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. Our approach shows promising results with real data.

Introduction

Large software projects depend on test managers and system engineers to verify and validate system correctness. System correctness leads to reliability. In the verification process, the system is evaluated to determine if it meets the design requirements\(^2\). 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 is 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 or bug reports. The majority of these approaches use natural language information and information retrieval techniques to relate reports. However, Wang et al. \(^{12}\) 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 bug report problem is described by Anvik et al. \(^1\). This approach uses textually information. 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 bug reports based on Cosine similarity. This automatic process detects 29% of all duplicate bug 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 bug reports. Cosine similarity is then applied to this statistical model. If a new bug 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 bug report is a duplicate. However, as stated in Jalbert et al. \(^4\), the method described above, incorrectly filtered out 10% of non-duplicate bug reports in the dataset.

The results from the Natural Language Processing approach used by Runeson et al. \(^9\) resulted in finding 40% of all duplicate bug reports. This approach uses the natural language text of the bug 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 bug reports within a certain time frame through the bug report’s description field, header and the project name. However, this approach produces a suggested list size between 5 and 15 which requires human intervention. In Jalbert \(^4\), the authors use surface features, textual semantics and graph clustering to detect duplicate bug reports. Their approach detected 8% of all duplicate bug reports. The authors perform distance metrics on the title and the description of the bug report. Natural Language Processing techniques were then applied to the textual data. Cosine similarity is then applied to the documents in the corpus. The textual similarity metric is then used to produce a graph. A graph clustering algorithm was then 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 bug report, detected 29%-50% of all duplicate bug 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 bug report was submitted, its textual information was compared against this model. As a result of this comparison, a list of potential duplicate bug reports were sent to the triager. This approach also resulted in a low precision and recall rate.

The authors Wang et al. \(^12\) used Natural Language information and execution trace information to determine duplicate bug 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. \(^9\). In Wang et al., the use of not only natural language but execution information resulted in the detection of 67%-93% of all duplicate bug reports in the Firefox open source bug repository. The authors contend that the use of natural language techniques do not efficiently detect duplicate bug 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 were to calculate both the natural-language based and the execution-language based similarities between the new bug report and existing bug reports. These calculations were based on a set of two heuristics. As a result of these calculations, a suggested list of possible duplicate bug reports were generated. This list of potential bug reports were then sent to the triager for examination. In the approach described by Wang et al., the authors use the equation (3) 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 bug reports. This approach also produced a suggested list, which requires human intervention.

An 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 bug reports might not realize that these two error reports are duplicate. These two examples illustrate that it is not often sufficient to detect duplicates error reports manually or by Natural Language Processing. We propose an approach that uses execution traces.

For figure 1, the Jacard similarity measure resulted in a similarity measure of 44% while the Cosine similarity measure resulted in a measure of 95%. With the implementation of SEQUITUR, which uses sequential information, a more precise similarity measure should be achieved.

Judith Providence
\( A := \{ e_i | e_i \in \sigma_A, \ 1 \leq i \leq n \} \)

\( B := \{ e_i | e_i \in \sigma_B, \ 1 \leq i \leq m \} \)

The similarity of a pair of execution traces is calculated by using the Cosine similarity formula \(^{\text{(12)}}\) \(^{\text{(4)}}\). Execution traces \( \sigma_A \) and \( \sigma_B \) are transformed into vectors. For two vectors \( \sigma_A = < w_{A1}, w_{A2}, \cdots, w_{Ac} > \) and \( \sigma_B = < w_{B1}, w_{B2}, \cdots, w_{Bc} > \), where \( \sigma_A, \sigma_B \in \mathbb{N}^{|E|} \), where \( c = |E| \), the similarity of two error reports is defined by equation (2).

\[
\text{Sim}(\sigma_A, \sigma_B) = \frac{\sum_{i=1}^{N} w_{Ai}w_{Bi}}{\sqrt{\sum_{i=1}^{N} w_{Ai}^2 \sum_{i=1}^{N} w_{Bi}^2}} (2)
\]

Where \( w_{Ai} \) and \( w_{Bi} \) are defined as how often \( e_i \in E \) is present in \( \sigma_A \), respectively \( \sigma_B \).

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 \(^{\text{(12)}}\). 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 \( v_i (1 \leq i \leq n) \) is the weight of the \( i_{th} \) index term in the vector \( < v_1, v_2, \cdots, v_n > \).

\[
v_i = tf_i \times idf_i \quad (3)
\]

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 (4).

\[
idf_i = \log \frac{Dsum}{Dv_i} \quad (4)
\]

Where \( Dsum \) is the total number of documents and \( Dv_i \) 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 \( v_i \) was modified:

\[
w_i = 2 \times \log_2 (w_i \text{ of } e_i) \quad (5)
\]

### 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. Sixty-two execution traces were created from various error reports in the 2004 Eclipse bug repository. The sixty-two error reports included seventeen duplicate pairs. Method calls 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 used for several reasons. It is an established way of determining similarities in error reports \(^{\text{(12)}}\). Method calls provide a reasonable middle ground to recognize similar usages of functionality. 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. The goal is to determine if an appropriate similarity measure can be applied to execution traces with the result of matching each execution trace its duplicate.

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

- is a particular method called or not (binary)
- how often is each method called (integer)
- in what order are methods called (sequential)

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 several distance metrics. These distance metrics were evaluated based on two requirements. The similarity measure must apply to traces of different lengths. The measure must clearly separate traces that are not from duplicate error reports. Metrics such as the Euclidean distance metric, the Manhattan distance metric and the Minkowski distance metrics, often used in similarity measures were researched\(^7\) \(^{11}\). These metrics numerically describe how far apart two objects are to each other. As our research progressed, we decided to use the Jacard similarity coefficient and the Cosine similarity measure. The advantage of using the Jacard Index is that it can be applied to categorical data. The data attributes depend on the presence or absence of a certain property\(^{(10)}\). Our dataset was converted to binary vectors in which the data attributes depended on the presence or absence of a method in the error report. The advantage of using the Cosine similarity measure is that it is an established measure of similarity. It takes in more information from the traces.

**An Example**

We used Traviando\(^{(5)}\), a trace analyzer that runs in the Eclipse IDE environment. Calculations based on the number of events were used in the Jacard similarity measure. Each method call was labeled as an event. For the Jacard distance metric, comparisons were based on method calls(events) from two execution traces.

In figure 1 and figure 2, we see that the bulk of all duplicate error reports are within the 75th percentile. In figure 1, overlap-
ping of duplicate and non-duplicate error reports occur at about 43%. What is needed are similarity measures that show a more clear distinction between duplicate and non-duplicate reports. A clear distinction of similar(duplicate) and dissimilar(non-duplicate) error reports would allow us to establish a basis for the detection of errors in software. Therefore, our next step used the Cosine similarity measure. Figure 2, shows the results of the Cosine similarity measure applied to the same traces. This technique gives a more exact matching of the data set. From observing the data set, we see that all similar traces have a similarity above 91%. However, there is still a overlap of similar and dissimilar traces from around 91% to 98%.

On Going Work
Currently, we are developing a more precise similarity measure for the dataset. We are delving into defining similarity by the sequence of method calls that exist in each trace rather than the binary(Jacard) or integer(Cosine) features of the dataset. We currently implement SEQUITUR(8). SEQUITUR is a algorithm with a promising concept. It provides for a one-pass on-the-fly computation of identical subsequences that is promising for long execution traces. For better calibration of the trace data, we are evaluating the effectiveness of error location. This involves a weighing scheme that places more value on the location of the error. If the errors are documented as occurring at the end of the trace, the weighing scheme will adjust to place a heavier emphasis on that location. We are also looking into class level granularity of trace data.

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
In this paper, sixty-two traces were generated from the Eclipse IDE bug repository. We described two similarity measures for the determination of similarity in execution traces. The Jacard and the Cosine similarity measure were evaluated. We see that the Cosine similarity measure performs better than the Jacard Similarity measure. However, there is still a need to investigate how well the similarity measure performs when used in conjunction to the Sequitur Algorithm.

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

References


