Assisting Developers Towards Fault Localization by Analyzing Failure Reports

Krystalenia Tatsi[1] **Kostas Kontogiannis**[2]

[1]National Technical University of Athens
Athens, Greece 15780

[2]Western University
London ON., Canada N6A 5B7

***Abstract*—Large software applications encompass many components with complex interdependencies. When a failure occurs, developers usually have limited information and time in their disposal for localizing the root cause of the observed failure. The most common information developers have readily access to includes failure reports, stack traces, and event logs. In this context, a major challenge is to devise techniques that assist developers utilize this information in order to zero-in their focus on specific methods that have a high probability of containing the root cause of the observed failure. Once such an initial set of methods has been identified, other more elaborate, complex, and computationally expensive data flow analyses could be applied. In this paper, we present a technique which aims to identify such an initial set of suspicious methods by first, retrieving information from failure reports obtained from Bugzilla repositories, second by combining this information with graph models that denote actual dependencies obtained from the subject system's source code in order to create an hypothesis space and third, by applying a ranking score to identify methods that have high likelihood of containing the root cause. The technique is shown to be tractable when applied to systems with several thousands of source code methods and exhibits high accuracy on the obtained results.**

I. Introduction

A common problem software developers face when a failure occurs, is to locate its root-cause [4]. This is a daunting task when the system comprises several thousands of files and very complex interactions. Event logs and stack traces may provide valuable information, but these are not always easy to decipher especially when the root cause is hidden deep in a call sequence, or it is embedded in complex data or control dependencies which can only be revealed by performing detailed data flow analysis [1], [2]. Furthermore, developers do not always have readily access to the appropriate infrastructure in order to replicate the state of the system when a failure occurred. This is especially true for mobile applications that depend on a number of unforeseen parameters (e.g. the device's battery level, the network provider, the state of the device), or in large systems with complex component dependencies. Strict time-to-fix constraints may make also difficult for developers to re-run test cases or accurately replicate the failure. Finally, many industrial systems are not written in one single language, but comprise different modules written in different languages, including also scripting languages and complex configuration files. This makes difficult for detailed data flow analysis to be applied, as such analysis would require not only parsers for all languages used but also, specialized linkers so that an end-to-end data flow analysis (e.g. slicing and dicing) accross all modules can be applied. In this paper, we take a different approach, proposing a technique which provides to developers a ranked list of methods in which the root cause of the observed failure is most probably localized. Once such a list of methods or functions is identified, then a more detailed data flow analysis can be applied. This addresses not only tractability analysis issues but also, the multiple language problem mentioned above, as the methods which are listed as possible root causes, most often are contained in specific modules likely written in one programming language, or refer to a more localized system context. In our quest to devise such a technique, we aim to use information that is only available in the bug repository entries, minimizing thus the dependence of the fault localization process on specialized and detailed source code analyzers.

In this respect, the question is "*whether it is possible, by using only information available in the bug repository, to help developers identify a set of functions or methods where with a high degree of confidence the root-cause of the observed failure resides*". Information that is typically found in bug repositories contains descriptions of the observed failure, developers' comments, and stack traces. Our premise was put into test by developing a layered approach towards root-cause analysis and fault localization. More specifically, we propose a technique which operates in two main phases, an expansion phase, and a matching phase. In the expansion phase, first the source code entities that appear in an initial failure report R are expanded by considering all source code entities to which these entities have an incoming or an outgoing source code relation and second, by collecting and expanding all reports in the repository which are conceptually similar to the initial failure report, using Latent Semantic Indexing and cosine similarity, yielding thus an hypothesis space. A ranking score is then used in order to rank the source code entities in the hypothesis space. The ranking score is based on how the source code entities in the hypothesis space relate with source code entities of the initial report.

The objective behind this expansion and consequent ranking

is twofold. First given a small set of source code entities we want to create a search space that is large enough to ensure within a level of confidence that the root cause will not be omitted, aiming thus for a high recall. This is the expansion phase. Second, once such a large but coherent search space is identified, we would like to select those elements that highly relate with the extended set of the tokens found in the initial bug report. This is the filtering or selection phase.

The approach has been tested in six large open source systems each one of which is written in various programming languages, with promising results. The results indicate very high levels of recall and the majority of the methods which constitute the root cause consistently rank in the top 20 positions among several thousand candidates in the search space. The system suffers from low precision as many candidates are returned in the result, but the ranking mechanism compensates to a certain extent for the low precision.

This paper is organized as follows. In Section II we present related work in bug localization. In Section III we outline the architecture of the proposed system, while in Section IV we present the domain model and the bug report token extraction phase. In Section V we discuss the generation of the search space and we present the ranking of the elements of the search space. In Section VI we present the results of applying the proposed method in the bug repositories of six large open source systems, while in Section VII we conclude the paper and provide pointers for future research.

II. Related Work

Fault localization is an area where significant and intensive research has been conducted over the past years. The approaches which have been proposed by various research groups fall in three main categories. The first category encompasses techniques that are based on the analysis of information that can be extracted either by the static analysis of the source code, or the analysis of test case traces. In this category indicative works include the approach proposed by Chen and Cheung [2] where the concept of dynamic dicing for program debugging is introduced, the work by Agrawal et.al [1] where specific input data from successful and unsuccessful tests are used to compute more accurate slices for program debugging, and the work by Wong and Sugeta [3] where execution dices are combined with the premise that code that passes more tests is likely not to contain faults, while code that is involved in more failed tests is more likely to contain faults. Another interesting approach is the work by Renieris and Reiss [5] where the authors present a nearest neighbor based approach between successful and unsuccessful tests modeled as sequences of statements executed by these tests, combined with a program dependence graph for capturing additional statements which may be the root-cause of the observed failure, and the work by Jones and Harrold [6], where the Tarantula system is presented.

The second category deals with the use of data mining and machine learning techniques that aim to associate specific input data, program states, or outcomes of test cases with specific system behavior or system faults. In this category indicative works include the approach by Wong and Qi [8] where back propagation neural networks are trained by coverage data (i.e. executed statements) for each test as well as by the outcome of each such test. Other works in this category include the approach by Brun and Ernst [10] where properties of correct and incorrect models are used to build a learning model. Once the model is built, properties of new programs can then be used by the model to infer how these properties associate with faulty behavior. The work by Livshits and Zimmermann [11] introduces a system which utilizes the mining of revision histories as well as dynamic analysis in order to reveal application specific coding patterns some of which are responsible for inducing system failures. The work by Zhou et al. [12] introduces an approach where text mining and machine learning are utilized to classify bug reports as corrective or as entries related to adaptive or perfective maintenance operations. The work by Nguyen et al. [20] utilizes a topic based model to associate entries of a bug report with source code structures in order to locate buggy source code files given a bug report. The work by Le et al. [19] proposes a multi-model approach for locating bugs by combining information retrieved from bug report repositories and program data collected during the execution of test cases. The approach builds a model that maps a bug to its possible location. The work by Chappell et al. [22] compares initial results obtained by using machine learning techniques for finding bugs with the results obtained by using classic static analysis techniques. An analysis of different techniques for bug localization which use text analysis models can be found in Rao [21].

Finally, the third category deals with model-based approaches which aim to utilize models that denote various types of dependencies such as cause-effect relationships, data and control flow dependencies in source code entities, or dependencies that explain the difference between the observed and the expected system behaviors. In this category indicative works include the work by Wotawa et al. [14] where first order logic is used to classify properties of programs and test cases and, the work by Mateis et al. [13] which utilizes dependency models to denote the structure of a program, while first order logic models are used to denote program behaviors. The work by Zawawy et al. [15] uses goal models to denote cause-effect relations between system behavior and system properties. SAT solvers and probabilistic reasoning can then be used to verify or deny specific hypotheses given collected event logs from the running system.

Our system falls between the category of systems that perform data mining and the systems that perform source code analysis. More specifically, data mining is applied to scout the bug repository, while source code analysis is applied to extract source code relations from the system under consideration.

III. System Architecture

The system follows a data flow architectural style which is composed of four main phases as depicted in Figure 1. In the first phase, an extraction component utilizes an XMLRPC client to read reports from Bugzilla repositories and populate

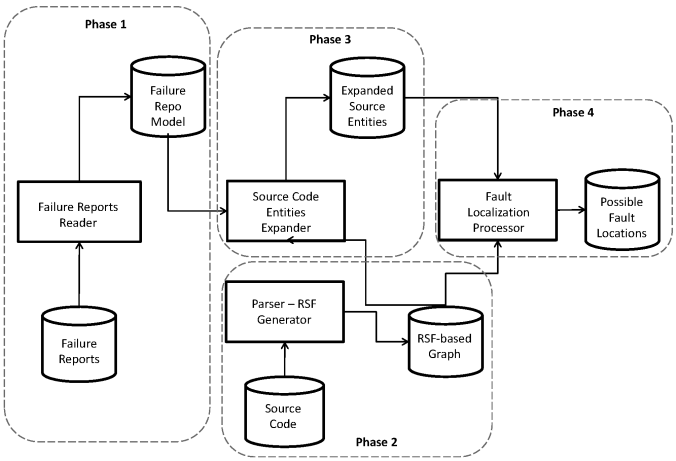

Fig. 1: Overall System Architecture

a meta-model yielding an instance model for each Bugzilla record extracted by the reader. The instance model is then stored in the form of JSON records in MongoDB for persistent storage and further processing. The meta-model serves as an isolation layer between the different types of bug repositories and the back end analysis. In this respect, different readers can access different types of bug repositories and populate the same meta-model. In the second phase, the source code of the subject system is parsed and a graph model of its source code is created. The nodes of the graph correspond to source code entities such as *files*, *functions*, *variables*, and *types*, while edges correspond to relations such as *includes*, *calls*, *uses*, *sets*, *defines*, and *declares*. In the third phase, a bug report preprocessing component utilizes NLP as well as the information stored in MongoDB in order to identify in each bug report the tokens that correspond to source code entities. This is achieved by comparing each token, through a hash table, with entities extracted from the source code. Furthermore, in this third phase the set of the source code entities found in the initial report is augmented by additional source code entities from other reports collected by using Latent Semantic Indexing and applying cosine similarity based search and expansion. This augmented set forms the hypothesis space. Finally, in the fourth phase, the bug localization is achieved by analyzing the source code system graph against the expanded set of source code entities and by ranking the hypotheses.

IV. Information Extraction and Modeling

A. Source Code Representation

In order to generate a graph model of the source code entities (SCE) we have used the Fetch framework (Fact Extraction Tool-Chain) [23]. This tool chain is composed of the *Red Hat Source Navigator* which parses the code and generates an intermediate *dbdump* as a collection of *.snav* files, the *Snav2Famix* which analyzes the *.snav* files and generates a *.cdif* file which is a comprehensive representation of the AST of the source code parsed, and the *cdif2rsf* tool which generates a *.rsf* file (Rigi Standard Format file) that contains source code entities (i.e. files, functions, variables, types) and relations between them.

More specifically, the *rsf* file provided by *cdif2rsf* represents source code entities and their relations by triplets in the form:

$$ENTITY1 \xrightarrow{\text{RELATION}} ENTITY2 \quad (1)$$

B. Bug Report Domain Model

The purpose of the domain model is to abstract and denote bug repository information in a way that is agnostic to the underlying structure or schema utilized by the specific repository framework. The outline of the domain model is depicted in Figure 2. The key element of the domain model is a *BugReport* class which is associated with a *Component* which in turn is part of a *Product*. A *BugReport* is associated with a *Description* which is a narrative of the symptom written in plain natural language along with possibly additional information such as stack traces, and references to source code entities (e.g. file names, variable names, function names and, types). A *BugReport* record may have dependencies with other records. Such dependencies pertain to whether a record *blocks* or is *blocked-by* another record or is a *continuation-of* or *depends-on* other records. A *BugReport* may be associated with a *Comment* which is also written in free plain text and may contain additional information with references to specific source code entities. Finally, a *Comment* is also associated with a *Creator*, an *Assignee* and, an *End_User*.

An example JSON segment denoting information extracted from a Bugzilla repository for the gtk+ system is presented below.

```
{
"BugReport": [
{
"creation_time": "Wed Jan 07 11:08:00 EET 2004",
"resolution": "INCOMPLETE",
"summary": "Crash in GtkTextView
(gtk_text_layout_validate_yrange)",
"severity": "critical",
"product": "gtk+"
"status": "VERIFIED"
}
],
"comments": [
{
"creation_time": "Wed Jan 07 11:05:07 EET 2004",
"text": "Package: gedit Severity: critical
Version: GNOME2.2.2 2.2.2
Debugging Information:  Backtrace was
generated from '/usr/bin/gedit'  (no debugging
symbols found)...Using host libthread_db
library \"/lib/tls/libthread_db.so.1\".
[Thread debugging using libthread_db
enabled] [New Thread -1218553264
(LWP 3542)] 0x00a6d5ce in _dl_sysinfo_int80 ()
from /lib/ld-linux.so.2 #0  0x00a6d5ce
in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
}]}
```

C. Failure Report Extraction Phase

The first part of the overall fault localization process is to extract, represent, and store the different entries (i.e. records) found in bug repositories such as Bugzilla. The failure report

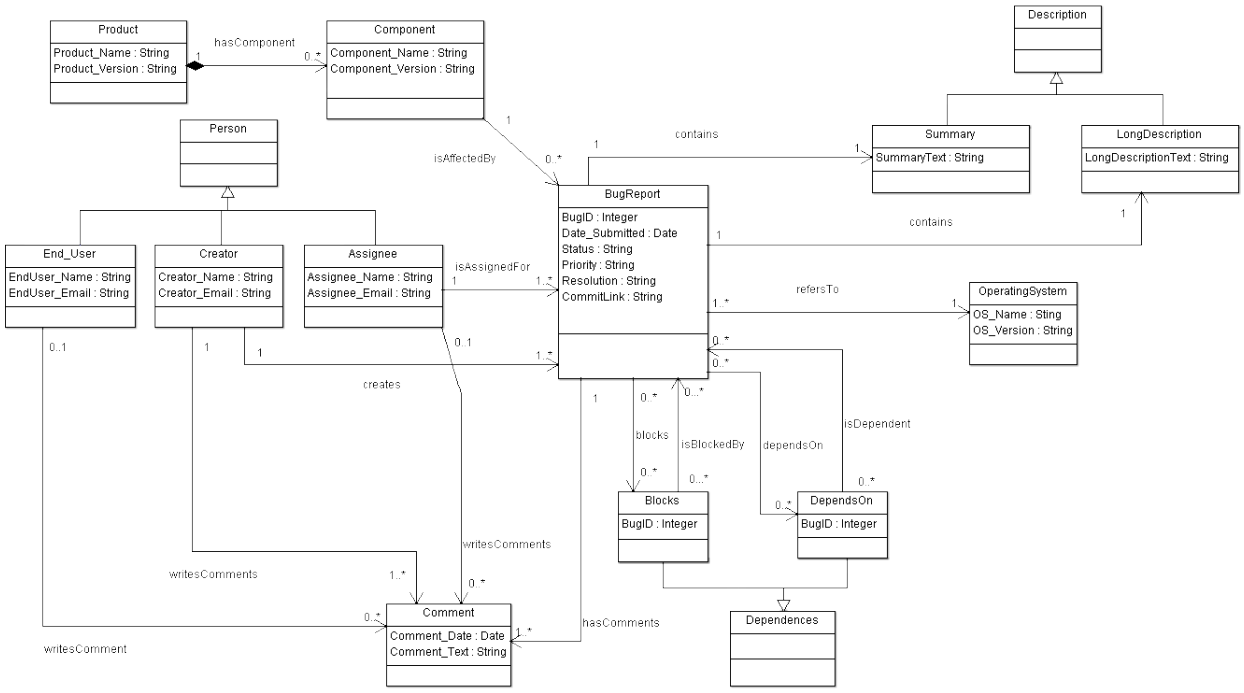

Fig. 2: Bug Repository Meta Model

extraction phase is based on the proxy pattern [9] where an abstract interface API and a dedicated domain model are used to decouple the specific service APIs offered by different bug repositories (e.g. Bugzilla, Mantis, Trac, TeamConcert), as well as their record structure and schema, from the back end analysis components. Our extraction service API is used to a) establish connection to different repositories, b) extract records, and c) parse the records and populate the domain model.

The bug report extraction framework utilizes the XMLRPC protocol to access a specific bug repository for a given product. The obtained information is then passed through an analyzer/populator which instantiates the corresponding part of the domain model for each entry read. Consequently, a transformer module, transforms the extracted reports which are now represented in the unified domain model, in JSON format, and a loader module stores each obtained report to a MongoDB which is configured for this task. However, the descriptions and the comments found on each Bugzilla repository record do not conform to a specific standard, although a basic set of fields and reporting guidelines are prescribed for use by each support team. In this context, we have to identify and process the source code entities which are embedded in the submitter's comments and description of the observed failure, as these constitute important pieces of information for fault localization. These tokens of interest include names of source code related entities such as directory names, module names, files, functions, variables, and types, as well as the context in which these appear (e.g. configuration options, pre-processing directives, environment variables). Since the *description* and the *comments* sections of a bug report are free text narratives, their preprocessing using a tokenizer is a necessary first step. The preprocessing step receives as input a set of free text segments, one for each report *description* and *comments* section, and outputs a set of strings where each one is a source code entity, omitting all the unrelated free text words. For this work, we have utilized Apache's NLP analyzer which parses and tags the description and comments section of the bug report, eliminating at the same time stop words. The extracted collection of tokens is then compared with the tokens that appear in the .rsf file that is generated by the Fetch tool chain, as discussed in Section IV-A. Such comparison is based on the use of a hash table and is performed once. The result is a set of source code entity tokens $t_1^R, t_2^R, ...t_k^R$ (i.e. file names, function names, variable names, and types) for each bug report record R in the repository $\mathcal{D}$.

V. Fault Localization

By examining a large number of cases, we observed that the set of the source code entity tokens that appear in the initial bug report (i.e. the first reporting of a failure), in most cases does not contain the root cause of the observed symptom. For this reason, before we are able to localize the root cause we need to expand this initial set (we refer to it as the search space generation phase) and consequently rank each element in the search space with respect to its "connectedness" to the tokens of the initial set. To better illustrate the approach we refer to Figure 3.

As depicted in Figure 3, let R be the initial failure report risen by the specific failure we focus on. As bug reports form a chain in a bug repository $\mathcal{D}$, an initial report is the first report

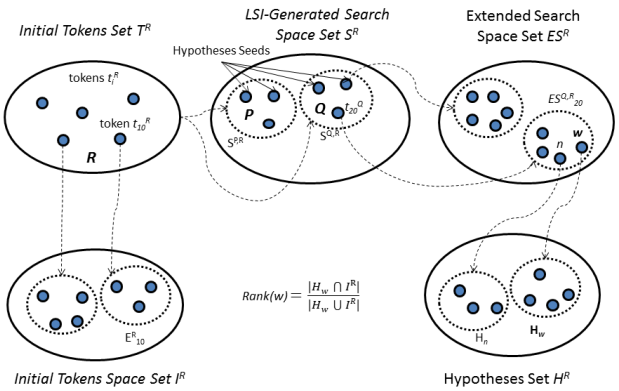

Fig. 3: Expansion Example

appearing on this particular chain of records related to this specific failure. In this respect, after applying the NLP tagging, stop word elimination and, source code token identification, let $T^R = \{t_1^R, t_2^R, ..., t_k^R\}$ be the set of source code entity tokens extracted by the report record R. For example in the upper left corner of Figure 3 we observe the set T^R, containing the tokens t_1^R, t_2^R,... ,t_k^R.

The expansion of the set T^R is performed in four steps. In the first step, each token $t_i^R \in T^R$ is expanded by collecting all other source code token entities which are connected to t_i^R by using the relations *Accesses*, *Sets*, *DefinedIn*, *DeclaredIn*, *MethodBelongsToClass* and, *Calls* in the .rsf file. The relations which are used for such an expansion according to the different types of source code entities involved, are depicted in Table I. For example, if token t_{10}^R is the function name *"getCustomer'*, then any other function that is called by *"getCustomer"* or *"getCustomer'* calls, will be collected through the *Calls* relation. We refer to this set as E_i^R. The union of all such sets E_i^R for $i = 1, 2, , k$ is referred to as I^R. For example, as depicted in the lower left part of Figure 3, the token t_{10}^R is expanded to set E_{10}^R in I^R.

In the second step, we identify all documents (i.e. report records) in the repository $\mathcal{D}$ which are similar to the initial report R, by applying Latent Semantic Indexing and cosine similarity, using all tokens $t_i^R \in R$ as a query in the corpus of documents $\mathcal{D}$. For each record $P \in \mathcal{D}$ that bears cosine similarity above a certain threshold (0.9 in our case) with R, we obtain its set of source code entities T^P yielding thus the set $\{t_1^P, t_2^P, ... t_n^P\}$. We refer to this set also as $S^{P,R}$, denoting that record P bears similarity with the initial record R. We refer to the union of all sets $S^{X,R}$ for all records X that bear similarity with record R as S^R, and we consider each token $t_j^X \in S^R$ as an initial hypothesis seed.

In the third step, each token t_j^X of a set $S^{X,R} \in S^R$, is expanded by incoming or outgoing .rsf relation paths of length 3. The union of all such expanded sets yields the expanded set $ES_j^{X,R}$. The union of all sets $ES_j^{X,R} \cup S^R$ yields the set ES^R. An expansion of more than three relations deep would yield a very large set in the fourth step, especially for systems that their source code entities are connected by many relations (e.g. when the system has a dense call graph) and generates significant amount of noise.

Source Code Type	Relation Type Used
GlobalVar	Accesses
Attribute	Sets
File	DefinedIn, DeclaredIn
Class	MethodBelongsToClass
Method	Calls
Function	Calls

TABLE I: Relations used to generate expansions

In the fourth step, each source code entity w in an expanded set $ES_j^{X,R}$ along with its neighboring source code (i.e. the source code entities that are related to w) forms the hypothesis space H_w for the element w. All expansions utilize *incoming* and *outgoing* relations for a given source code entity, as depicted in Table I.

In the example in Figure 3, a bug report record *R* is parsed and a set of tokens $T^R = \{t_i^R | i = 1, 2, , m\}$ is generated. Each such a token $t_i^R \in T^R$ is expanded to generate the set E_i^R. The union of all such sets produces the set I^R. At the same time reports P and Q are identified as similar to R using LSI and cosine similarity with token sets $\{t_1^P, t_2^P, t_m^P\}$ and, $\{t_1^Q, t_2^Q, t_k^Q\}$ respectively. The union of all these sets is referred to as the *Search Space Set* S^R, where each element $t_i^X \in S^R$ (e.g. the token t_{20}^Q in Figure 3) is considered a seed for generating the hypothesis space. In the next step, each such token t_i^X that appears in a report in S^R, is mapped to a fully extended set of tokens by gathering all source code entities that can be reached by a path of length 3. In this respect, for each $t_i^X \in S^R$ there is a corresponding set $ES_i^{X,R}$. The union of all $ES_i^{X,R}$ sets produces the set ES^R. For example, token t_{20}^Q (which was collected because report R bears similarity with report Q), yields the set $ES_{20}^{Q,R}$ which contains the tokens n and w. The expansion of tokens n and w of path length 3 in the rsf graph collects additional tokens which form the hypotheses sets H_n, and H_w respectively. The rank of the hypothesis w is provided by the formula in equation (3).

A. Latent Semantic Based Indexing and Expansion

Latent Semantic Indexing (LSI) takes a vector space representation of documents based on term frequencies as a starting point, and applies a dimension reduction operation on the corresponding term/document matrix using the singular value decomposition algorithm [16]. Similarities among documents and queries can be more reliably estimated in the reduced space representation than in the original representation. This is because documents which share topics will have a similar representation in the reduced space representation, even if they have few or even no terms in common. LSI is commonly used in areas such as web retrieval, document indexing [17] and feature identification [18]. Latent Semantic Indexing consists of the following phases:

First, is the creation of a vocabulary of tokens from entries of the input corpus (i.e. the bug repository). This is referred to as the text tokenization and is achieved by extracting terms from the entries of the input corpus and refining the terms by applying a normalization process and selection process. In our

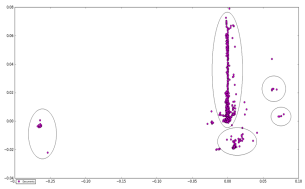

Fig. 4: LSI Bug Report Clusters for Amarok

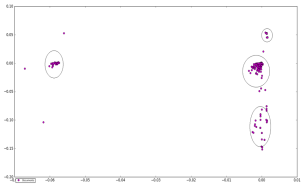

Fig. 5: LSI Bug Report Clusters for Dolphin

case the result of this step is a list of tokens that are related to source code entities of the subject software system. The tokenization and stemming steps are applied to all bug reports in the repository, and results to a complete vocabulary that contains flat words (source code entities such as file names, function names, variable names, types) from all records in the bug repository.

Second, is the creation of the term-document matrix. This is an m by n dimensional matrix where m represents the total number of documents in the corpus and n is the total number of terms (vocabulary) generated in first step. Thus, a row of this matrix is a signature vector corresponding to a bug report. A column of this matrix is a vector representing a term in the vocabulary.

Third, is the definition of a distance function between the terms of a bug report and a set of terms that are considered as search keywords. In the context of bug report indexing, each bug report R in the repository $\mathcal{D}$ is considered a document, and the similarity function is the cosine similarity between two vectors.

Finally, is the application of the cosine similarity function for obtaining the documents (i.e. bug reports) that best match a collection of search keywords (i.e. source code entities in the initial bug report) which are considered the query. In this respect, a conceptual similarity can be established between the tokens of the initial bug report and the rest of the bug reports in the repository.

As a simplified example, consider a bug repository which contains reports R_1, R_2, ..., R_n containing the source code entities *classA.method1(var1)*, *classB.method2(var2)*, *classA.method2(var1, var2)*. The vocabulary of terms is then $[classA, method1, method2, var1, var2]$, which is a vector of size 5. If the report R_1 in the repository contains the source code entity *classA.method2(var1, var2)*, then its term frequency vector index is $[1, 0, 1, 1, 1]$ indicating that in the first position which corresponds to the term *classA* in the vocabulary, this term appears once in R_1. The same idea holds for the other terms for the vector representing R_1. If we wanted now to compute the conceptual similarity between a bug report Q that contains the source code entity *classA.method1(var1)* and which has the term frequency vector $[1, 1, 0, 1, 0]$, with the bug report R_1 and which has the term frequency vector index $[1, 0, 1, 1, 1]$ as mentioned above, we estimate the cosine similarity between the two vectors. This estimated cosine similarity in this case is 0.75 [26]. The value of 1 indicates a full alignment while a value of 0 indicates orthogonal vectors (i.e. not matching vectors). A value of -1 indicates diverging concepts.

Bug reports with adjacent coordinates relate to similar concepts through cosine similarity and form concept clusters. Figures 4 - 5 depict such clusters formed in all bug reports for the Amarok and Dolphin systems (please see also case studies in Section VI).

In order to select the closest reports Q among the ones which bear cosine similarity with the initial bug report R we must consider a threshold value. This threshold value varies for each system being considered and can be set by the programmer, depending on how large the search space the programmer wants to be. We have considered our experiments with a threshold value of 0.9. The reports Q, P.... which bear cosine similarity above the threshold value form the Search Space Set S^R, as depicted in Figure 3. In order to trim the Search Space we also eliminate tokens which frequently appear in the bug reports in the S^R. Such tokens include the name of the system, and other tokens which for some reason appear in most reports (e.g. the token *public*). A separate redundancy threshold with value 0.7 is used to discard the tokens which are so frequent that basically provide only noise. For tokens in T^R which do not appear in any of the obtained reports in S^R, we set their score to max and keep them all. The formula used to discard the commonly occurring terms is based on the Inverted Document Frequency (IDF) concept as follows:

$$HF\_score(t) = log(\frac{\#\ \ reports \in S^R}{\#\ \ reports\ \ \ containing\ \ t}) \quad (2)$$

B. Hypotheses Ranking

The fault localization process is based on a score which depicts the "closeness" a source code entity w (i.e. the hypothesis for a root cause) has with other source code entities t_j^R, as these have been extracted from the initial bug report R. The "closeness" of a root cause hypothesis source code entity to the source code entities of the initial report are evaluated according to the reachability properties of the .rsf graph which denotes the dependencies and relations between the system's entities. The rationale is that the source code entities t_j^R which relate to the symptoms will have a path leading to the root causes of the observed failure, and that this path contains nodes (i.e. tokens) which are in the "vicinity" of, or a reachable by both the elements of the set I^R pertaining to source code entities of the initial symptom, and the elements of the set H_w pertaining to the source code entity w which is considered the hypothesis.

Referring to Figure 3 let us assume that the initial bug report R contains tokens (i.e. source code entities) t_i^R, where i = 1, 2,...,k. As discussed these tokens form the set T^R. By taking each token in T^R along with its neighboring tokens using the .rsf relations we create the set I^R. Furthermore, let us assume that we have applied Latent Semantic Indexing in order to represent as vectors of token frequencies each bug report in the

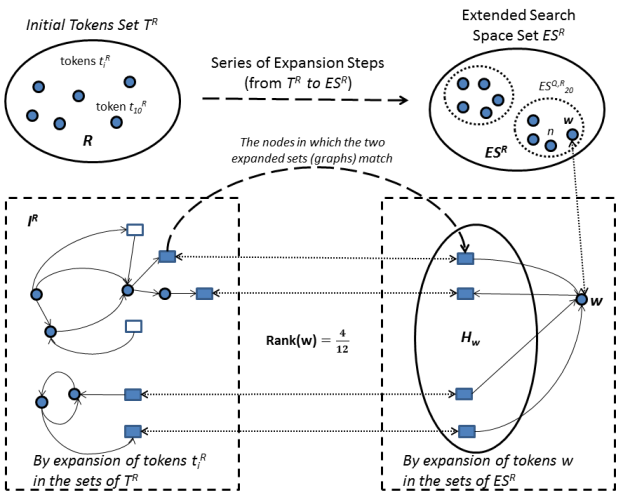

Fig. 6: Matching Schematic Example

repository $\mathcal{D}$. Using all tokens t_i^R in R as a query, we identify all bug reports which bear a cosine similarity with the initial report R that is above a certain threshold. For our experiments we have used a threshold value of 0.9. The collection of all such bug reports which bear similarity with R is identified as the search space S^R. Each element of such identified bug report (e.g. the token t_j^Q) is expanded by collecting all source code elements that are pointed-by or point-to the token (e.g. the token t_{20}^Q), creating clusters of tokens $ES_j^{Q,R}$. The union of all such clusters forms the extended search space ES^R. An element w in a cluster in $ES_j^{Q,R}$, is an hypothesis (i.e. a possible root cause). The hypothesis w in a cluster $ES_j^{Q,R}$ along with its neighboring elements using the .rsf relations forms the set H_w The hypothesis ranking score is based on the overlap between the tokens in H_w and the tokens in the extended initial token set I^R using the following formula:

$$Rank(w) = \frac{|H_w \cap I^R|}{|H_w \cup I^R|} \tag{3}$$

The rationale behind this approach is to find a maximal set that overlaps with the tokens found in the initial report as well as with the tokens that are in the neighborhood (i.e. can be accessed by .rsf relations) of the initial report.

As an example of the idea behind the ranking mechanism let us consider the schematic in Figure 6. On the upper left corner of Figure 6, the tokens of the initial report are collected and form the set T^R. This is also depicted in Figure 3. This set is extended as discussed in Section V yielding the set I^R which is depicted in the lower left part of Figure 6 and also in Figure 3. The nodes that pertain to three steps expansion from a token of the initial set T^R are depicted as rectangles in the lower part of Figure 6. This extended set corresponds to a graph that has the elements of the set I^R as its nodes, and the rsf relations between these nodes as its edges. Similarly, the set T^R is also extended by finding similar reports and in turn extending their elements as discussed in Section V, yielding the ES^R depicted in the top right part of Figure 6. Each element w in every set in ES^R is an hypothesis and yields the set H_w, depicted lower right part of Figure 6. The set H_w corresponds to a graph, as discussed above, which has the elements of the set H_w as its nodes, and the rsf relations between these nodes as its edges. This graph has common nodes with the graph that corresponds to the set I^R produced by the expansion of the initial report. In Figure 6 these common nodes are shown as solid filled rectangles. The rectangular nodes that match are shown in Figure 6 as aligned with the dashed double pointing arrows. The computed rank for w is then the cardinality of the intersection of the two sets (i.e. the number of the matched elements) divided by the cardinality of the union of the two sets. In the example of Figure 6 there are four matched elements (the four solid filled rectangles), while the cardinality of the union is 12 nodes (excluding w), yielding a rank of 4/12 = 0.33. The idea behind this technique is to compare the neighborhood of the nodes which are in the vicinity (in rsf graph terms) of the source code entities in the bug report, with the neighborhood of the nodes which are in the vicinity (in rsf graph terms) of the source code entities in all bug reports in the repository which are conceptually similar (in cosine similarity terms) with the initial bug report.

System Name	SLOC	# Files	# total Reports
Amarok	265,116	2,089	17,976
Dolphin	102,354	853	7,227
Kopete	387,753	3,729	9,765
Konqueror	112,436	6,248	38,135
GTK+	118,352	6,248	21,803
Nautilus	180,675	2,720	52,130

TABLE II: Details for the systems used as case studies

VI. Case Studies

In order to evaluate the proposed approach we have conducted a series of experiments on six large open source systems each of which is written in different programming and scripting languages. The evaluation focuses on assessing *a)* the performance and characteristics of the search space generation phase, and *b)* the accuracy of the obtained results as compared with the ground truth results indicated in the final resolution of the each initial bug report for which we had access to. We have evaluated our approach on the *Amarok 2.8* , the *Dolphin 2.1*, the *Kopete 1.0.80*, the *Konqueror 4.0*, the *GTK+ 3.19*, and the *Nautilus 3.8.0* system. The *Amarok* system is an open source music player running on Windows, Unix, and Linux platforms. The *Dolphin* system is a file management system and is part of the KDE released applications. The *Kopete* system is an industrial strength instant messenger framework that could integrate with a number of systems such as AIM, ICQ, Windows Live Messenger, Yahoo, Jabber, Gadu-Gadu and others. The *Konqueror* is a web browsing, file management system, and file browsing system for local or remote files. It is based on the KHTML engine, and it supports pdf and word editing as well as spreadsheet functionality. The *GTK+* system is a toolkit for implementing graphical user interfaces. It is part of the GNU project and it is suitable for small or large integrated applications. Finally, the *Nautilus* is a file management system

System	Avg. # of reports obtained via LSI	Avg. # of SCEs per input report (reports)	Avg. size of S^R (tokens)	Avg. size of ES^R (tokens)
Amarok	3,104	11.76 (7)	4,132	8,056
Dolphin	336	6.83 (12)	262	1,057
Kopete	1,457	10.7 (8)	5,537	9,569
Konqueror	5,215	113.4 (5)	8,263	11,774
GTK+	9,597	14.46 (13)	12,746	21,537
Nautilus	7,931	11.7 (7)	1,654	1,606

TABLE III: Details for the systems used as case studies

for the GNOME platform. It supports the management of local file systems as well as remote file systems accessible via FTP, HTTP, WebDAV, and SFTP servers. Basic source code size statistics and bug repository related statistics for these systems are depicted in Table II.

A. Search Space Generation Case Studies

The objective of these case studies is to evaluate the overall behaviour of the use of LSI for generating a coherent search space and consequently forming collections of hypotheses. The case studies focused on *a)* assessing how the LSI-obtained reports relate to each other and whether they form a coherent set, and *b)* assessing the characteristics of the obtained results as a function of the size initial report's token set.

Table III depicts the characteristics of the generated search space as a function of the size of the initial report's token set. More specifically, we have obtained on average 3,104 reports for Amarok per initial bug report by applying LSI and a similarity threshold score of 0.9. Each initial bug report out of the 7 we have considered, contains on average 11.76 source code entities. The generated search space S^R contains on average 4,132 source code entities for each of the 7 initial bug reports R being considered for Amarok, while the extended search space ES^R contains on average 8,056 tokens. Similar statistics are depicted for the other five systems we have examined and are also depicted in Table III. The important aspect to note is that starting with less than fifteen tokens in the initial report, the search space and the extended search space contains three orders of magnitude as many related tokens as the initial bug report R. For example, in GTK+, as depicted in Table III, the 14.46 tokens in the initial report yield, through the search space generation process, 12,746 tokens for the set S^R, and 21,537 source code entity tokens, for the set ES^R.

B. Accuracy of the Obtained Results

In this set of experiments, we have considered different initial bug reports for the six systems we have applied the technique on. Tables IV - IX present the obtained results. More specifically, as depicted in Table IV for the Amarok system, we have considered 7 different cases pertaining to 7 different initial bug reports for which we have access to their ground truth, that is how each of these initial bug reports were finally resolved by a *FINAL* type of record entry in the bug repository. In this respect, for each such initial bug report, we have clear and unambiguous information on which

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
323156	getTrack()	**86**/3286	100	0.0304
323614	slotShufflePlaylist()	**6**/8963	100	0.0335
	SortWidget()	**5**/8963		
	run()	**7**/8963		
323635	Base(QWidget)	**118**/8938	83.3	0.0335
	BallsAnalyzer(QWidget)	**Not Found**		
	BlockAnalyzer()	**19**/8963		
	resizeEvent(QResizeEvent)	**8**/8963		
	paintEvent(QpaintEvent)	**1**/8963		
	drawBackground()	**28**/8963		
325006	createMenus()	**Not found**	50	0.0112
	loadQtBinding()	**1**/8968		
328445	updateTimeLabelTooltips()	**14**/4537	100	0.022
334479	addTrack()	**5**/8926	50	0.0112
337725	toString()	**1**/5061	100	0.0198

TABLE IV: Fault Localization Results for Amarok

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
161385	setUrl(KUrl)	**2**/1112	100	0.089
250787	InformationPanelContent(QWidget*)	**14**/1090	100	0.460
	showItem(KFileItem)	**1**/1090		
	showItems(KFileItemList)	**2**/1090		
	showIcon(KFileItem)	**5**/1090		
	showPreview(KFileItem,QPixmap)	**3**/1090		
267171	run()	**1**/1297	66.67	0.154
	UpdateItemStatesThread()	**2**/1297		
	updateItemStates()	**Not Found**		
287829	selectedItems()	**2** /1276	100	0.078
302264	updateItemStates()	**5**/169	100	1.775
	setData()	**8**/169		
	run()	**25**/169		
303742	slotRoleEditingFinished()	**30**/911	100	0.120
304524	editedRoleChanged()	**8**/1231	100	0.162
	closeRoleEditor()	**2**/1231		
306147	slotRoleEditingFinished()	**64**/1330	100	0.075
306167	paint()	**5**/1400	100	0.071
306459	createSecondaryView(int)	**786**/1277	100	0.156
	KFileItemModel(QObject*)	**7**/1277		
307254	slotTrashActivated()	**3**/233	100	0.429
308018	closeRoleEditor()	**2**/956	100	0.209
	editedRoleChanged()	**1**/956		

TABLE V: Fault Localization Results for Dolphin

methods were modified in order for the initial bug report to be resolved. The tokens that correspond to the names of the methods/functions modified as part of the resolution constitute the ground truth for this initial bug report.

While conducting all of our experiments, we have excluded the *FINAL* resolution report from the repository.

Going back to the obtained results, in Table IV and for the initial bug report with ID 323156 its resolution entailed the modification of the method *getTrack()* in class *sqlRegistry* (not shown in Table IV for space considerations). The proposed technique for the bug report ID 323156 considered a search space of 3286 tokens (as these were generated by applying LSI, cosine similarity and expansion), and the score of the method *getTrack()* by applying equation (3) placed this method in the 86th position among 3286 candidates (first line in Table IV). The recall of the obtained 3286 results is 100% and the precision is 0.0304%. Similarly, for bug ID 323614 in Table IV there were three methods in the ground truth set which were placed in 6th, 5th, and 7th position among 8963 hypothesis candidates calculated by our expansion method.

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
153117	hp_removeDupe()	**3**/599	100	0.167
153533	slotAddClosedUrl(KonqFrameBase*)	**Not Found**	66.66	0.020
	testAddTab()	**4**/9819		
	testDuplicateSplittedTab()	**374**/9819		
155225	saveConfig()	**16**/9419	100	0.012
155434	focusNextPrevNode(bool)	**8**/10266	100	0.029
	setActiveNode(NodeImpl*)	**1687**/10266		
	setFocusNode(NodeImpl*)	**687**/10266		
156658	openBrowserWindow()	**6**/6889	100	0.087
	createNewWindow()	**5**/6889		
	...WindowWithSelection()	**4**/6889		
	...WindowFromProfile()	**1**/6889		
	...WindowFromProfileAndUrl()	**2**/6889		
	...ProfileUrlAndMimeType()	**3**/6889		

TABLE VI: Fault Localization Results for Konqueror

The recall was also 100% and the precision 0.0335%. In Table IV we observe also that there were two methods, the method *BallsAnalyzer()* which was not found by our method as it was not captured while generating the search space S^{323635} for bug report ID 323635, and the method *createMenus* pertaining to the resolution of the bug report ID 325006 which was also not found. More specifically, for the bug report ID 325006 the ground truth includes two methods *createMenus()* and *loadQtBinding()*. The second method was found by our technique and was even placed in the first position among 8968 candidate tokens. However, the Fetch tool was not able to generate a source code relation in the .rsf file which links these two methods. Therefore, it may be possible to enhance the accuracy of this technique by using source code analyzers which generate more detailed relations between source code entities.

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
243653	editAccount(QWidget*)	**Not found**	0	0
251226	logout(QString)	**1**/10020	100	0.0099
254494	setDisplayNameSourceContact()	**6**/6682	100	0.0897
265295	logout(QString)	**1** /10017	100	0.0099
268056	smt_messageSent()	**124**/10021	100	0.0299
	gotFileMessage()	**287**/10021		
	deleteTasks()	**74**/10021		
270797	TranslatorPlugin()	**12**/6676	100	0.0299
	TranslatorPlugin(QObject*,QStringList)	**37**/6676		
273070	TranslatorPlugin()	**12**/6674	100	0.0299
	TranslatorPlugin(QObject*,QStringList)	**37**/6674		
277606	setDisplayNameSourceContact()	**10**/6672	100	0.0149

TABLE VII: Fault Localization Results for Kopete

The results from the other five systems examined, indicate similar results. More specifically, in most systems and for most initial bug reports examined, the method captures the ground truth and exhibits high recall values. This is depicted in the Tables V - IX which list the results from the other five systems.

Furthermore, most of the obtained results are ranked within the first twenty positions of the overall search space. An exception can be seen in one method in bug report 306459 as shown in Table V. Similar exceptions occur for in bug reports 153533, 155434 in Table VI, in bug report 268056 in Table VII, in bug report 757282, 758442, 758609, 760942, and

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
757282	gtk_window_resize()	**9**/21555	100	0.0139
	..._configure_request_size()	**94**/21555		
	..._resize_to_geometry()	**3**/21555		
757805	get_shadow_width()	**2**/21572	100	0.0092
	gtk..._get_shadow_width()	**3**/21572		
758442	..._context_set_path()	**13**/21651	100	0.0092
	gtk..._add_to_widget_path()	**235**/21651		
758609	gtk_window_show()	**171**/21563	100	0.0092
	gtk_window_move()	**2**/21563		
758901	..._wayland_window_configure()	**5**/18407	100	0.0054
759091	get_widget_coordinates()	**5**/21555	100	0.018
	tab_prelight()	**1**/21555		
	get_tab_at_pos()	**11**/21555		
	..._leave_notify()	**2**/21555		
759299	..._window_set_transient_for()	**12**/21602	100	0.0046
759705	gtk_window_show()	**22**/21564	100	0.0092
	gtk_window_realize()	**5**/21564		
759764	warn_response()	**6**/21555	100	0.0046
760213	..._set_window_cursor()	**20**/21548	100	0.0046
760640	gtk_notebook_destroy()	**83**/21659	100	0.0046
760942	..._border_window()	**10**/9053	100	0.0111
761128	..._entry_draw_text(GtkEntry*,cairo_t*)	**132**/21535	100	0.0046

TABLE VIII: Fault Localization Results for gtk

761128 in Table VIII, and in bug reports 697890, and 703349 in Table IX. However, in almost all these cases a single method is affected, and we believe that the developers having seen the rest of the results for the same record, will be able to pin point the low ranked function as well. Future work should include the identification of additional relations which may link these methods in the ground truth set, so that we can further enhance the accuracy of the proposed method and increase the ranking score of the methods which are currently ranked lower that they should.

Finally, the examination of the bug reports for which their results were ranked low by our approach, revealed that these were isolated modules with very limited relations with other source code entities and for which the Fetch tool did not bring enough rsf relations to consider. It will be possible by considering another source code relationship extractor, to be able to increase the ranking score of these isolated entities.

Bug IDs	Methods	Rank / List Size	Recall (%)	Precision (%)
333265	real_update_menus(NautilusView*)	**10**/3247	100	0.031
697183	apply_columns_settings()	**6**/2695	100	0.074
	create_and_set_up_tree_view()	**3**/2695		
697890	filtering_changed_callback(gpointer)	**23**/3673	100	0.081
	invalidate_one_count()	**88**/3673		
	collect_all_directories()	**25**/3673		
698190	column_chooser_use_default_callback()	**1**/3673	100	0.109
	get_column_order()	**7**/3673		
	get_visible_columns()	**8**/3673		
	nautilus_list_view_reset_to_defaults()	**3**/3673		
702546	nautilus_window_slot_dispose()	**5**/4284	100	0.023
71480	custom_icon_file_chooser_response_cb()	**73**/4267	100	0.023
703349	append_directory_contents_fields()	**143**/2914	100	0.034

TABLE IX: Fault Localization Results for Nautilus

VII. Conclusion

Identifying the location of a bug given a symptom or a failure report is a difficult task. The software engineering

community has proposed a number of different techniques that aim to address the problem. One type of techniques aims to pinpoint the statements which constitute the root causes for a failure. These techniques utilize static and dynamic analysis, machine learning, as well as models that link root-causes with symptoms. These techniques require full access to test cases, test results, and specialized parsers and analyzers to perform specialized static analyses such as slicing and dicing. Another type of techniques aims to identify the files or even functions of the system which are highly suspicious of containing the root cause of an observed failure. In this paper, we report results of a system that allows for the identification of a ranked list of functions and methods that have a higher likelihood, based on our experimental results, to contain the root-cause of a failure. The major benefit of the approach is that it uses information readily available from bug repositories, and information that can be easily obtained from the source code using only simple parsers or scanners, eliminating thus at this level, the use of detailed source code analyzers. This work can be extended in a number of ways. First, it is interesting to experiment by considering a richer set of source code entity relations. This can be achieved by using a different or additional source code fact extractors and analyzers such as srcML. Second, the ranking method can be altered so that the score can be based on graph comparison as opposed to token matching. More specifically the tokens I^R that stem from the initial bug report along with their relationships can be considered as the "source" graph, while the hypothesis token set H_m along with the relations can be considered as the "target" graph. The ranking score can then be the distance between these two graphs. A third direction is to consider examining graph related properties between the hypothesis tokens and the tokens in the initial bug report. For example, we could investigate whether there are cliques formed when we consider these two sets as nodes of the rsf graph, and rank the resulting nodes by their connectivity strength and properties (e.g. using a hub and authorities type of analysis). Finally, one could also consider the history of changes, giving thus higher ranking into hypotheses that relate to error prone or frequently changed source code elements.

REFERENCES

[1] H. Agrawal, J. Horgan, S. London, W. Wong, "Dynamic Program Slicing". In Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation, pp. 246- 256, White Plains, New York, June 1990.

[2] Y. Chen, Y. Cheung, "Dynamic Program Dicing". In Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society, pp. 378-385 September 1993.

[3] W. E. Wong, T. Sugeta, Y. Qi, and J. C. Maldonado, "Smart Debugging Software Architectural Design in SDL". In Journal of Systems and Software, 76(1):15-28, April 2005.

[4] Iris Vessey, "Expertise in debugging computer programs: A process analysis". In International Journal of Man-Machine studies 23, 5, pp. 459-494, 1985.

[5] M. Renieris and S. P. Reiss, "Fault Localization with Nearest Neighbor Queries". In Proceedings of the 18th IEEE International Conference on Automated Software Engineering, pp. 30-39, Montreal, Canada, October 2003.

[6] J. A. Jones and M. J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault- Localization Technique". In Proceedings of the 20th IEEE/ACM Conference on Automated Software Engineering, pp. 273-282, Long Beach, California, USA, December, 2005.

[7] W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization". In Journal of Systems and Software, 83(2):188-208, February, 2010.

[8] W. E. Wong and Y. Qi, "BP Neural Network-based Effective Fault Localization". In International Journal of Software Engineering and Knowledge Engineering 19(4):573-597, June 2009.

[9] Frank Buschmann, Kevin Henney, Douglas C. Schmidt, "Pattern-Oriented Software Architecture, Volume 4, A Pattern Language for Distributed Computing". Willey Publishers, 2007.

[10] Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions". In Proceedings of the 26th International Conference on Software Engineering, pp. 480- 490, Edinburgh, UK, May 2004.

[11] B. Livshits, T. Zimmermann, "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories". In Proceedings of the 10th European Software Engineering Conference, ACM New York, pp. 296–305, Lisbon 2005.

[12] Y. Zhou, Y. Tong, R. Gu, H. Gal, "Combining Text Mining and Data Mining for Bug Report Classification". In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE Computer Society, pp. 311-320, September 2014.

[13] C. Mateis, M. Stumptner, and F. Wotawa, "Modeling Java programs for diagnosis". In Proceedings of the 14th European Conference on Artificial Intelligence, pp.171-175, Berlin, Germany, August 2000.

[14] F. Wotawa, M. Stumptner, and W. Mayer, "Model-based debugging or how to diagnose programs automatically". In Proceedings of the 15th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems: Developments in Applied Artificial Intelligence, pp. 746-757, Cairns, Australia, June 2002.

[15] H. Zawawy, S. Mankovskii, K. Kontogiannis, J. Mylopoulos, "Mining Software Logs for Goal-Driven Root Cause Analysis". In book "The Art and Science of Analyzing Software Data", eds. C. Bird, T. Menzies, T. Zimmermann, Waltham: Morgan Kaufmann, 2015, pp. 519-554.

[16] G. Golub and C. Reinsch, "Singular value decomposition and least squares solutions". In *Numerische Mathematik*, vol. 14, pp. 403–420, April 1970.

[17] L. Wu, J. Feng, and Y. Luo, "A personalized intelligent web retrieval system based on the knowledge-base concept and latent semantic indexing model". In *Software Engineering Research, Management and Applications, ACIS International Conference on*, vol. 0, pp. 45–50, 2009.

[18] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining probabilistic ranking and latent semantic indexing for feature identification". In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, (Washington, DC, USA), pp. 137–148, IEEE Computer Society, 2006.

[19] T. B. Le, R. J. Oentaryo, D. Lo, "Information Retrieval and Spectrum Based Bug Localization: Better Together". In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM New York, pp. 579-590, 2015.

[20] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, "A topic-based approach for narrowing the search space of buggy files from a bug report". In Proceedings of the 2011 Automated Software Engineering Conference pp. 263-272, November 2011.

[21] S. Rao, A. Kak, "Retrieval from Software Libraries for Bug Localization: A Comparative Study of Generic and Composite Text Models". In Proceedings of the 8th Working Conference on Mining Software Repositories, pp. 43-52, May 2011.

[22] T. Chappell, C. Cifuentes, P. Krishnan, S. Geva, "Machine Learning for Finding Bugs: An Initial Report". In Proceedings of IEEE International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE), IEEE Computer Society, pp. 21-26, Klagenfurt, Austria 2017.

[23] B. Du Bois, B. Van Rompaey, K. Meijfroidt, E. Suijs, "Supporting Reengineering Scenarios with FETCH: an Experience Report". In ECEASST Journal, vol. 8, Nov. 2007.

[24] V. Tzerpos, R. C. Holt, "ACDC : An Algorithm for Comprehension-Driven Clustering". In Proceedings of IEEE Seventh Working Conference on Reverse Engineering, IEEE, pp. 258-267, 2000.

[25] Famoos, http://scg.unibe.ch/archive/famoos/, Imber(1991)Imber:1991:CDI:184274.184298 Mike Imber, Software Engineering Environments, 1991.

[26] http://scistatcalc.blogspot.ca/ 2015/11/cosine-similarity-calculator.html