Computing and Informatics, Vol. 35, 2016, 1110–1140

EARLY FAILURE PREDICTION IN SOFTWARE PROGRAMS: DIMENSIONALITY REDUCTION KERNEL

Somaye Arabi Naree

Department of Computer Science
Faculty of Mathematical Sciences and Computer, Kharazmi University
Tehran, Iran
e-mail: `s.arabi@khu.ac.ir`

Saeed Parsa

Department of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
e-mail: `parsa@iust.ac.ir`

Abstract. The aim of this paper is to build an online failure prediction classifier for monitoring the behavior of programs. The classifier predicts the termination state of the program execution paths as failing or passing. This could be achieved by mapping each execution path as a vector into a feature space whose dimensions represent common sub-paths amongst failing and passing execution paths. The main contribution of this paper is to treat the failure prediction problem as a classification task of execution paths in a customized feature space. The main dilemma is the size and the number of space dimensions, affecting the speed of the classifier. The size of the dimensions could be reduced by shortening the length of the common sub-paths, used as the space dimensions. The length of common sub-paths is affected by repeated patterns in program executions. Replacing the consecutively repeated patterns with only a single iteration in execution paths, reduces the size of the common sub-paths. The number of dimensions could be reduced by removing dimensions which have projection onto others. This paper proposes two kernels which measure similarity amongst execution paths in an implicit feature space with reduced dimensionality. Our experiments demonstrate a significant reduction in time overhead of the failure prediction classifier while preserving accuracy.

Keywords: Online failure prediction, kernel classifier, dimensionality reduction, feature space, execution path

1 INTRODUCTION

Failure prediction is of great concern in safety critical software systems. To predict upcoming failures, program execution states are monitored by applying failure prediction models. Prediction models are embedded within the program body immediately after certain decision making expressions which affect program execution paths. These expressions are called predicates [1, 2]. In this regard, the constructed prediction models should be accurate and fast enough to avoid intolerable overhead on the program execution time. To achieve this, we have proposed a domain specific SVM classifier, separating failing and passing execution paths in a program execution space [3]. The execution space is built as a Cartesian space, in which each dimension represents the number of appearance of a certain predicate within the program execution path. The main problem is that the execution paths are not linearly separable in the execution space. Therefore, the execution paths have to be mapped into a new feature space. It is shown that the common sub-paths traversed by different executions of a program are appropriate features to construct a feature space in which passing and failing executions of the program could be classified with a maximum possible margin [3]. Therefore we apply a feature expansion technique to map the execution space into a feature space in which dimensions represent common sub-paths between each pair of the execution paths. To classify execution paths in the feature space, the SVM classifier could apply a kernel function. Kernel function enables SVM classifier to classify execution paths in a high-dimensional, implicit feature space without ever computing the coordinates of execution paths in that space, but rather by simply computing the inner products between the images of all pairs of execution paths in the feature space. The inner products between the images of all pairs of execution paths in the feature space could be computed by a similarity measurement between the execution paths. To classify an execution path as failing or passing its similarity with the existing execution paths is measured using our customized kernel function, All Common Sub-paths (ACS).

The dimensionality of the feature space apparently affects the time required for measuring the similarities. Therefore, a major dilemma concerning the performance of a classifier is to minimize the dimensionality of its feature space [4]. To minimize the dimensionality, the size and the number of the feature space dimensions should be minimized.

In order to reduce the number of dimensions, the dimensions should be selected in such a way that they cannot be projected on each other. In other words the selected dimensions should be independent of each other. To this end, the longest disjoint common sub-paths amongst execution paths are considered as the dimensions of the feature space [4, 5]. The resultant kernel is named All Disjoint Longest Common Sub-paths (ADLCS) kernel.

The sizes of the dimensions are dependent on the length of the common sub-paths, used as the space dimensions. The size of a common sub-path, represented as a sequence of program predicates, is greatly affected by iterative executions of the predicates. In this article, it is shown that the identical sub-paths of predicates in loop iterations or function calls are the most important factor of increasing the size of the feature space dimensions and subsequently slowing down the performance of the kernel in measuring the similarities amongst execution paths. This article proposes two new similarity based kernels, called Extended ACS (EACS) and Extended ADLCS (EADLCS), which reduce the time overhead caused by the iterative loops or function calls. To achieve this, the proposed extended kernels measure similarities in a feature space for which the sub-paths related to the dimensions are shortened. The sub-paths are shortened by replacing consecutively identical predicate subsequences with a single representative subsequence. Applying these kernels to reduce the dimensionality of the feature space, our experiments demonstrate significant reduction in time overhead of failure predictions while preserving the accuracy.

Our proposed technique for reducing the dimensionality of feature space is also of great use to build relatively high speed string classification kernels. The speed of string kernels is of great importance in online classifications used in many time critical applications such as automotive assistant systems, remote medical devices, disaster prediction and remote monitoring of flight control software systems. In such applications the main goal is to accelerate the classification of successive streams of input data to early predict the upcoming failures [6].

The remaining parts of this paper are organized as follows. In Section 2, we discuss related works on failure prediction. An overview of the method including some definitions and proposed kernels for online failure prediction is described in Section 3. In Section 4 the advantages of our method over existing failure prediction methods are demonstrated through experiments, and Section 5 concludes the paper.

2 RELATED WORKS

Most of the failure prediction methods are monitoring based methods meaning that they monitor the current state of the running system to predict the upcoming failures [7]. The monitoring based methods can be divided into three categories: observation of failures, monitoring of symptoms, detection of errors [8, 9, 10, 11].

Observation of failures: The basic idea of failure prediction based on previously observed failures is to estimate the probability distribution of future failure event. The probability distribution of future failures can be estimated by formal approaches such as Bayesian classifiers or rather heuristic approaches such as counting and thresholding.

Monitoring of symptoms: This approach is appropriate for some type of faults affecting the system gradually, which are also known as service degradation. A well-known example for such types of faults are memory leaks. The basic idea of failure prediction based on monitoring of symptoms is that faults like memory leaks can be detected by their side-effects on the system such as exceptional memory usage, CPU load, or disk I/O. These side-effects are called symptoms.

Detection of errors: The basic idea of this group of failure prediction methods is to analyze the occurrences of previous error events in order to evaluate the current state with regard to upcoming failures. One of the successful approaches which are applied in this category is a pattern recognition. The main idea is to analyze faulty sequences of system events to form faulty patterns. Then a function is applied to measure similarities between an observed sequence of events with learned faulty patterns. Failure prediction is then performed by classification based on pattern similarity measures as depicted in Figure 1.

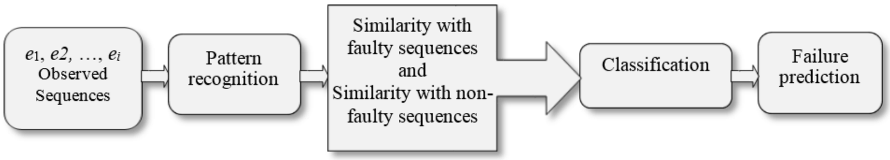

e_1, $e2$, …, e_i Observed Sequences
Pattern recognition
Similarity with faulty sequences and Similarity with non-faulty sequences
Classification
Failure prediction

Figure 1. Failure prediction on observed sequences according to similarity measure with failure patterns

Similarity amongst event sequences could be measured by applying an alignment algorithm. However, to our knowledge, no failure prediction approaches applying pair wise alignment algorithms were published. The proposed approach in this paper uses the pairwise alignment algorithm in the kernel function of an SVM classifier to distinguish failing and passing execution sequences of a program.

In the remaining parts of this section some well-known online failure detection methods in the category of detection of errors are analyzed.

Failure prediction using Markov models: The Markov chains have been used as program behavioral models to perform anomaly detection [12, 13]. Each state in the Markov chain represents a number of distinct observations, program predicate values, in different successful training executions at a specific instrumented location of the program. Since all the program states may not be observed at the training phase, an anomalous state is added to the Markov chain. This state hypothetically contains all predicate values not observed at the training stage. After the Markov chain is built, some thresholds are chosen for each transition. These thresholds are further applied at runtime to determine the anomalous transitions.

Since some faults are manifested in long-range dependencies among the program states, a model which does not consider such dependencies may fail to detect some specific faults in the program. Using a k-order Markov chain, it is not possible to predict bugs relevant to more than k consecutive predicates. This is due to the fact that in the k-order Markov chain the probability of each state is independent of the $k + 1$ past states which are connected, but not directly, to the state. The time overhead imposed by this model to the program execution time is extremely high. The main reason is that it takes a long time to compare all the values of the predicates in an execution path with their corresponding predicates in Markov states.

Dynamic invariant detection: Dynamic invariant detection methods [14, 15, 16] run a program and infer those properties that are true over the observed passing executions. These properties are called invariants. DIDUCE [12] is an online anomaly detection system which uses dynamic invariant detection. Anomalies could be detected through any violation of the program invariants. However, dynamic invariant detectors do not perform any analysis on the program source code to check the correctness of the inferred invariants. Therefore a large complete test suite is required to ensure the correctness of invariants.

Online bug detection using Extended Finite State Automaton: Argus [17] constructs runtime statistics on a sliding window over each program execution. Runtime statistics are used to build an extended finite-state-automaton (ext-FSA) to monitor the programs runtime control-flow behavior. In the ext-FSA, each state represents a runtime event, and transitions indicate the order of runtime events. Each transition is labeled with the distribution of transition frequency in different training executions. Argus assumes the anomalous transitions in test executions imply faulty behavior of the program. As Markov models, FSA suffers from finding faults which are related to a sequence of state transitions. Argus imposes high run-time overhead due to the excessive number of states which increases the search space.

Statistical techniques: Statistical approaches estimate parametric or non parametric models for each program execution state from the data gathered of program executions in different input test data. Then a statistical test on the probability of the program execution states applies to be generated by the estimated model. The generated probabilities of the program execution states could be used to assign an anomaly score to the test executions of program in the real environment [7, 18].

3 OVERVIEW OF THE METHOD

In this section the main idea of the proposed failure prediction method is described in detail. To this end, first some basic definitions are depicted in Section 3.1 and the detail of the proposed method is indicated in Section 3.2.

3.1 Problem Statements

The problem is early prediction of a program termination state based on the similarity of the program execution path with training failing and passing execution paths. Execution paths ending to each predicate are classified as failing (i.e. abnormal) or passing (i.e. normal). The normal paths are separated from failing ones within the program execution space through the application of a customized SVM classifier.

Let us assume that $P = \{p_1, p_2, \ldots, p_N\}$ is a set of predicates belonging to program R and $E_i = \{e_{i1}, e_{i2}, \ldots, e_{im}\}, 1 \leq i \leq N$ is the set of all execution paths ending to predicate $p_i \in P$ in m different executions of R.

Definition 1 (Execution Path)**.** An execution path $e_{ij} = \langle p_1, \ldots, p_i \rangle \in E_i$, $1 \leq j \leq m$, is a sequence of the predicates executed up to the predicate $p_i \in P$ at the j^{th} execution of the program R.

Definition 2 (Training Execution Path)**.** A training execution path is represented as a sequence $te_{ij} = \langle e_{ij}, AR \rangle$ where $AR \in \{-1, 1\}$ indicates the actual result of the program termination state in execution path e_{ij} while -1 and 1 represent failing and passing program termination states, respectively. The set $TE_i = (TE_{ipass} \cup TE_{ifail}) = \{te_{i1}, te_{i2}, \ldots, te_{im}\}$ represents all the training execution paths which are applied to build the failure prediction classifier for the predicate p_i.

Definition 3 (Failure Prediction Classifier)**.** A failure prediction classifier is a function $C_i : e_{ij} \to PR$, where PR indicates the predicted program termination state resulted from execution path e_{ij}. That is, for each execution path $e_{ij} \in P^*$, function C_i predicts the termination result of e_{ij} as $C_i(e_{ij}) = PR \in \{-1, 1\}$, where -1 and 1 indicate failing and passing termination states, respectively. To build a failure prediction classifier, a similarity measurement function Sim is required as the core of the classifier to measure the similarities between each pair of failing and passing execution paths.

Definition 4 (Fault Suspicious Path)**.** An execution path e_{ij} is classified as a fault suspicious path provided that e_{ij} is more similar to the unsuccessful paths than the successful ones. To be more precise, an execution path e_{ij} is reported as fault suspicious if the conditions in Relation (1) are satisfied:

$$\text{Sim}(e_{ij}, TE_{(ipass)}) < \text{Passing-threshold and Sim}(e_{ij}, TE_{(ifail)}) > \text{Failing-threshold.} \tag{1}$$

In Relation (1), the function Sim measures the similarity amongst e_{ij} and execution paths in $TE_{(i)}$. We further apply Sim as the kernel function of SVM classifier, described in Section 3.2.

Definition 5 (Cost of Failure Prediction)**.** Let $e_{ij} = \langle p_1, \ldots, p_i \rangle$ be an execution path then execution sub-path $e_{ij}[1, i'] = \langle p_1, \ldots, p_{i'} \rangle$ $(1 \leq i' \leq i)$ is a prefix of e_{ij}. The failure prediction classifier C_i is built in a way that for any execution path $e_{ij} = \langle p_1, \ldots, p_i \rangle$, there exists a positive integer i' such that $C_i(e_{ij}[1, i']) = C_i(e_{ij}[1, i' + 1]) = C_i(e_{ij}[1, i' + k]) = C_i(e_{ij})$ $(k \geq 0)$. In other words, C_i predicts the termination state of the execution path e_{ij} based on only the prefix $e_{ij}[1, i']$. The length of the prefix that C_i checks in order to make the failure prediction is called the cost of the failure prediction, denoted by $\text{Cost}(C_i, e_{ij}) = i'$. Clearly, $\text{Cost}(C_i, e_{ij}[1, i']) = \text{Cost}(e_{ij}[C_i, 1, i' + 1]) = \text{Cost}(C_i, e_{ij}[1, i' + k])$ for $k \geq 0$ and trivially $\text{Cost}(C_i, e_ij) \leq$

$\|e_{ij}\|$ for any execution path e_{ij}. Given a set of training execution paths TE for the program under test, the cost of the failure prediction is defined as in Relation (2):

$$\text{Cost}(C, TE) = \frac{\sum_{\langle e,AR\rangle \in TE} Cost(C,e)}{\|TE\|} \tag{2}$$

and the accuracy of the failure prediction classifier C is denoted as in Relation (3):

$$\text{Accuracy}(C, TE) = \frac{\|\{e|C(e) = PR \wedge (e, PR) \in TE\}\|}{\|TE\|}. \tag{3}$$

Generally, to improve the efficiency of early prediction of program failures it is needed to reduce the prediction cost while preserving the prediction accuracy at a satisfactory level.

3.2 Proposed Method

In this paper the aim is to classify execution paths, represented as a sequence of predicate values, as two classes of failing or passing. The raw execution paths are not linearly separable so we need to map paths into a feature space. Using kernel trick, instead of explicitly mapping each execution path in the feature space, we apply kernel functions to measure the similarities between each pair of execution paths. We have proposed two similarity functions (kernel functions) over pairs of execution paths. Proposed similarity functions enables SVM classifier to classify paths in a high dimensional space, implicit feature space without ever computing the coordinates of the mapped paths in that space, but rather by simply computing the inner products between the images of all pairs of paths in the feature space. Let A and B be two execution paths and A' and B' be the images of A and B in the feature space respectively, then the similarity function should be defined in such a way that Similarity$(A, B) = \varphi(A).\varphi(B) = A'.B'$.

In this section two algorithms describing our similarity based kernels, EACS and EADLCS, are presented. These kernels map execution paths into an implicit feature space F^N in such a way that the distance of each pair of the mapped paths reflects their similarity measure. The main decision to be made is to determine the similarity notions or features of the paths to be applied in the mappings. We have selected each sub-path s commonly appearing in two or more of the execution paths as a distinct similarity feature. Each axis of the feature space F^N corresponds to a single similarity feature s. The mapping function $\varphi(e)$, described in Relation (4), maps each execution path e to a point e' in F^N. Each dimension of e' indicates the number of occurrences of the sub-path s corresponding to that dimension in e.

$$\varphi : e \to e' = \varphi(e) \in F^N \tag{4}$$

where

$$\varphi(e) = (f(s_1, e), f(s_2, e), \ldots, f(s_N, e)) \in F^N$$
$$f(s_i, e) = |\{(s_1, s_2) : e = s_1 s_i s_2\}|, s_i \in P^* \text{ and } 1 \leq i \leq N$$

In the above relation each dimension of e' is computed by the function $f(s_i, e)$, as the number of occurrences of the sub-path s_i in e.

The speed of the kernel classifier is dependent on the number and the size of the sub-paths, selected as the dimension of the feature space. In the following subsections, we propose the ACS and its extended versions EACS and EADLCS kernels. We discuss the benefits and applications of each of the EACS and EADLCS kernels.

3.2.1 ACS-Kernel

In this section a new sequence matching algorithm, ACS-Kernel, to measure the similarity between program execution paths is presented. ACS-Kernel measures similarity between each pair of execution paths in an implicit feature space where dimensions are all common sub-paths amongst execution paths. The algorithm measures the similarity between any two given paths S and T as the sum of products of $|L_S(v)|$ and $|L_T(v)|$, representing the number of occurrences of sub-path v in S and T, respectively. The ACS algorithm consists of two parts named ACS-Kernel and ProcessNode. The ACS-Kernel function begins with identifying all sub-paths of its inputs S and T and keeping them in $L_S(\phi)$ and $L_T(\phi)$, respectively. Then, ACS-Kernel invokes the ProcessNode function to compute all the sub-paths of S and T, including common subsequence of predicates.

Algorithm ACS-Kernel
Input: Execution paths S and T, **Output:** Similarity measure between S and T

```
ACS-Kernel ($S, T$)
Begin
    Let $L_S(\phi) = \{(S(i, |S|), 0) : i = 1 : |S|\}$
    Let $L_T(\phi) = (T(i, |T|), 0) : i = 1 : |T|$
    Similarity = 0;
    Call ProcessNode($\phi, 0$)
End
ProcessNode ($v$, depth)
Begin
    if (depth != 0)
        /* $|L_S(v)|$ is the number of supaths of $S$ which begin with $v$ and $|L_T(v)|$ is the
        number of supaths of $T$ which begin with $v$ */
        Similarity += $|L_S(v)| \times |L_T(v)|$
    if (depth > $|S|$ or depth > $|T|$)
```

```
        return;
    else if ($L_S(v)$ and $L_T(v)$ both are not empty)
    Begin
        while (There exists $(u, i)$ in the list $L_S(v)$)
            Add $(u, i+1)$ to the list $L_S(vu_{(i+1)})$;
        while (There exists $(u, i)$ in the lists $L_T(v)$)
            Add $(u, i+1)$ to the list $L_T(vu_{(i+1)})$;
        for (each $p \in P$ ) // $P$ is the set of all predicates in the program
            Call ProcessNode $(vp, \text{depth}+1)$
     End
End
```

In the ACS algorithm, $S(i, j)$ is a sub-path of S which starts at predicate number i and ends up with the predicate number j. ProcessNode(v, depth) computes the number of sub-paths of S and T which include v and compute the value of the Similarity variable.

The ACS kernel algorithm measures the similarity between each pair of execution paths in a feature space in which each dimension represents a common sub-path. The similarity measures are dependent on the length of the compared paths. To normalize the similarity measures to range $[0, 1]$, the following relation is used:

$$\text{NormalizedKernel}\,(S, T) = \frac{\text{Kernel}(S, T)}{\sqrt{\text{Kernel}(S, S) \times \text{Kernel}(T, T)}} \tag{5}$$

A. Computation cost of the ACS-Kernel algorithm

The order of ACS-Kernel to measure similarity between each pair of execution paths, S and T, is $O(|S|^2 + |T|^2)$. ACS-Kernel is applied to measure the similarities between all possible pairs of execution paths $(S, T) \in TE$ where TE indicates the set of execution paths used to train the SVM classifiers. The similarity measure between each pair of execution paths (S^i, T^j) is kept in the entry $[i, j]$ of the kernel matrix M. Therefore the order of computation cost for each row i of M is $O(m|S^i|^2 + \sum_{j=1}^{m} |T^j|^2)$ where m indicates the number of the paths $S^i \in TE$ and $|T^j|$ indicates the length of the execution path T^j. In order to reduce the computation cost, firstly a record of all sub-paths of S^i are computed and kept in a Trie structure. This requires $O(|S^i|^2)$ time. Then the similarity between S^i and the other paths represented by the columns of the matrix is computed. In this way we traverse and process path S^i only once. Therefore, the time complexity for computing each row of the kernel matrix is reduced to $O(|S^i|^2 + \sum_{j=1}^{m} |T^j|^2)$.

B. Benefits of the proposed ACS-Kernel

In this section it is shown that ACS-Kernel can be efficiently applied to predict software failures. There are three reasons for this assertion.

Semi positive definite: The first reason is that ACS-Kernel is semi positive definite. A classifier which uses a semi positive definite kernel is guaranteed to find the global optimum solution instead of local optimum [4, 6]. A kernel such as ACS-Kernel is semi positive definite if it satisfies Relation (6) for each pair of paths e_i and e_j.

$$\begin{gathered}\text{ACS-Kernel}(e_i, e_j) = \langle \varphi(e_i), \varphi(e_j) \rangle, \\ \varphi : e \to \varphi(e) = (f(s_1, e), f(s_2, e), \ldots, (s_N, e)) \in F^N, \ s_i \in P^*.\end{gathered} \tag{6}$$

If N is the number of all possible subsequences of the program predicates, each path e is mapped into an N dimensional feature space F^N where each dimension is a subsequence of program predicates which is shared between at least two execution paths of the program. Obviously, N can be very large and it takes a long time to map all the program execution paths. Computing ACS-Kernel does not involve evaluation of the mapped vectors $\varphi(e)$, instead it is computed directly according to the path vectors e without any direct mapping.

It can be shown that ACS-Kernel is semi positive definite because the results of the kernel for two path vectors e_i and e_j are the same as the dot product of corresponding mapped vectors in the feature space as shown in Relation (7).

$$\text{ACS-Kernel}(e_i, e_j) = \langle \varphi(e_i), \varphi(e_j) \rangle = \sum_{s \in P^*} \varphi_s(e_i) \varphi_s(e_j). \tag{7}$$

Polynomial time complexity: Minimal time overhead to map execution path vectors and measure the similarities in the feature space is another desired feature of the ACS-Kernel. In general the time required for mapping execution paths vectors into a feature space is of an exponential order [4, 6]. As described above, the time required by the ACS-Kernel for mapping execution paths and computing similarities is of a polynomial order.

Customized kernel function: To predict termination state of a program execution at each program predicate, a dedicated classifier separating failing and passing executions of the program is required. To build a dedicated classifier a kernel function measuring similarities among program execution paths, represented as sequences of predicates, is required. Existing SVM kernel functions could only measure the similarities amongst execution paths based on predicates as independent features of program executions. They cannot be easily applied to measure the similarity amongst sequences of inter-related predicates representing a program execution path. The ACS-Kernel function outperforms the existing ones in the sense that it can simply measure the similarities among program execution paths in terms of their sub-paths.

3.2.2 EACS-Kernel

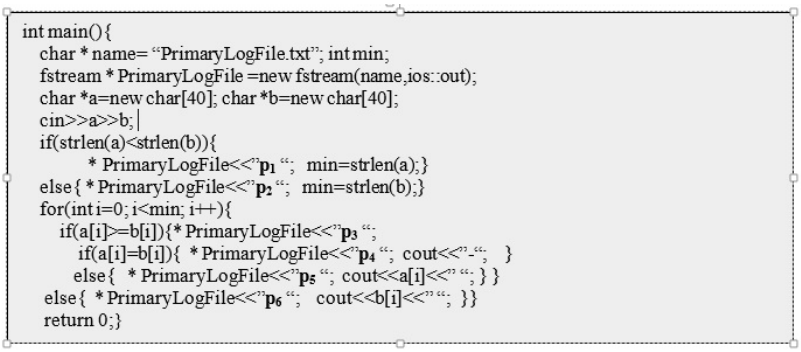
The feature space dimensions of ACS-Kernel are all common sub-paths appearing in execution paths. But there are many repeated subsequences of predicates in these common sub-paths because the existence of iterations in the program code. It is a time consuming task to measure the similarities among execution paths when there are consecutive repeated patterns of predicate subsequences in the executions. The difficulty is when these repeated patterns begin or terminate with the same subsequence of predicates. To have a sense of difficulty, imagine two execution paths $s = \langle p_1, p_2, p_3, p_2, p_3, p_4, p_5\rangle$ and $t = \langle p_1, p_2, p_3, p_2, p_3, p_2, p_3, p_4, p_6\rangle$. The sub-path p_2, p_3 appears consecutively twice in the execution path s and three times in t while the first and last iterations of p_2, p_3 begin and end up with the same predicates, p_1 and p_4, in both s and t.

To resolve the difficulty we have developed the EACS-Kernel which uses the ACS-Kernel but reduces its time overhead. Our studies show that there is no efficient solution to alleviate the difficulty in the previous approaches to failure prediction. Some failure prediction approaches control such iterative patterns in a random manner [18]. For example they consider only n consecutive iterations of repeated sub-paths where n is selected randomly. It is obvious that this random approach considerably decreases the model precision.

In this section, an extended version of ACS-Kernel, called EACS, is proposed to measure the similarity between pairs of execution paths including only a single iteration of each consecutively repeated sub-path along with its number of iterations. It should be noticed that replacing each consecutively repeated subsequence with a single iteration of the subsequence along with its number of iteration results in no loss of accuracy. To have a better understanding of our proposed approach let us consider the instrumented program in Figure 2 a). As shown in Figure 2 b), the program execution path is shortened by replacing each of its consecutively repeated predicate subsequences with a single representative subsequence.

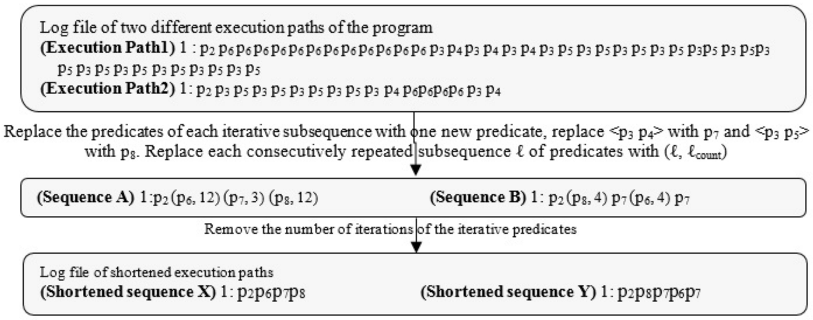
```
**Algorithm EACS-Kernel**
Input: Two execution paths $A$ and $B$.
Output: Similarity measure, $S_{AB}$, between $A$ and $B$.
Step0. Shorten $A$ and $B$ into $X$ and $Y$, respectively.
Step1. $S_{xy} = \mathit{ACS\text{-}kernel}(X, Y)$
Step2. // Traverse sequence $A$ from left to right
    For each subsequence, $\ell$, repeated $\ell_{count}$ times ($\ell_{count} > 1$) in $A$ do
    Begin
        Step 2.1.  $\mathit{Inc1} += (\ell_{count} - 1) \times \ell_{Y-\text{appear}}$ let $\ell_{Y-\text{appear}}$ = total no. of $\ell$ in $Y$
        Step 2.2.
        For $i = 1$ to $\ell_{Y-\text{appear}}$ do
            // find identical windows containing $\ell$ in both $X$ and $Y$
        Begin
            $L_i$ = NextCommonConsecutiveIteration($\ell$) in $X$ and $Y$
            Head = Subsequence $H$ appearing immediately before $L_i$ in both $X$ and $Y$.
            Tail = Subsequence $T$ appearing immediately after $L_i$ in both $X$ and $Y$.
            $w_i$ = Head + $L_i$ + Tail and $w_{i\ell}$ = Index of $L_i$ in $w_i$
            $\mathit{Red1} += \sum_{i=1}^{\ell\text{Y}-\text{appear}}((\text{len}(w_i) - w_{i\ell}) \times (w_{i\ell} - 1))$
        End For
        Step 2.3. Replace $\ell$ in $X$ with $(\ell, \ell_{count})$.
    End For
Step3.  // Traverse sequence $B$ from left to right
    For each subsequence, $\ell$, repeated $\ell_{count}$ times in $B$ do
    Begin
        Step 3.1.
            $\ell_{A-\text{appear}}$ = total number of appearances of $(\ell, \ell_{icount})$ in $A$.
            $\ell_{icount}$ = number of iterations of $\ell$ in the $i^{\text{th}}$ appearance of $(\ell, \ell_{icount})$ in $A$.
            $\mathit{Inc1} += (\ell_{count} - 1) \sum_{i=1}^{\ell\text{A}-\text{appear}} (\ell_{icount} \times (\ell_{icount} + 1)/2)$
        Step 3.2.
        For $i = 1$ to $\ell_{A-\text{appear}}$ do
        Begin
            If $\ell_{icount} >= 3$ then $\mathit{Red2} += (\ell_{icount} - 2) \times (\ell_{icount} - 1) \times \frac{\frac{(2\times\ell_{icount}-3)}{3}+1}{4}$
            If $\ell_{icount} - 2 >= \ell_{count}$ then
            $\mathit{Red2} -= (\ell_{icount} - \ell_{\text{count}} - 1) \times (\ell_{icount} - \ell_{\text{count}}) \times \frac{\frac{(2\times\ell_{icount}-2\times\ell_{\text{count}}-1)}{3}+1}{4}$
            $L_i$ = NextCommonConsecutiveIteration($\ell$) in $Y$ and $A$
            Head = Subsequence $H$ appearing immediately before $L_i$ in both $Y$ and $A$.
            Tail = Subsequence $T$ appearing immediately after $L_i$ in both $Y$ and $A$.
            $w_i$ = Head + $L_i$ + Tail and $w_{i\ell}$ = Index of $L_i$ in $w_i$
            $\mathit{Red1} += \sum_{i=1}^{\ell\text{A}-\text{appear}} ((\text{len}(w_i) - w_{i\ell}) \times (w_{i\ell} - 1))$
        End For
        Step 3.3. Replace $\ell$ in $Y$ with $(\ell, \ell_{icount})$.
        $\mathit{Inc2} = \sum_{i=1}^{2\times\ell A-\text{appear}} ((len(w_i) - w_{i\ell}) \times (w_{i\ell} - 1))$
            $+ (\text{len}(w_i) - w_{i\ell} - \ell_{icount} + 1) \times (\ell_{icount} - 1)$
    End For
Step4. Evaluate the overall similarity between $A$ and $B$: $S_{AB} = S_{XY} + \mathit{Inc1} + \mathit{Inc2} - \mathit{Red1} - \mathit{Red2}$
```

EACS-Kernel measures similarity between main execution paths A and B according to the similarity between shortened execution paths X and Y. Each consecutively repeated subsequence ℓ of predicates in A or B is represented by (ℓ, ℓ_{count}), where ℓ count indicates the number of iterations of the subsequence ℓ within A or B. The EACS-Kernel calls ACS-Kernel to measure similarity between shortened execution paths X and Y. Then EACS-Kernel applies a windowing technique to find identical iterative patterns in A and B and modifies the computed similarity to the similarity between the main execution paths A and B. Step 0 in the algorithm shortens A and B into X and Y by replacing each (ℓ, ℓ_{count}) with ℓ. Step 1 calls ACS-Kernel to compute similarity between shortened paths X and Y. Step 2



```
int main(){
   char * name= "PrimaryLogFile.txt"; int min;
   fstream * PrimaryLogFile =new fstream(name,ios::out);
   char *a=new char[40]; char *b=new char[40];
   cin>>a>>b;
   if(strlen(a)<strlen(b)){
         * PrimaryLogFile<<"p1 "; min=strlen(a);}
   else{ * PrimaryLogFile<<"p2 "; min=strlen(b);}
   for(int i=0; i<min; i++){
      if(a[i]>=b[i]){* PrimaryLogFile<<"p3 ";
         if(a[i]=b[i]){ * PrimaryLogFile<<"p4 "; cout<<"-";  }
         else{  * PrimaryLogFile<<"p5 "; cout<<a[i]<<" "; } }
    else{ * PrimaryLogFile<<"p6 ";  cout<<b[i]<<" "; }}
    return 0;}
```

a)


Log file of two different execution paths of the program
(Execution Path1) 1 : p_2 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_6 p_3 p_4 p_3 p_4 p_3 p_4 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5
(Execution Path2) 1: p_2 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_5 p_3 p_4 p_6 p_6 p_6 p_6 p_3 p_4
Replace the predicates of each iterative subsequence with one new predicate, replace <p_3 p_4> with p_7 and <p_3 p_5> with p_8. Replace each consecutively repeated subsequence ℓ of predicates with (ℓ, ℓ_{count})
(Sequence A) 1:p_2 (p_6, 12) (p_7, 3) (p_8, 12)
(Sequence B) 1: p_2 (p_8, 4) p_7 (p_6, 4) p_7
Remove the number of iterations of the iterative predicates
Log file of shortened execution paths
(Shortened sequence X) 1: $p_2p_6p_7p_8$
(Shortened sequence Y) 1: $p_2p_8p_7p_6p_7$

b)

Figure 2. An example of shortening the length of execution paths: a) An instrumented code, b) Building shortened execution paths

and Step 3 in the algorithm modify the similarity considering the repeated subsequences ℓ of predicates in the main paths A and B. Step 2 in the algorithm finds identical windows containing ℓ in both A and Y. Step 3 in the algorithm finds identical windows containing ℓ in both B and X. In the algorithm, w means window and len(p) represents the length of sequence p. To show how EACS-Kernel algorithm works, let us consider the example in Figure 3.

A. Computation cost of the EACS-Kernel algorithm

The computation cost of EACS-Kernel algorithm is computed as follows:

Step 1: Cost of step 1 is equal to the computation cost of the ACS-Kernel function on shortened paths. The cost of this step is $O(|X|^2 + |Y|^2)$.

Step 2: Computing *Inc1* in step 2.1 requires finding the locations in the path Y where repeated subsequence ℓ is appeared. Therefore the cost of this step is $O(\ell_{Y-\text{appear}})$.

Input: A = <p_1 p_2 p_3,4 p_4 p_5 p_3,2 p_6> B = <p_8 p_2 p_3,3 p_4 p_5 p_3,2 p_7>

Step0: X = < p_1 p_2 p_3 p_4 p_5 p_3 p_6> Y = < p_8 p_2 p_3 p_4 p_5 p_3 p_7>

Step1: $S_{X,Y} = ACS - Kernel(X,\ Y) =\ 17$

Step2:Traverse A from left to right
First iteration: ℓ=p_3, ℓ_{count}=4
Step 2.1: p3 is appeared 2 times in Y therefore: $\ell_{Y-appear}$=2 , Inc1 = 6
Step 2.2: for each appearance of ℓ=p_3 in Y find similarity windows with X:
w_1 = < p_2 p_3 p_4 p_5 p_3 > , $w_{1\ell}$= 2 , $L_{W(1)}$ = 5 , Red1 = 3 , w_2= -
Step 2.3: X = < p_1 p_2 p_3,4 p_4 p_5 p_3 p_6>
Second iteration: ℓ=p_3 , ℓ_{count}=2
Step 2.1: p3 is appeared 2 times in Y therefore:$\ell_{Y-appear}$=2 , Inc1 = 8
Step 2.2: for each appearance of ℓ=p_3 in Y find similarity windows with X:
w_1= - , w_2= < p_3 p_4 p_5 p_3> , $w_{2\ell}$ = 4 , $L_{W(2)}$ = 4 , Red1 = 3
Step 2.3: X = < p_1 p_2 p_3,4 p_4 p_5 p_3,2 p_6>

Step3: Traverse B from left to right
First iteration: ℓ=p_3 , ℓ_{count}=3
Step 3.1: p3 is appeared 2 times in A therefore $\ell_{A-appear}$=2, ℓ_{1count}=4 , ℓ_{2count}=2
Inc1 = Inc1 + (3 – 1) * ((4 * (4 + 1) / 2) + (2 * (2 + 1) / 2)) = 8 + 26 = 34
Step 3.2: for each appearance of ℓ=p_3 in A find similarity windows with Y:
First iteration:
$\ell_{1count} \geq 3$ therefore :Red2 = 4
w_1=<p_2 p_3> , $w_{1\ell}$ = 2 , $L_{W(1)}$ = 2 , Red1 = 3
w_2= < p_3 p_4 p_5 p_3> , $w_{2\ell}$ = 1 , $L_{W(2)}$ = 4 , Red1 = 3 , w_3= -
Step 3.3: Replace ℓ in Y with (ℓ, ℓ_{icount}): Y = < p_8 p_2 p_3,3 p_4 p_5 p_3 p_7>
Second iteration:
w_1=< p_2 p_3,3 > , $w_{1\ell}$ = 2 , $L_{W(1)}$ = 4 , L_1=3 , Inc2 = 2
w_2 = < p_3,3 p_4 p_5 p_3 > , $w_{2\ell}$ = 1 , $L_{W(2)}$ = 6 , L_1 = 3 , Inc2 = 8 , w_3 = -
Second iteration: ℓ=p_3 ℓ_{count}=2
Step 3.1: p_3 is appeared 2 times in A therefore $\ell_{A-appear}$=2
ℓ_{1count}=4 , ℓ_{2count}=2 , Inc1 = 47
Step 3.2:
First iteration:
for ℓ=p_3 in A find similarity windows with Y
$L_1 \geq 3$ therefore : Red2 = 7
w_1=- , w_2= - , w_3= <3,3 4 5 3> , $w_{3\ell}$ = 6 , $L_{W(3)}$ = 6 , Red1 = 3 , w_4 = -
Step 3.3: Replace ℓ in Y with (ℓ, ℓ_{icount}) : Y = < p_8 p_2 p_3,3 p_4 p_5 p_3,2 p_7>
Step 3.2:
Second iteration: w_1=- , w_2 = - , w_3 = < p_3,3 p_4 p_5 p_3,2 > , $w_{3\ell}$ = 6 , $L_{W(3)}$ = 7 , L_1 = 2 , Inc2 = 13

Step4: $S_{A,B} = S_{X,Y} + Inc1 + Inc2 - Red1 - Red2 = 17 + 47 + 13 - 3 - 7 = 67$

Figure 3. An example of EACS-Kernel algorithm on two sample execution paths

Computing *Red1* in step 2.2 needs to find the similarity window for each occurrence of ℓ in Y. Usually in most cases the similarity windows are very short but in the worst case finding the similarity window needs to traverse all the predicates of path Y. Therefore the cost of this step is $O(\ell_{Y-\text{appear}} \times |Y|)$. Since steps 2.1 and 2.2 of the algorithm should be done for each iteration of subsequence ℓ in A, the total cost of step 2 is $O(\ell_{A-\text{appear}} \times (\ell_{Y-\text{appear}} + \ell_{Y-\text{appear}} \times |Y|)) = O(\ell_{A-\text{appear}} \times \ell_{Y-\text{appear}} \times |Y|)$.

Step 3: The complexity of computing *Inc1* and *Red2* is $O(m_A)$ as described above.

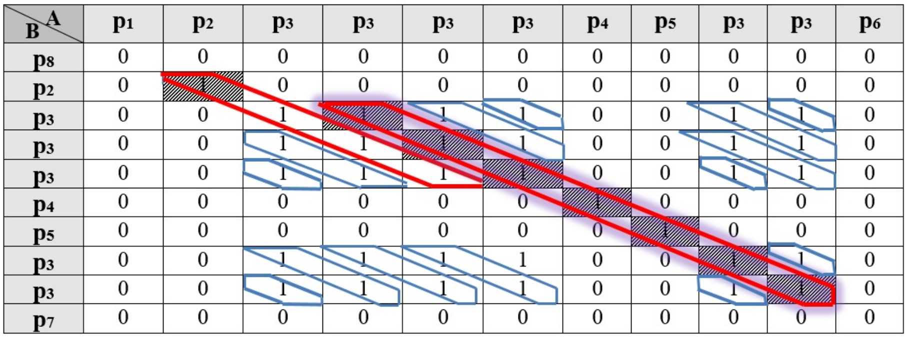
The complexity of computing *Red1* is $O(\ell_{A-\text{appear}} \times |A|)$. Computing *Inc2* needs to find the similarity windows therefore this step requires $O(\ell_{A-\text{appear}} \times |A|)$ time as computing *Red1*. Therefore the complexity of step 3 is $O(\ell_{Y-\text{appear}} \times (\ell_{A-\text{appear}} + 2 \times \ell_{A-\text{appear}} \times |A|)) = O(\ell_{A-\text{appear}} \times \ell_{Y-\text{appear}} \times |A|))$.

Step 4: The cost of step 4 is $O(1)$. The overall complexity of EACS-Kernel algorithm is $O(|X|^2 + |Y|^2 + \ell_{A-\text{appear}} \times \ell_{Y-\text{appear}} \times |Y| + \ell_{A-\text{appear}} \times \ell_{Y-\text{appear}} \times |A|)$. Since the total number of appearances of a specific subsequence in different locations of a program execution path is small, $\ell_{Y-\text{appear}}$ and $\ell_{A-\text{appear}}$ could be considered as constant parameters in practice. Thus, the computation cost of EACS-Kernel algorithm to measure the similarity between main sequences A and B could be considered as $O(|X|^2 + |Y|^2)$. However, computation cost of ACS-Kernel algorithm to measure the similarity between A and B is $O(|A|^2 + |B|^2)$. Therefore the time overhead of EACS-Kernel algorithm to measure the similarity between original paths A and B is less than ACS-Kernel algorithm on A and B because the length of shortened paths X and Y is very smaller than the original paths A and B. Thus EACS-Kernel decreases the time overhead of ACS-Kernel on sequences containing repeated patterns.

3.2.3 EADLCS-Kernel

A major difficulty with the EACS-Kernel is that its feature space dimensionality could be very high because most of the common sub-paths considered as the dimensions have overlaps. In order to reduce the dimensionality, the longest disjoint common sub-paths amongst the execution paths can be selected as the dimensions of the feature space. To achieve this, a new algorithm, EADCLS (Extended All Disjoint Longest Common Sub-paths Kernel), is presented below. The basic idea behind EADCLS is to build a similarity matrix M for each pair of execution paths A and B, whose columns represent the predicates in A and its rows represent the predicates in B, is built. The components $M_{i,j}$ of the matrix are either one or zero dependent on whether the i^{th} predicate in A and the j^{th} predicate in B are identical or not. The longest non-overlapping consecutive ones on the main diagonals are selected as the longest common sub-paths between the execution paths. For instance in Figure 4, similarity matrix M and the common sub-paths between two execution paths $A = \langle p1\, p2\, p3\, p3\, p3\, p3\, p4\, p5\, p3\, p3\, p6\rangle$ and $B = \langle p8\, p2\, p3\, p3\, p3\, p4\, p5\, p3\, p3\, p7\rangle$

are represented. Two longest common sub-paths between execution paths A and B are $\langle p2\, p3\, p3\, p3\rangle$ and $\langle p3\, p3\, p3\, p4\, p5\, p3\, p3\rangle$ as shown in Figure 4. After removing the overlaps between the longest common sub-paths, two subsequences $\langle p2\rangle$ and $\langle p3\, p3\, p3\, p4\, p5\, p3\, p3\rangle$ are selected as disjoint and longest common sub-paths between A and B.



B \ A	p1	p2	p3	p3	p3	p3	p4	p5	p3	p3	p6
p8	0	0	0	0	0	0	0	0	0	0	0
p2	0	1	0	0	0	0	0	0	0	0	0
p3	0	0	1	1	1	1	0	0	1	1	0
p3	0	0	1	1	1	1	0	0	1	1	0
p3	0	0	1	1	1	1	0	0	1	1	0
p4	0	0	0	0	0	0	1	0	0	0	0
p5	0	0	0	0	0	0	0	1	0	0	0
p3	0	0	1	1	1	1	0	0	1	1	0
p3	0	0	1	1	1	1	0	0	1	1	0
p7	0	0	0	0	0	0	0	0	0	0	0

Figure 4. The basic idea of similarity measure between each pair of execution paths by EADLCS-Kernel

The algorithm EADLCS-Kernel finds the similarity between two execution paths S and T according to their disjoint longest common sub-paths. In step 1 of the algorithm, the longest common sub-paths between S and T, are found. In effect, this is similar to finding the longest consecutive ones on the main diagonals of the similarity matrix. In step 2 of the algorithm, the overlaps of the other common sub-paths in S and T with the longest common sub-path is removed from the sub-paths. Practically, this is similar to removing the overlaps from the main diagonals apart from the diagonal including the longest sub-path. The longest common sub-path is inserted in the list of dimensions for building the feature space. Removing the longest common sub-paths from the list of common sub-paths, the algorithm is repeated as far as there are no more overlaps.

Selecting the longest non overlapping common sub-paths as the dimensions of the feature space, the dimensions which have full projection on the longest common sub-paths are removed. In this way the dimensionality of the feature space is reduced. In comparison with EACS-Kernel while preserving the accuracy, EADLCS-Kernel increases the speed of the failure prediction, significantly.

In the algorithm EADLCS-Kernel S and T are sequences of elements where each element contains the name Name, the number of appearance Count of a predicate in the execution path of program. MaximumCommonSubpaths is a set of sequences each representing a relatively longer Common sub-path between the execution path sequences S and T. Step 1 in the algorithm finds common sub-paths between S and T and collects them in the MaximumCommonSubpaths set. The repeated overlapping predicates $S_{(j+Sj.count-min)} \dots S_{(j+Sj.count)}$ and $T(k + Tk.count -$

$min)\ldots T(k+Tk.count)$ are selected as starting predicates of a new common sub-path. Step 2 in the algorithm selects maximum non-overlapping sub-paths amongst the common sub-paths. To have a better understanding of the EADLCS-Kernel algorithm, an example is shown in Figure 5.

```
Algorithm EADLCS-Kernel
Input: Let $S$ and $T$ be two distinct execution path sequences of a program $P$.
Output: Similarity measure between two execution path sequences $S$ and $T$.
Step 1: For $i := 1$ to $|T|$ do // Traverse $T$ Starting at $T_1$ to $T_n$, Where $n = |T|$
Begin
    Initiate $k := i$;
    For $j := 1$ to $|S|$ do // Traverse $S$ Starting at $S_1$ to $S_n$, Where $n = |S|$
     Begin
        If the name of the predicate $T_k$.name is the same as $S_j$.name then
         Begin
            If the number of appearance $T_k$.count is equal to the $S_j$.count then
                If a new common sub-path is not already started then
                    NewCommonSubpath.StartPoint := position of $S_j$ and $T_k$ in paths
            Else // the number of appearance $T_k$.count is not equal to the $S_j$.count
            Begin
                Let $min$ := the minimum value of $S_j$.count, $T_k$.count;
                If a new common sub-path is already started then
                 Begin
                    NewCommonSubpath.EndPoint := position of $S_{j+min-1}$ and $T_{k+min-1}$ in paths
                        Add NewCommonSubpath to the MaximumCommonSubpaths set;
                End if a new common sub-paths is already started
                NewCommonSubpath.StartPoint := position of $S_{j+Sj.count-min}$ and $T_{k+Tk.count-min}$
                in paths
            End Else
          End if the name of the predicate $T_k$.name is the same as $S_j$.name
         Else If a new common sub-path is already started then
         Begin
             NewCommonSubpath.EndPoint := position of $S_j$ and $T_k$ in paths
            Add NewCommonSubpath to the MaximumCommonSubpaths set;
         End Else
         $k$++;
     End For
     If a new common sub-path is already started then
     Begin
        NewCommonSubpath.EndPoint := position of $S_{j+Sj.count}$ and $T_{k+Tk.count}$ in paths
       Add NewCommonSubpath to the MaximumCommonSubpaths set;
    End if
End For
Step 2: While (MaximumCommonSubpaths set is not empty) do
```

```
Begin
    Find the longestCommonSub-path in MaximumCommonSubpaths
    For each Sub-Path in MaximumCommonSubpaths
        Remove overlaps(Sub-Path, longestCommonSub-path)
    $similarity(s,t) +=$ length of LongestCommonSub-path;
End While
```

Let ExecutionPath1=<$p_1 p_2 p_3 p_3 p_3 p_3 p_4 p_5 p_3 p_3 p_6$> and ExecutionPath2=<$p_8 p_2 p_3 p_3 p_3 p_4 p_5 p_3 p_3 p_7$> be two execution paths of a program P.

Input: S = <(p_1,1) (p_2,1) (p_3,4) (p_4,1) (p_5,1) (p_3,2) (p_6,1)>
T = <(p_8,1) (p_2,1) (p_3,3) (p_4,1) (p_5,1) (p_3,2) (p_7,1)>

Step 1:
Iteration 1:
MaximumCommonSubpaths={[StartPoint=(2,2), EndPoint=(5,5) , common subpath=< $p_2 p_3$ $p_3 p_3$>], [StartPoint =(4,3), EndPoint =(10,9) , common subpath=< p_3 $p_3 p_3$ $p_4 p_5 p_3$ p_3>]}
...

Iteration 4:
MaximumCommonSubpaths={[StartPoint =(2,2), EndPoint =(5,5) , common subpath=< $p_2 p_3$ $p_3 p_3$>], [StartPoint =(4,3), EndPoint =(10,9) , common subpath=< p_3 $p_3 p_3$ $p_4 p_5 p_3$ p_3>],[StartPoint =(3,8), EndPoint =(4,9) , common subpath=< $p_3 p_3$>]}
...

Step2:
longestCommonSubpath=[StartPoint =(4,3), EndPoint =(10,9) , common subpath=< p_3 $p_3 p_3$ $p_4 p_5 p_3$ p_3>]
MaximumCommonSubpaths={[StartPoint =(2,2), EndPoint =(2,2) , common subpath=< p_2>], [StartPoint =(4,3), EndPoint=(10,9) , common subpath=<p_3 $p_3 p_3$ $p_4 p_5 p_3$ p_3>]}

Figure 5. An example of EADLCS-Kernel algorithm on two execution paths

A. Computation cost of the EADLCS-Kernel algorithm

The time complexity of EADLCS-Kernel algorithm to measure similarity between each pair of execution paths is computed as follows:

Step 1: The complexity of step 1 to find the longest common sub-paths between S and T is $O(|S||T|)$.

Step 2: The complexity of step 2 for removing the ovelaps of the common sub-paths with the longest common sub-paths is $O((|S|+|T|)^2)$.

Therefore, the order of EADLCS-Kernel is $O((|S|+|T|)^2)$.

B. Benefits of the proposed EADLCS-Kernel

The kernel is semi positive definite: The semi positive definiteness of the EADLCS-Kernel could be explained as follows:

1. {disjoint longest common sub-paths between each pair of execution paths} $\subseteq$ {all common sub-paths amongst the execution paths}
2. Let F_{EADLCS} indicate the feature space where all the disjoint longest common sub-paths are considered as the dimensions and F_{EACS} indicates the feature space where all common sub-paths are considered as the dimensions. According to the previous item, F_{EADLCS} is a subspace of F_{EACS}.
3. If we show that F_{EADLCS} spans F_{EACS}, then it can be concluded that EADLCS-Kernel is semi positive definite like EACS-Kernel. To proof this it is enough to show that the dimensions of F_{EACS} can be projected onto the dimensions of F_{EADLCS} [19]. From 1 it is obvious that all common sub-paths have projection onto the disjoint longest common sub-paths.

EADLCS-Kernel is faster than the EACS-Kernel: EADLCS-Kernel reduces the number of feature space dimensions as well as the size of the dimensions. It is important to show that the accuracy reduction of the EADLCS-Kernel in comparison with the EACS-Kernel is tolerable. With this assumption, EADLCS-Kernel would be efficient to apply as the core of the failure prediction classifier. The amount of accuracy reduction of EADLCS-Kernel is explained in Section 4.

4 EXPERIMENTAL RESULTS

In order to demonstrate the performance of EADLCS-Kernel, the results of several experiments conducted on two well-known test suites, Siemens and SPEC2000, are described in this section. A brief description of the Siemens and SPEC2000 test suites is presented in Table 1.

Siemens is one of the benchmark suites more related to bug detection for software testing. Siemens benchmark contains seven small programs each program is associated with some versions. Each version is injected with one bug. Faulty versions simulate a wide variety of realistic bugs. Better testing tools can distinguish more buggy versions from correct ones. Siemens programs and related test cases are obtained from Software Repository Infrastructure (SIR) [20]. SPEC2000 benchmark have been used before by several failure prediction methods to measure the performance. SPEC2000 programs are obtained from Standard Performance Evaluation Corporation website [21]. In order to evaluate our failure prediction method, the following criteria are applied:

1. **Code coverage:** The effect of code coverage property and the size of the training dataset, applied for building the failure prediction classifier on its performance, accuracy and precision.

Application	Lines of Code	Number of Test Cases	Number of Predicates	Description
Siemens	2803	555962	472	Siemens contains seven programs print tokens: lexical analyzer print tokens2: lexical analyzer replace: pattern replacement schedule: priority scheduler schedule2: priority scheduler tcas : altitude separation tot info: information
SPEC2000	40138	10430	4375	Standard Performance Evaluation Corporation. SPEC2000 contains 14 programs: 164.gzip, 254.gap, 197.parser, 175.vpr, 177.messa, 179.art, 181.mcf, 183.equake, 186.crafty, 188.ammp, 255.vortex, 256.bzip2, 300.twolf, 176.gcc

Table 1. A brief description of two test suites

2. **Alarm notification:** The amount of time a predicted failure notification could be postponed.
3. **Quality:** Evaluation and comparison of the quality, in terms of performance, precision and accuracy, of the proposed failure prediction approach with the existing ones.
4. **Dimensionality:** The effect of feature space dimensionality on the failure prediction speed and accuracy.
5. **Time overhead:** The amount of time overhead imposed by the failure prediction classifier on the program execution time.

4.1 Code Coverage

The code coverage criterion uses the measure of the program execution paths covered by the training data set to evaluate the thoroughness of the resultant failure prediction classifier. Figure 6 shows the impact of the code coverage, by the data set applied to run the Siemens programs, on the resultant SVM classifier quality. We have applied a code coverage measurement tool, Testwell CTC++ tool [22], to measure the coverage of our training data set. CTC++ applies a code coverage metric called Linear Code Sequence and Jump (LCSAJ) to measure the coverage of acyclic paths resulted from a test suite. In order to measure the LCSAJ coverage capability of an applied test suite, a metrics called Test Effectiveness Ratio (TER) is used. The TER metric for a test set T, applied to run a program P, is computed as follows:

$$TER = \frac{\#\,\text{LCSAJs exercised in program } P \text{ by } T}{\#\,\text{feasible LCSAJs of program } P}. \tag{8}$$

Figure 6 shows the F-measure of the failure prediction classifier built with EADLCS and EACS proposed kernels on seven programs of Siemens test suite according to TER metric. As shown in Figure 6, F-measure of the proposed failure prediction classifier with the EADLCS kernel is 0.82 while its test effectiveness ratio is about 60 %.

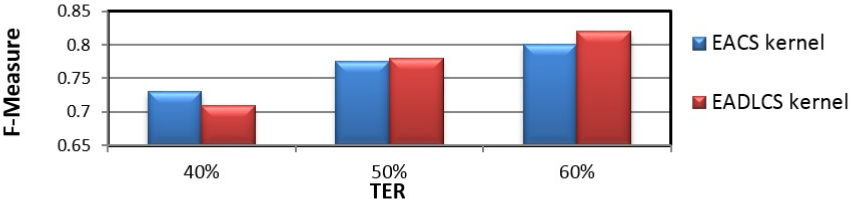


Figure 6. F-Measure of failure prediction classifier with EACS and EADLCS kernels on Siemens according to TER metric

The path coverage capability of the test suites is measured by applying mutation testing as well. Faults or mutations, some small changes in the code, are automatically seeded into program code, then tests are run. If the tests fail then the mutation is killed and if the tests pass then the mutation lives. The quality of the tests can be gauged from the percentage of mutations killed. Mutation testing is applied to examine whether the test suite is sufficiently complete to detect the incorrect paths due to the faults seeded in the program [23, 24, 25]. Milu is an efficient and flexible C mutation testing tool [26].

In practice, it is difficult to obtain failing execution paths on programs that are ready for deployment. Therefore the sizes of failing and passing training sets are unbalanced. In Figure 6 the proportion of failing to passing test cases is only about 16 %. But it is interesting to note that the quality of the proposed method is not appreciably affected by these imbalanced data sets. This advantage is achieved by SVM which does not lose its accuracy with unbalanced data [27]. When there is a small number of failing runs, SVM classifier, in a conservative manner, predicts failure by mistake and, as could be seen in Figure 7, the False-Positive Rate increases while Precision is decreased. Consequently, the failure prediction model tends to increase the number of alerts. However, the less number of failing executions are left without failure alert and hence the Recall rate increases. The interesting point is that, in practice, the quality of the proposed technique which is specified by F and Accuracy metrics does not harm, noticeably, as shown in Figure 7.

4.2 Alarm Notification

When the execution path of a program is detected as anomalous by the failure prediction classifier, the classifier could warn the upcoming failure in the early stages of the execution or the alarm could be postponed for some time before program failure. In this section we study the effect of postponing the failure warnings during the execution of programs in real environments. In this experiment, Failure Notification

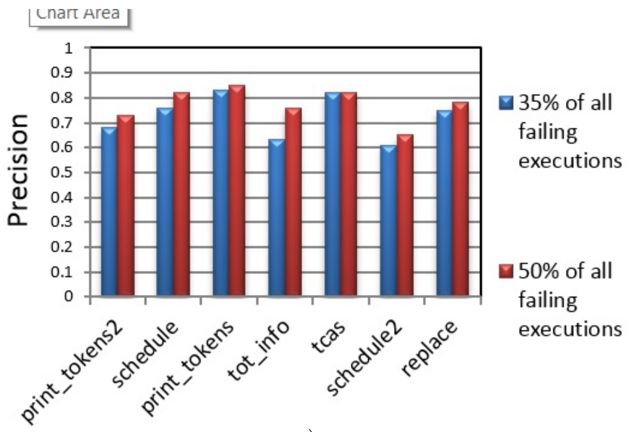
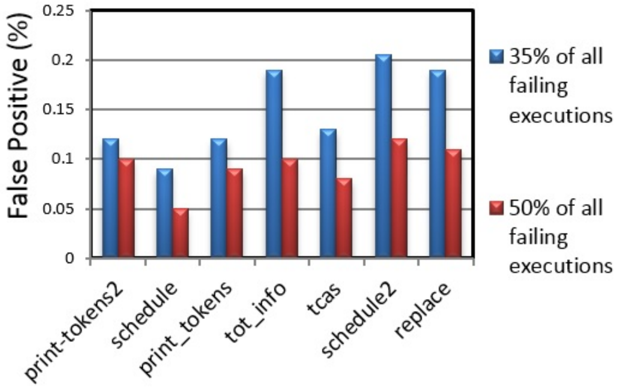
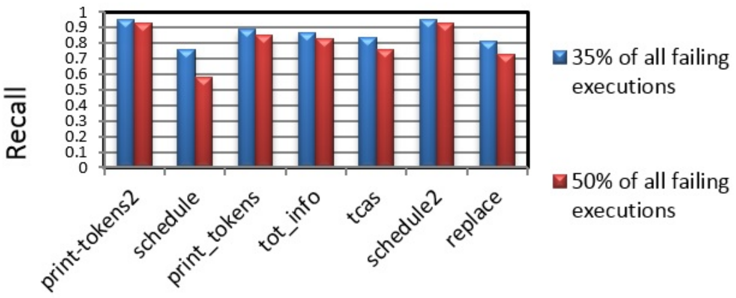


Chart Area
Precision
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
print_tokens2
schedule
print_tokens
tot_info
tcas
schedule2
replace
35% of all failing executions
50% of all failing executions

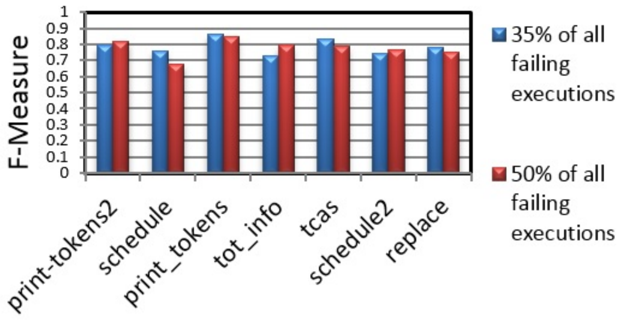
a)


False Positive (%)
0.25
0.2
0.15
0.1
0.05
0
print-tokens2
schedule
print_tokens
tot_info
tcas
schedule2
replace
35% of all failing executions
50% of all failing executions

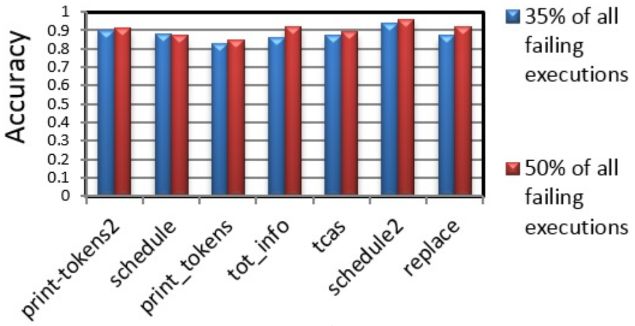
b)


Recall
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
print-tokens2
schedule
print_tokens
tot_info
tcas
schedule2
replace
35% of all failing executions
50% of all failing executions

c)



d)



e)

Figure 7. Evaluation of failure prediction classifier with EACS-Kernel according to different percentages of failing executions: a) Precision, b) False Positive, c) Recall, d) *F*-measure, e) Accuracy

Postponing (FNP) metric is defined:

FNP = the number of times the execution path of the program is detected as failing by the classifier before the failure warning is announced.

Early warnings of failure could provide programmers enough time to understand and fix the problematic errors. Our experiments show that usually postponing failure alarm could reduce the number of incorrect failure warnings (i.e. false positive) and the precision of the failure prediction classifier increases as shown in Figure 8. However, in some of the faulty execution paths, the early failure detections of the classifier could be missed by increasing FNP and the program may run in an apparently correct execution flow in subsequent stages of execution. This situation may result in increasing false negative rate. With due attention to increase false negative and decrease false positive by increasing the FNP, the accuracy of the prediction classifier usually remains constant. *F*-Measure is completely dependent on the changes of precision and accuracy therefore it could be a suitable metric to select the appropriate FNP for the programs under test. From the results in Figure 8, it is evident that the three programs of Siemens, print_tokens2, schedule and schedule2, have highly distinguishing sub-paths around the very beginning of the executions. However, it is dependent on the location of faults in the program.

4.3 Quality

To have an appropriate basis for estimating the quality of EACS and EADLCS kernels, we have compared them to each other and to some other approaches on seven programs of Siemens and 13 programs of SPEC2000 test suite. The kernels are applied in the SVM classifier. Figure 9 shows the experiments on Siemens programs. Figure 9 shows that the failure prediction classifier by the proposed kernels provides more accurate failure predictions than the known approaches like Argus and DIDUCE. As shown in Figure 9, EADLCS kernel has slightly more false positive alarms compared with EACS kernel and therefore the overall precision of EACS is greater than EADLCS kernel. However, false negative alarms of EADLCS are lower than EACS kernel, indicating higher Recall of EADLCS kernel.

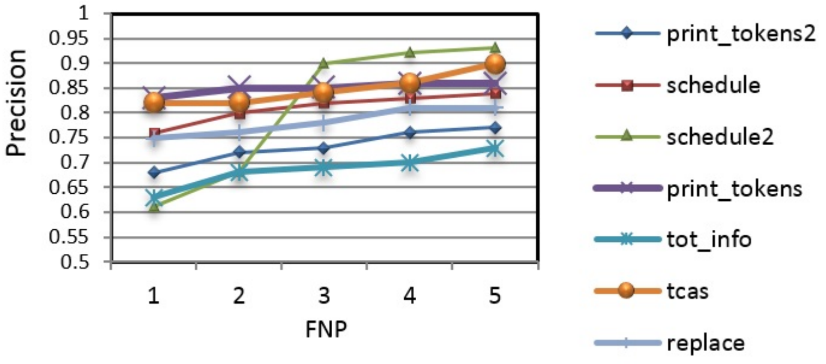
According to these experiments, we may conclude that the failure prediction model with EADLCS kernel acts more conservatively compared to the model with EACS kernel. But there is a smaller number of failing executions in which the EADLCS kernel cannot predict failure. Figure 10 shows the comparison of the failure prediction classifier to some known existing approaches in the field of failure detection of software programs. The horizontal axis shows the human effort to reach the main cause of failure from the predictions reported by the bug detection approach. In the worst case 100 % of program code is examined to distinguish the main cause of failure. On average, when 10 % of code are examined, failure prediction classifier can help the user localize 63 % of the bugs. Argus and DIDUCE are able to localize 58 % and 60 % of the bugs respectively when 10 % of the code are examined. Although online failure prediction major design goal is runtime bug detection, the Figure 10 shows that the bug reports our approach generates are precise enough that it can be used for efficient bug localization as well.

4.4 Dimensionality

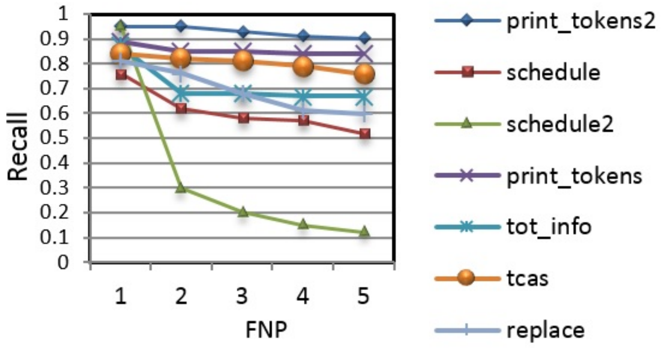
Figure 11 shows the impact of reducing the size and the number of dimensions of the feature space on time overhead and accuracy of the failure prediction classifier.

As shown in Figure 11, it is concluded that the size reduction of feature space dimensions reduces the accuracy of the EADLCS and EACS kernel classifier by 8.6 % and 9 % while it reduces time overhead by 40 % and 43 %, respectively, compared to ADLCS and ACS kernels.

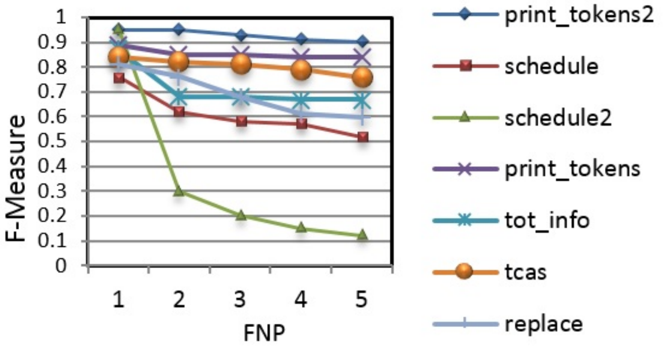
From Figure 11 it is concluded that the dimensionality reduction in EADLCS kernel improves the time overhead on program execution time by 62 % compared to EACS kernel while the accuracy is reduced by only 0.01 %.



a)

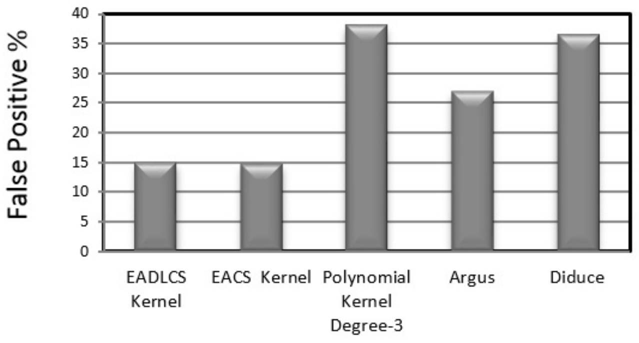


b)

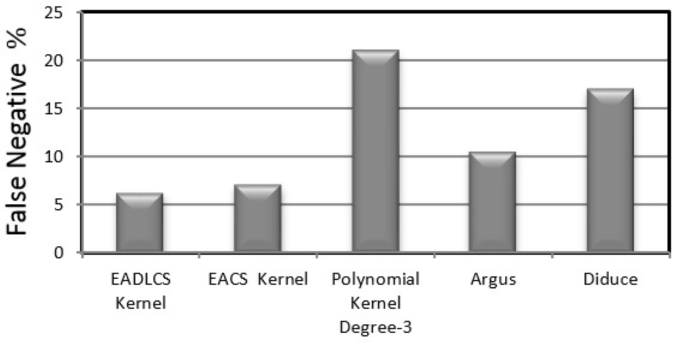


c)

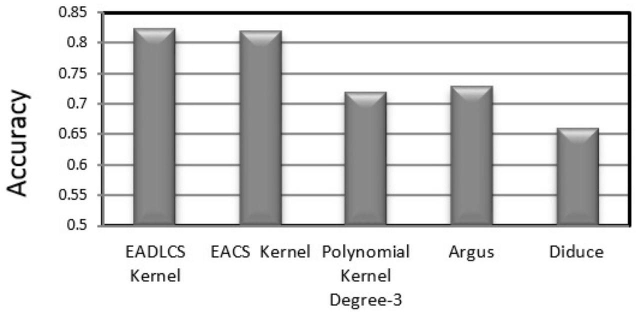
Figure 8. Effect of FNP choice on the EACS kernel failure prediction classifier in Siemens programs: a) Precision, b) Recall, c) *F*-Measure



a)



b)



c)

Figure 9. Compare failure prediction classifier to existing approaches: a) False Positive, b) False Negative, c) Accuracy

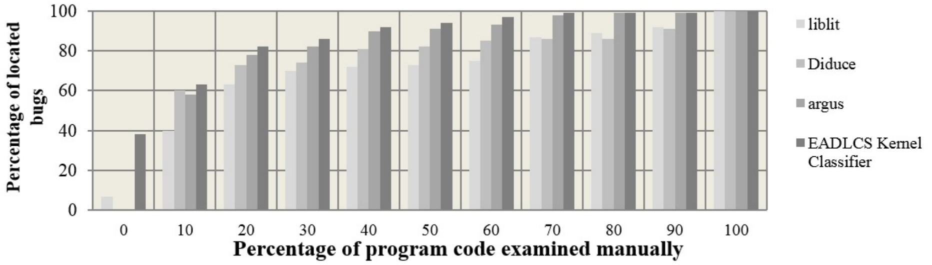


Figure 10. Compare failure prediction classifier to existing approaches: bug located vs. code examined manually

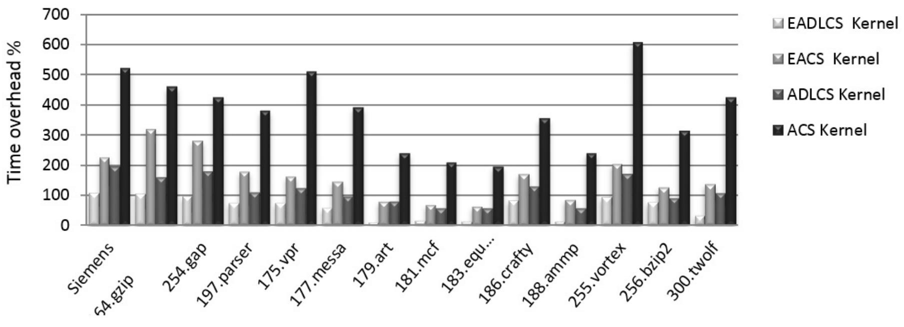
4.5 Time Overhead

In Figure 12, the runtime overhead of SVM classifier with EACS, EADLCS and polynomial [28] kernels are compared with C-DIDUCE [16] and Argus [17] on SPEC2000 programs. Siemens benchmark applications are very small (some are less than 100 lines of code), hence not suitable to measure time overhead.

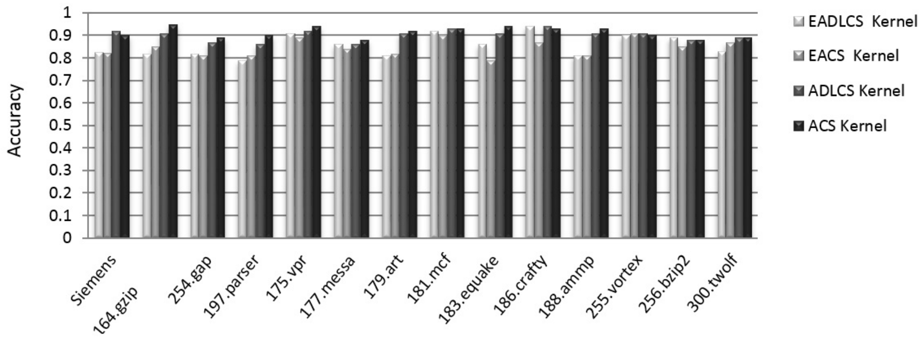
As shown in Figure 12, the runtime detection overhead of SVM failure prediction classifier with EADLCS kernel is less than the existing approaches. The SVM classifier with EADLCS kernel suffers from an average of 120 % execution time overhead on SPEC2000 programs which is nearly 30 % better than the approaches like C-DIDUCE. As shown in Figure 9 and Figure 10, the failure prediction classifier generates more accurate predictions than the best known bug detection techniques. Therefore it can be concluded that the proposed approach not only reduces the overhead at run time but provides more accurate predictions than the existing methods like Argus and DIDUCE.

5 CONCLUSIONS AND FUTURE WORKS

In this paper, a kernel classifier for online prediction of upcoming failures in software programs is presented. The classifier predicts the termination state of a program execution path as failing or passing by computing its similarity with two regions of failing and passing training executions in a customized feature space. The feature space dimensions are the commonly traversed paths among failing and passing training executions. The speed of the classifier mainly depends on the size and the number of the feature space dimensions. The main contribution of this paper is to improve the speed of the failure prediction classifier by reducing the dimensionality, number and the size of the dimensions of the classifier feature space. The number of feature space dimensions is reduced by removing the dimensions which have projection on each other. Moreover, the sizes of the dimensions are reduced by replacing each consecutively repeated pattern with a single iteration of the pattern. We have proposed two new kernels to measure similarities in the feature space with reduced dimensionality.



a)



b)

Figure 11. Failure prediction classifier with EACS and EADLCS kernels: a) Time overhead, b) Accuracy

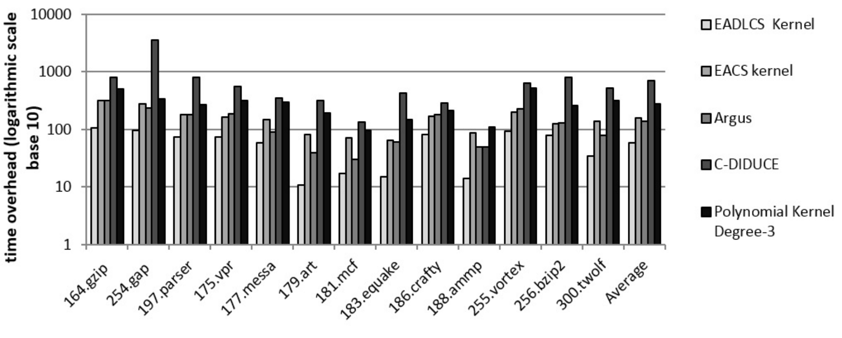


Figure 12. Time overhead of EACS and EADLCS kernel classifiers compared with Polynomial kernel classifier, Argus and C-DIDUCE

The experimental results show that the proposed kernel classifier improves the speed of the failure prediction while preserving accuracy of the predictions. Our future study is going to focus on the reduction of failure prediction overhead by applying a window on execution paths of programs to reduce the size of the execution path sequences to be classified.

REFERENCES

[1] Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. 2nd Edition. Morgan Kaufmann, 2009.

[2] Liblit, B.: Cooperative Bug Isolation. Doctoral dissertation, University of California, 2004.

[3] Parsa, S.—Arabi, S.: Software Online Bug Detection: Applying a New Kernel Method. IET Software, Vol. 6, No. 1, February 2012, pp. 61–73.

[4] Shawe-Taylor, J.—Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, 2004.

[5] Parsa, S.—Arabi, S.: A New Semantic Kernel Function for Online Anomaly Detection of Software. ETRI Journal, Vol. 34, 2012, No. 2, pp. 288–291.

[6] Herbrich, R.: Learning Kernel Classifiers Theory and Algorithms. 1st Edition. MIT Press, 2002.

[7] Salfner, F.—Lenk, M.—Malek, M.: A Survey of Online Failure Prediction Methods. ACM Computing Survey, Vol. 42, 2010, No. 3, pp. 1–42.

[8] Mili, A.—Tchier, F.: Software Testing: Concepts and Operations, Quantitative Software Engineering Series. 1st Edition. John Wiley and Sons, 2015.

[9] Liu, H.—Xu, L.—Yang, M.—Yan, M.—Zhang, X.: Predicting Component Failures Using Latent Dirichlet Allocation. Mathematical Problems in Engineering, Vol. 2015, Article ID 562716, 2015.

[10] Hamill, M.—Goseva, K.: Exploring Fault Types, Detection Activities, and Failure Severity in an Evolving Safety-Critical Software System. Software Quality Journal, Vol. 23, 2015, No. 2, pp. 229–265.

[11] Zhang, P.—Muccini, H.—Polini, A.—Li, X.: Run-Time Systems Failure Prediction Via Proactive Monitoring. Proceedings of 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), November 2011, pp. 484–487.

[12] Baah, G. K.—Gray, A.—Harrold, M. J.: OnLine Anomaly Detection of Deployed Software: A Statistical Machine Learning Approach. Proceedings SOQUA, 2006, pp. 70–77.

[13] Salfner, F.: Event-Based Failure Prediction: An Extended Hidden Markov Model Approach. Doctoral dissertation, Humboldt-Universität zu Berlin, Germany, 2008.

[14] Pytlik, B.—Renieris, M.—Krishnamurthi, S.—Reiss, S.: Automated Fault Localization Using Potential Invariants. Proceedings International Workshop Automated and Algorithmic Debugging, Ghent, Belgium, 2003, pp. 273–276.

[15] Alipour, M. A.—Groce, A.: Extended Program Invariants: Applications in Testing and Fault Localization. Proceedings of the Ninth International Workshop on Dynamic Analysis (WODA 2012), 2012, pp. 7–11.

[16] Fei, L.—Midkiff, S. P.: Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies. Proceedings PLDI, 2006, pp. 84–95.

[17] Fei, L.—Lee, K.—Li, F.—Midkiff, S. P.: Argus: Online Statistical Bug Detection. Proceedings FASE, 2006, pp. 308–323.

[18] Chandola, V.—Banerjee, A.—Kumar, V.: Anomaly Detection: A Survey. ACM Computing Surveys (CSUR), Vol. 41, 2009, No. 3, Article No. 15.

[19] Strang, G.: Introduction to Linear Algebra. Fourth Edition, Wellesley Cambridge Press, 2009.

[20] Software Infrastructure Repository. Available on: `http://sir.unl.edu/`.

[21] Standard Performance Evaluation Corporation. Available on: `https://www.spec.org/cpu2000/`.

[22] Testwell CTC++ tool. Available on: `http://www.testwell.fi/`.

[23] Ammann, P.—Offutt, J.: Introduction to Software Testing. 1st Edition. Cambridge University Press, 2008.

[24] Untch, R. H.—Offutt, J.—Harrold, M. J.: Mutation Analysis Using Mutant Schemata. Proceedings Introduction Symposium Software Testing and Analysis, New York, NY, USA, 1993, pp. 139–148.

[25] Jia, Y.—Milu, M. H.: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. Proceedings Introduction Conference Testing: Academic and Industrial Conference Practice and Research Techniques, Windsor, UK, August 2008, pp. 29–31.

[26] Milu. Available on: `http://www.dcs.kcl.ac.uk/pg/jiayue/milu`.

[27] Chen, D.—He, Q.—Wang, X.: On Linear Separability of Data Sets in Feature Space. Intell Conference Development and Learning, Vol. 70, London, UK, 2007, pp. 2441–2448.

[28] Chang, C. C.: LIBSVM: A Library for Support Vector Machines. ACM Transactions on Intelligent Systems and Technology (TIST), Vol. 2, 2011, No. 3, Article No. 27.

Somaye Arabi Naree received her B.Sc. in computer science from Alzahra University, Iran, and both her M.Sc. and Ph.D. degrees in computer science, from the Iran University of Science and Technology. She is Associate Professor of mathematics and computer science at the Kharazmi University. Her research interests include software testing, software engineering and data mining.

Saeed Parsa received his B.Sc. in mathematics and computer science from Sharif University of Technology, Iran, and both his M.Sc. and Ph.D. degrees in computer science from the University of Salford, England. He is Associate Professor of computer science at the Iran University of Science and Technology. His research interests include software testing, software engineering and reverse engineering.