EXECUTION TRACE ANALYSIS FOR THE SOFTWARE DEVELOPMENT OF SAFETY-CRITICAL SYSTEMS

Judith Providence    Peter Kemper
College of William and Mary
Department of Computer Science
Williamsburg, Virginia 23185

Abstract

High quality software is an important part of mission critical systems. In order to attain high quality software and detect potential errors, the software must go through a rigorous testing process. Our goal is to automatically relate test failures based on the similarity of their execution traces. Our research involves determining the similarity of execution traces based on appropriate measures. More specifically, our research focuses on if more information results in a better identification of duplicate error reports. We compared three different approaches which were based on the cosine similarity formula. Each approach progressively contained more information about the execution traces. The approaches involved a binary-based, integer-based and a grammar-based version of the cosine similarity formula. Based on empirical evidence, we conclude that the binary-based approach out-performed the integer-based and grammar-based approach in determining the similarity of execution traces.

Introduction

Large software projects depend on test managers and system engineers to evaluate system correctness and system reliability. In this process, the system is evaluated to determine if it meets design requirements. In the validation process, the system is tested to determine if the design performs its intended task. In the testing phase, when a failure occurs, error reports are generated. As error reports are generated, execution traces which provide for a more exact detail and reproducibility of the error are also generated. After reviewing a large amount of error reports, the test manager faces the challenging task of clustering similar reports or execution traces. The clustering of the execution traces is useful for the assignment of personnel to fix the error based on experience. The clustering of traces reveals duplicate or similar execution traces. It allows the test manager to identify hotspots. In this paper, we work on identifying a measure that will identify similarity in execution traces.

Related Work

There have been several approaches developed, for the detection of duplicate error reports. The majority of these approaches use natural language information
and information retrieval techniques to relate reports. However, Wang et al.\(^9\) which we will discuss later used natural language and execution information. The use of execution information aligns itself with our work.

In natural language processing, an automated approach to the error report problem is described by Anvik et al.\(^1\). The authors compare the filtering out of duplicate reports to the filtering of spam in email. They state that if most spam can be automatically filtered out then by using an automatic process, most duplicate reports can also be filtered out. The authors presented an automatic approach to the detection of duplicate error reports based on cosine similarity. This automatic process detects 28% of all duplicate error reports in the Firefox bug repository. In this approach, machine learning techniques are first employed to build a statistical model. The statistical model is updated constantly through the submission of new error reports. Cosine similarity is then applied to this statistical model. If a new error report is tagged as being a duplicate by this statistical model, then the three most similar existing reports are presented to the triager for examination. The triager then examines the suggested list to determine if the new error report is a duplicate. However, as stated in Jalbert et al.\(^4\), the method described above, incorrectly filtered out 10% of non-duplicate error reports in the dataset.

The results from the natural language processing approach used by Runeson et al.\(^7\) resulted in finding 40% of all duplicate error reports. This approach uses the natural language text of the error report coupled with natural language processing techniques such as tokenization, stemming, stop words removal, vector space representation, and similarity calculations to determine duplicates. The natural language processing techniques were evaluated on the Sony Ericsson defect management system. In the natural language approach, the authors developed a thesaurus of the most common words used. They compared error reports within a certain time frame through the error report’s description field, header, and the project name. However, this approach was based on a suggested list size of 5 and 15 and required human intervention. In Jalbert\(^4\), the authors use surface features, textual semantics and graph clustering to detect duplicate error reports. Their approach detected 8% of all duplicate error reports. The authors performed distance metrics on the title and the description of the error report. Natural language processing techniques were then applied to the textual data. Cosine similarity was applied to the documents in the corpus. The textual similarity metric was used to produce a graph. A graph clustering algorithm was applied to the graph. However, due to the low recall rate of 8%, research is still needed to determine similarities between duplicate error reports.

Another approach developed by Hiew\(^3\) which also uses the natural language text of the error report, detected 29%-50% of all duplicate error reports in the Firefox bug repository. Natural language processing techniques were used to cluster groups of similar reports. A model was then built based on this clustering. When a new error report was submitted, its textual information was compared against this model. As a result of this comparison, a list of potential duplicate error reports were sent to the triager. This approach also resulted in a low recall rate.

The authors Wang et al.\(^9\) used natural language information and execution trace information to determine duplicate error reports. By using both natural language
and execution traces, the authors’ goal was to increase the number of duplicate reports detected above that of the previous results noted in Runeson et al.\(^7\) In Wang et al., the use of not only natural language but execution information resulted in the detection of 67%-93% of all duplicate error reports in the Firefox open source bug repository. The authors contend that the use of natural language techniques does not efficiently detect duplicate error reports. The use of execution traces would detect similar abnormal behaviors in software applications which coupled with natural language techniques would detect a higher rate of duplicate reports. The authors’ approach was to calculate both the natural-language based and the execution-information based similarities between the new error report and existing error reports. These calculations were based on a set of two heuristics. As a result of these calculations, a suggested list of possible duplicate error reports were generated. This list of potential error reports were then sent to the triager for examination. In the approach described by Wang et al., the authors used equation 2 to calculate the vector space model. However, it is stated in Jalbert et al.\(^4\), based on empirical data, the inverse document frequency is not effective in determining the similarities between duplicate error reports. This approach also produced a suggested list, which requires human intervention. More recently, Sun et al.\(^8\) achieved a recall rate of 37%-71% using a modified version of the $BM25F$, a similarity measurement tool specifically designed to retrieve duplicate error reports. Their technique applies to natural processing of duplicate error reports. The authors refer to the summary and description of the error report as textual features. All other features of the error reports are considered categorical features. Such categorical features are the version numbers, the priority and the product field. Duplicate error reports are processed by using informational retrieval techniques such as stemming and stop word removal. The approach determines the similarity in the textual and categorical features of the error reports. The authors extend the $BM25F$ by the addition of a weight scheme, which is the weight of the terms in the query. A retrieval function is used to record the textual and categorical information. The retrieval function has 19 free parameters which is optimized by the use of gradient descent.

Motivating Example

In this section, we present duplicate error reports from the Eclipse bug repository. The steps to reproduce are different for each report. They both share a common step which is to enable multi-stroke help. A test manager who is responsible for looking through several hundred error reports might not realize that these two error reports are duplicate. An error report consists of a title and a textual description on the nature of the error and necessary steps to reproduce the error. Tables 1 and 2 cite error reports 66674 and 66182 from the Eclipse bug repository that both describe an error with a multi-stroke keyboard binding in Eclipse. While both show similarities with words “multi-stroke”, “key” and “key binding”, which supports some kind of relationship, the details of the reports differ substantially such that an identification as duplicates is highly unlikely. Wang et al.\(^9\) also gives an example of similar reports that are based on common words and a difference in equivalent meanings which would not be detected as duplicates. These error reports reveal that the same error can be identified and
expressed by error reporters in many different and subjective ways. While natural language processing techniques are advanced enough to be able to process these varied descriptions into a usable similarity measure, we believe that execution traces provide an objective and accurate description on how to produce the error as well as a description that is rich in detail. We will discuss three ways to measure similarity between execution traces.

Table 1. 66674 in the Eclipse bug repository

| Error-66674 |
| Problem with Multi-stroke help |
| Description |

Error report 66674 was submitted on June 11th, 2004, it included the description, "If I configure the Keys to use the popup after a certain time, selecting the letter when the popup is visible does not open the selected view. Once the popup is visible, the arrow keys must be used to select a view."

Steps to Reproduce
1-Type multi-stroke SHIFT-ALT-Q
2-Y to open synchronize view. Close the view.
3-Type multi-stroke SHIFT-ALT-Q to get popup
4-Type Y

Table 2. 66182 in the Eclipse bug repository

| Error-66182 |
| KeyBindings- multi-stroke keyboard shortcut popup swallows key |
| Description |

Error report 66182 was submitted on June 8th, 2004, it included the description, "Nothing happens for all multi-stroke keybindings."

Steps to Reproduce
1-Preferences - Workbench - Keys
2-Choose Emacs configuration
3-On the Advanced tab, check "Help me with multi-stroke keyboard shortcuts"
4-Edit a file
5-Press Ctrl-X
6-Wait until the popup shows up
7-Press Ctrl+S

Definitions

The Cosine similarity measure is based on the vector space model. The vector space model is used as a widely known technique in information retrieval\(^9\). In the vector space model, documents are represented by \(n\)-dimensional vectors where \(n\) represents the number of unique index terms in the document and \(w_i(1 \leq i \leq n)\) is the weight of the \(i_{th}\) index term in the vector \(<w_1, w_2, \cdots, w_n>\).

\[
w_i = tf_i \times idf_i \tag{1}
\]

The term frequency is represented by \(tf_i\), which is the number of occurrences of the \(i_{th}\) index terms in the document. The \(idf_i\) or inverse document frequency is represented by equation 2.

\[
idf_i = \log \frac{D_{sum}}{D_{w_i}} \tag{2}
\]

Where \(D_{sum}\) is the total number of documents and \(D_{w_i}\) is the number of documents that contain the \(i_{th}\) index term. We have adapted this technique to determine the similarity between two traces. In our research \(w_i\) was modified for all three of our approaches.

For a pair of documents, one obtains two vectors \(q_1 = <w_{11}, w_{12}, \cdots, w_{1c}>\) and \(q_2 = <w_{21}, w_{22}, \cdots, w_{2c}>\), and the cosine similarity of the two documents is defined as

\[
\text{CosineSim}(q_1, q_2) = \frac{\sum_{i=1}^{c} w_{1i}w_{2i}}{\sqrt{\sum_{i=1}^{c} w_{1i}^2} \sqrt{\sum_{i=1}^{c} w_{2i}^2}} \tag{3}
\]
Technical Approach

Error reports from a large open source code base were analyzed to better understand the real world problems of determining the similarity of execution traces. Our dataset was based on the 2004 Eclipse bug repository. Two hundred twenty-five execution traces were created from various error reports in the 2004 Eclipse bug repository which included 44 duplicate pairs. Methods from each execution trace were used for evaluation. Execution traces can be analyzed at different levels of granularity. These levels include package/module, class, method, and statements.

The granularity of methods were also used for several reasons. Statements are very fine grained and produce an overwhelming amount of data. Packages/modules or class are very coarse grained and may not distinguish among different errors in a single package or class. Methods provide a reasonable middle ground to recognize similar usages of functionality. It is an established way of determining similarities in error reports\(^9\).

The amount of information taken into account for the similarity measure included:

- **Binary** - is a method called or not
- **Integer** - how often is each method called
- **Sequential** - in what order are methods called

In using a similarity measure, related conceptual problems are easily recognizable. Given a set of traces \(S\), a trace \(x\) can then be recognized as being new or as being a part of the original set \(S\). Given a set of traces \(S\), the traces can then be partitioned into groups based on their similarity. To determine similarity, we evaluated a binary-based, an integer-based and a grammar based approach. The binary-based approach is associated with Wang et al. approach to determining duplicates in execution traces.

In the binary-based approach, our dataset was converted to binary vectors in which the data attributes depended on the presence or absence of a method in the execution trace. In the integer-based approach, we evaluated the frequency of the methods by using the cosine similarity formula. In the grammar-based approach, we evaluated the frequency of rules based on rules generated from Sequitur. The grammar-based approach, uses Sequitur\(^6\) to compress traces into a set of rules which consists of digrams. Sequitur is an algorithm that replaces phrases that occur more than once by grammatical rules. It forms a hierarchical structure that represents the original sequences. It acts as a compression technique for large datasets. It has been extensively used in various areas such as in the compression of multi-megabyte DNA sequences. A key characteristic of the algorithm is that of the algorithm’s linear time processing capability. The algorithm uses hash tables for the adding and deletion of symbols. It uses pointer assignments which eliminate the need for arrays and that also provide constant-time access. We have applied Sequitur to execution trace data. Sequitur is based on two constraints on a context-free grammar:

\(p1\): no pair of adjacent symbols appears more than once in the grammar

\(p2\): every rule is used more than once

**Results**
Figure 1. binary-based

Figure 2. binary-based

Figure 3. integer-based

Figure 4. integer-based
The six figures is a graphical representation of the cosine similarity measures between duplicates and non-duplicate reports in the three approaches. In the figures, 1855 non-duplicate pairs of error reports were compared against 44 pairs of duplicate error reports. Our goal is to achieve a clear separation between the duplicates and non-duplicate error reports. The binary-based approach shows the greatest amount of separation between the duplicate and non-duplicate error reports. The integer-based approach shows a moderate amount of separation. The grammar-based approach results in the least amount of separation between the duplicates and the non-duplicate pairs. The binary-based approach achieves that goal.

Benefits to Aerospace Applications

Execution traces can result in determining how error traces differ from correct traces of a safety critical software program. Detecting duplicates may help to locate errors and relate failures to test traces. The third benefit is that execution traces can be implemented in the testing phase of a project. The fourth benefit is that execution traces can reduce the cost of testing.

Conclusion

We used 225 error reports which included 44 pairs of duplicate error reports. We evaluated a binary-based, integer-based and a grammar-based approach. The three approaches were based on the cosine similarity formula. Based on empirical evidence, the binary approach excels in the detection of duplicate error reports when using execution traces. The approaches that were reviewed in this report demonstrated that more detailed information is not necessary in determining the similarity of execution traces.

Acknowledgements
Special thanks is extended to Dr. Poshyvanyk and Bogdan Dit for helping to obtain the datasets for our research.

References


