely used in regression testing research, our findings may not be generalizable to other open-source or industrial projects with other test paradigms. Note however that our technique does not have to rely on language specific features and therefore we expect it to handle programs in other languages. For example, in another study [47], we showed that the information retrieval based bug localization is equally effective in C programs. The risk of insufficient generalization could be mitigated by applying REPiR on more subject systems (both open source and industrial). This will be explored in our future work.

V. Related Work

Reducing the time and cost of regression testing has been an active research area for near two decades. Researchers have already proposed various regression testing techniques, such as regression test selection [3], [45], prioritization [15], [39], and reduction [44]. Since our work is for regression test prioritization (RTP), this section is limited to the relevant

work in this area. For related work regarding IR in software engineering, please refer to Section II-B.

Wong et al. [58] introduced the notion of RTP to make regression testing more effective. They made use of program differences and test execution coverage from the previous version, and then sorted test cases in order of increasing cost per additional coverage. Rothermel et al. [46] empirically evaluated a number of test prioritization techniques, including both the *total* and *additional* test prioritization strategies using various coverage information. In that work, they also proposed the widely used APFD metric for test prioritization. Along the same line, Elbaum et al. investigated more code coverage information [16], and incorporated the cost and the severity of each test case for test prioritization [15]. Jones and Harrold [25] argued that there are important differences between statement-level coverage and modified condition/decision coverage (MC/DC) for regression testing, and proposed test reduction and prioritization using the MC/DC information. Jeffrey and Gupta [22] introduced the notion of relevant slices in RTP. Their approach assigns higher weight to a test case that has larger number of statements (branches) in its relevant slice of the output. However, a common limitation of these techniques is that they require coverage information for the old version, which can be costly to collect or may not be available in the repository.

Besides investigating different types of coverage information, researchers have also proposed various other strategies for RTP. Li et al. [30] used search-based algorithms, such as hill-climbing and genetic programming, for test prioritization. Jiang et al. [24] used the idea of adaptive random testing for test prioritization. Zhang et al. [61] recently proposed a spectrum of test prioritization strategies between the traditional *total* and *additional* strategies based on statistical models. However, according to the reported empirical results [24], [61], the traditional *additional* strategy remains one of the most effective test prioritization strategies.

There are also some approaches that do not require dynamic coverage information. Srikanth et al. [52] proposed a value-driven approach to system-level test case prioritization based on four factors: requirements volatility, customer priority, implementation complexity, and fault proneness of the requirements. Tonella et al. [56] used relative priority information from the user, in the form of pairwise test case comparisons, to iteratively refine the test case ordering. Yoo et al. [59] further used test clustering to reduce the manual efforts in pairwise test comparisons. Ma and Zhao [34] distinguished fault severity based on both users knowledge and program structure information, and prioritized tests to detect severe faults first. All these approaches require inputs from someone who is familiar with the program under test, which may be costly and not always available. To avoid manual efforts, Zhang et al. [66] proposed a static test prioritization approach, JUPTA, which extracts static call graph of a given test case to estimate its coverage. Later Mei et al. [39] extended the study and proposed more variants of JUPTA along the same way. Recently, Jiang and Chan [23] proposed a static test prioritization approach based on static test input information. However, all these approaches try to use static information to simulate code coverage, and thus may be imprecise. In contrast, REPiR is a fully automated and lightweight (does not require coverage collection or static analysis) test prioritization approach based on information retrieval, and has been shown to be more precise and efficient than many existing techniques.

Like our approach, there are also some test prioritization techniques that utilize the presence of natural english in source code artifacts. Arafeen and Do [1] first clustered test cases based on requirements, which are written in English and then prioritized each cluster based on code metrics. Nguyen et al. [40] used an IR-based approach for prioritizing audit tests in evolving web services where they use service change descriptions to query against test execution traces. Both of these approaches require requirement-documents or manual change descriptions with past test execution traces, which may not be available. Furthermore, they are designed and intended to be used when requirements or services change and may not be well suited for day-to-day regression testing. Thomas [55] proposed a test prioritization approach using a topic model approach where test cases are ordered based on their edit-distances. This technique does not utilize any change information and thus may be imprecise.

VI. Conclusion

To reduce the regression testing cost, researchers have developed various techniques for *prioritizing* tests such that the higher priority tests have a higher likelihood of finding bugs. However, existing techniques require either dynamic coverage information or static program analysis, and thus can be costly or imprecise. In this paper, we introduced a new approach, REPiR, to address the problem of regression test prioritization by reducing it to a standard IR problem. REPiR does not require any dynamic profiling or static program analysis. We rigorously evaluated REPiR using a dataset consisting of 24 version-pairs from eight projects with both real and seeded regression faults, and compared it with 10 RTP strategies. The results show that REPiR is more efficient and outperforms the existing strategies for the majority of the studied subjects. We also show that REPiR can be made oblivious of the underlying programming language for test-class prioritization, seldom losing accuracy. We believe that this alternative approach to RTP represents a promising and largely unexplored new territory for investigation, providing an opportunity to gain new traction on this old and entrenched problem of RTP. Moreover, further gains might be achieved by investigating such IR techniques in conjunction with traditional static and dynamic program analysis, integrating the two disparate approaches, each exploiting complementary and independent forms of evidence regarding RTP.

Acknowledgement: We thank Julia Lawall and Matthew Lease for their input on our draft. This work was funded in part by the National Science Foundation (NSF Grant Nos. CCF-1117902 and CCF-0845628).

REFERENCES

[1] M. J. Arafeen and H. Do. Test case prioritization using requirements-based clustering. In *ICST*, pages 312–321, 2013.

[2] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[3] T. Ball. On the limit of control flow analysis for regression test selection. In *ISSTA*, pages 134–142, 1998.

[4] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A two-step technique for extract class refactoring. In *ASE*, pages 151–154, 2010.

[5] B. Beizer. *Software Testing Techniques (2nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Increasing diversity: Natural language measures for software fault prediction. *JSS*, 82(11):1793–1803, 2009.

[7] D. Binkley and D. Lawrie. Applications of information retrieval to software development. *ESE (P. Laplante, ed.)*, 2010.

[8] D. Binkley and D. Lawrie. Applications of information retrieval to software maintenance and evolution. *ESE (P. Laplante, ed.)*, 2010.

[9] D. Binkley, D. Lawrie, and C. Uehlinger. Vocabulary normalization improves ir-based concept location. In *ICSM*, pages 588–591, 2012.

[10] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Metrics*. IEEE Computer Society, 2005.

[11] S. Deerwester, S. T. Dumais, G. W. Furn, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[12] H. Do and G. Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM*, pages 411–420, 2005.

[13] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *ISSRE*, pages 113–124, 2004.

[14] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[15] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE*, pages 329–338, 2001.

[16] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *TSE*, 28(2):159–182, 2002.

[17] H. Fang, T. Tao, and C. Zhai. A formal study of information retrieval heuristics. In *SIGIR*, pages 49–56, 2004.

[18] W. Frakes. A case study of a reusable component collection in the information retrieval domain. *JSS*, 72(2), 2004.

[19] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ASE*, pages 361–372, 2014.

[20] E. Hill, S. Rao, and A. Kak. On the use of stemming for concern location and bug localization in java. In *SCAM*, pages 184–193, 2012.

[21] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.

[22] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *COMPSAC*, pages 411–420, 2006.

[23] B. Jiang and W. Chan. Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization. In *COMPSAC*, pages 190–199. IEEE Computer Society, 2013.

[24] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *ASE*, pages 233–244. IEEE, 2009.

[25] J. A. Jones and M. J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *ICSM*, pages 92–101, 2001.

[26] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*, pages 119–129, 2002.

[27] D. Lawrie, H. Feild, and D. Binkley. Leveraged quality assessment using information retrieval techniques. In *ICPC*, pages 149–158, 2006.

[28] M. Lease, J. Allan, and W. B. Croft. Regression Rank: Learning to Meet the Opportunity of Descriptive Queries. In *ECIR*, pages 90–101, 2009.

[29] H. K. N. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, 1989.

[30] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *TSE*, 33(4):225–237, 2007.

[31] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *ASE*, pages 234–243, 2007.

[32] A. D. Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *ICSM*, pages 299–309, 2006.

[33] S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent dirichlet allocation. In *WCRE*, pages 155–164, 2010.

[34] Z. Ma and J. Zhao. Test case prioritization based on analysis of program structure. In *APSEC*, pages 471–478, 2008.

[35] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[36] A. Marcus and J. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114, 2001.

[37] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *ICSE*, pages 125–135, 2003.

[38] A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic. Using data fusion and web mining to support feature location in software. In *ICPC*, pages 14–23, 2010.

[39] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel. A static approach to prioritizing junit test cases. *TSE*, 38(6):1258–1275, 2012.

[40] C. D. Nguyen, A. Marchetto, and P. Tonella. Test case prioritization for audit testing of evolving web services using information retrieval techniques. In *ICWS*, pages 636–643, 2011.

[41] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual ACM SIGIR conference*, pages 275–281, 1998.

[42] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *MSR*, pages 43–52, 2011.

[43] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation as a way of life: Okapi at trec. *IPM*, 36(1):95–108, 2000.

[44] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998.

[45] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2):173–210, 1997.

[46] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *ICSM*, pages 179–188, 1999.

[47] R. K. Saha, J. Lawall, S. Khurshid, and D. E. Perry. On the effectiveness of information retrieval based bug localization for c programs. In *ICSME*, pages 161–170. IEEE, 2014.

[48] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, pages 345–355, 2013.

[49] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *IPM*, 24(5):513–523, 1988.

[50] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *FSE*, pages 297–298, 2009.

[51] A. Singhal. Modern information retrieval: A brief overview. *DEB*, 24(4):35–43, 2001.

[52] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *ISESE*, pages 64–73, 2005.

[53] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. In *ICIA*, pages 2–6, 2005.

[54] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang. Towards more accurate retrieval of duplicate bug reports. In *ASE*, pages 253–262, 2011.

[55] S. Thomas, H. Hemmati, A. Hassan, and D. Blostein. Static test case prioritization using topic models. *ESE*, 19(1):182–212, 2014.

[56] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *ICSM*, pages 123–133, 2006.

[57] X. Wei and W. B. Croft. Lda-based document models for ad-hoc retrieval. In *ICRDIRl*, pages 178–185, Seattle, WA, 2006.

[58] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *ISSRE*, pages 230–238, 1997.

[59] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, pages 201–212, 2009.

[60] C. Zhai. Notes on the lemur tfidf model (unpublished work). Technical report, Carnegie Mellon University, 2001.

[61] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pages 192–201, 2013.

[62] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pages 23–32, 2011.

[63] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: A change impact and regression fault analysis tool for evolving java programs. In *FSE*, pages 40:1–40:4, 2012.

[64] L. Zhang, M. Kim, and S. Khurshid. Faulttracer: a spectrum-based approach to localizing failure-inducing program edits. *Journal of Software: Evolution and Process*, 25(12):1357–1383, 2013.

[65] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, pages 765–784, 2013.

[66] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei. Prioritizing JUnit test cases in absence of coverage information. In *ICSM*, pages 19–28, 2009.

[67] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.