Reducing Failure Analysis Time: An Industrial Evaluation

Mojdeh Golagha, Alexander Pretschner
Department of Computer Science
Technical University of Munich
Munich, Germany
Email: {golagha, pretschn}@in.tum.de

Dominik Fisch, Roman Nagy
Research and Development
BMW
Munich, Germany
Email: {dominik.fisch, roman.nagy}@bmw.de

***Abstract*—Testing and debugging automotive cyber physical systems are challenging. Developing and integrating cyber and physical components require extensive testing to ensure reliable and safe releases. One important cost factor in the debugging process is the time required to analyze failures. Since large number of failures usually happen due to a few underlying faults, clustering failures based on the responsible faults helps reduce analysis time. We focus on the software-in-the-loop and hardware-in-the-loop levels of testing where test execution times are high. We devise a methodology for adapting existing clustering techniques to a real context. We augment an existing clustering approach by a method for selecting representative tests. To analyze failures, rather than investigating all failing tests one by one, testers inspect only these representatives. We report on the results of a large scale industrial case study. We ran experiments on ca. 850 KLOC. Results show that utilizing our clustering tool, testers can reduce failure analysis time by more than 80%.**

***Index Terms*—failure analysis; failure clustering; SiL testing; HiL testing; automotive CPS;**

I. Introduction

Testing and debugging automotive Cyber Physical Systems (CPS) are becoming more challenging because of rapid innovation. These innovations are implemented by adding applications and software features to cars. The applications execute on dozens of programmable electronic control units (ECU) that communicate via various communication buses such as CAN, FlexRay, LIN and automotive Ethernet. To test ECUs, developers perform multiple levels of testing, ranging from unit tests to functional acceptance tests. One important cost factor in testing and debugging such systems is the time required to analyze failures and localize faults (in our terminology, a failure is the deviation of actual runtime behavior from intended behavior, and a fault is the reason for the deviation [1]). In practice, there is always very limited time available to analyze failures and find underlying faults.

SiL (software-in-the-loop) and HiL (hardware-in-the-loop) are two levels of testing that encompass huge numbers of test cases (TC) with high execution times. Testing an ECU on the SiL level means that its *software components* are tested within a simulated environment model but without any actual hardware [2]. HiL testing aims at testing the integration of hardware and software of an ECU in a simulated environment. At the HiL level, the software hence runs on the final ECU, again within a simulated environment [2]. TCs in these two levels are frequently re-run as regression tests. Due to large number of TCs at the SiL and HiL levels, their execution time is significant. Therefore, in the process of analyzing failures, it would be too expensive to execute tests immediately after correcting a single fault. In practice, to remove all the underlying faults, testers dig into failing tests one by one to make sure they have found and resolved all the faults before re-running the tests. This process takes a significant amount of time to be finished.

Clustering TCs has been shown to help reduce failure analysis time. Failure clustering techniques attempt to group failing tests with respect to the faults that caused them [3]. If there are several failing TCs, these failing tests may be clustered such that tests in the same cluster fail because of the same hypothesized fault. In an ideal world, it is then sufficient to analyze only one representative TC for each cluster to discover all the underlying faults. This process eliminates the need for analyzing each failing test individually.

Jones et al. [4] introduced a parallel debugging process as an alternative to sequential debugging. They suggest that in the presence of multiple faults in a program, clustering failing tests based on their underlying faults, and assigning clusters to different developers for simultaneous debugging, reduce the total debugging cost and time. They propose two clustering techniques. Using the first technique, they cluster failures based on execution profiles and fault localization results. They start the clustering process by using execution profile similarities, and complete it using fault localization results. Their second technique suggests to only use the results of fault localization.

We applied their clustering techniques in our context not for debugging in parallel but for selecting representative tests to start the debugging process. However, by clustering failures and suggesting some representative tests to the testers, we can, firstly, take advantage of debugging in parallel. Secondly, we can reduce the effort needed to investigate all the failures. In fact, by a proper method of selecting representatives, we would be able to eliminate the need for debugging in parallel as well. Consider the following example. If there are 100 failing tests caused by 6 underlying faults, an ideal clustering finds 6 clusters from the failing tests, each cluster pointing to one distinct fault. The debugging in parallel approach then

suggests assigning these 6 clusters to 6 developers to work on them simultaneously. Meanwhile, our solution suggests selecting one representative for each cluster; thus, investigating 6 failures rather than 100 failures. Assigning the task to several developers is not necessary.

Hoegerle et al. [5] introduced another parallel debugging method which is based on integer linear programming [6]. They applied the above-mentioned second clustering technique of Jones et al. to compare it with their own debugging approach. Their results show that this clustering technique of Jones et al. is not so effective. Our experiments confirmed this finding. In fact, early results were so discouraging that we did not pursue this avenue any further. However, different from Hoegerle et al. who did not evaluate the first technique, we do provide evidence that in contrast to the second technique, the first technique is effective if it is adapted to the context and if it is used for clustering failures rather than debugging and localizing faults. Therefore, in our paper we propose a methodology for adapting the idea of clustering based on execution profiles and fault localization to a real context.

In our paper, we solve a problem, namely reducing failure analysis time, which was identified while conducting industry-oriented research with BMW for testing software releases of car ECUs. Together with the test engineers at BMW we defined the problem statement and the solution:

Problem. There are many test cases at the SiL and HiL levels of testing. These tests are frequently executed as regression tests. In case of failures, there are usually only a few underlying faults that cause a large number of failures. Considering the significantly high execution time of the tests, it is not always possible to find the first fault, resolve it and re-run all the tests. One current process in practice is investigating all failing tests to find the faults, removing the faults and re-running the tests. This approach makes debugging a very time-consuming process. How can we reduce failure analysis time?

Solution. We propose to cluster failing TCs with respect to the fault that caused them, select one representative for each cluster, investigate only the representatives, find the underlying faults and resolve them, then re-run the tests.

Contribution. We propose a methodology for adapting the idea of debugging in parallel [4] to a real context, including an approach to choosing adequate parameter values and a tailored approach for measuring the quality of clustering. We add a method for selecting representatives as a final step to the clustering technique introduced by [4]. We investigate the effectiveness of the adapted approach in a large-scale industrial study with ca. 850 KLOC. We also show how available software-related information at the SiL level can be used to analyze failures at the HiL level. Finally, we suggest new metrics for the effectiveness of test case clustering.

Organization. In section II, we describe our methodology and approach. In section III, we explain the experiment including evaluation metrics and results. We review related work in section IV. Finally, in section V, we conclude the paper and suggest future directions.

II. Methodology

In the following, we describe our methodology. We first explain the first clustering technique introduced by Jones et al. [4] that we use as part of our approach, while completing it with our two additions that are selecting representative tests and utilizing SiL information for analyzing HiL failures. Then, we explain our strategy for gathering different methods and metrics and identifying the best ones.

A. Approach

First, we run a test suite and extract an execution profile for each TC. Second, utilizing agglomerative hierarchical clustering [7], we build a tree of failing tests based on the similarity of execution traces. In order to cut this tree into clusters we need to know the best number of clusters. In the third step, we hence utilize fault localization techniques to decide on the best number of clusters. Then, we cut the tree into the found number of clusters. Finally, in the fourth step, we calculate the centers of the clusters and choose the failures which are closest to the centers as representative tests. The tool we implemented is meant to be used as an assisting tool for testers. Therefore, they receive the list of representatives to investigate them. Steps one, two and three are taken from [4]. In the following, we describe the steps in detail.

1) Running Tests and Profiling Executions: The first step is to run the tests and profile executions. To profile TC executions, we instrument the code using an open source tool for measuring code coverage for C++ and C.[1] Executing a program using this tool results in a report about which lines of code have been executed. The main advantage of this tool in our case is its compatibility with the Visual C++ compiler used in Microsoft Visual Studio, which, in the context of our case study, is the development tool used by our industry partner.

We developed a tool called Spectralizer. The functionality of Spectralizer consists of three stages. The first stage is to prepare test execution according to the startup parameters and the system under test to be analyzed. The next stage involves the execution of the tests while recording the coverage information and the test results using the coverage tool. The third stage is optional and depends on the level of abstraction at which the analysis should happen. Since the tool we use provides statement-level coverage only, for function-level coverage analysis the third step is needed to partially parse the source files to aggregate the statement-level information to function-level coverage. The coverage information is the execution profile and is fed to the next step as input. In our case study, since controlling the hardware is in the form of functions to read/write signals, it is important to use function-level profiles.

2) Generating Failure Tree: To cluster failures we utilize agglomerative hierarchical clustering. Hierarchical clustering groups data objects by creating a binary tree called *dendrogram*. The tree is a multilevel hierarchy, where clusters at one

[1] OpenCppCoverage, available at https://opencppcoverage.codeplex.com/, licensed under the GNU General Public License version 3 (GPLv3).

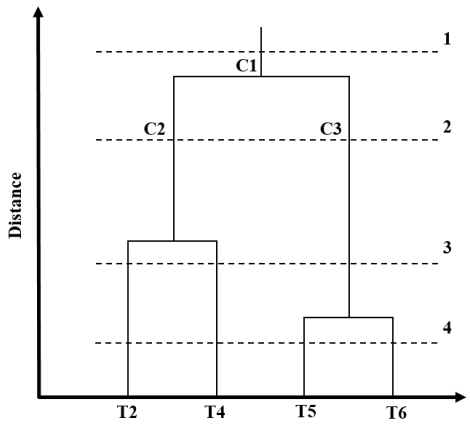

Fig. 1. Hierarchical Clustering of 4 Failing Tests

level are merged as clusters at the next level [7]. This method of clustering builds a hierarchy of clusters. Agglomerative hierarchical clustering is a bottom-up approach. Each data object is in its own cluster at the bottom level and pairs of clusters are combined as one moves up the hierarchy [8].

In our work, each single failure at first is in its own cluster. Using some distance metrics, the similarity between clusters is measured, and the two most similar clusters are merged to build a new cluster. Execution profiles received from previous step represent feature sets in the clustering. Considering the example of Fig. 1, there are 4 failing tests, namely T2, T4, T5 and T6. We compute distances between clusters by measuring the similarity between profiles. In Fig. 1, T5 and T6 have the most similar profiles. They are merged into one cluster as the tree grows upward. The process of merging two most similar clusters continues to the root of the tree where all the failures are in one cluster.

Hierarchical clustering does not need a predefined number of clusters, k. However, since we want a partition of (by definition: disjoint) clusters, the hierarchy needs to be cut at some point (if there is more than one underlying fault). There are a number of criteria in the literature to determine the cutting point. In this work, as suggested by [4], we use fault localization techniques to decide the cutting point as described in following.

3) Cutting the Failure Tree by Fault Localization: Fault localization is a debugging technique that identifies the faulty entities of a program. The spectrum-based, or coverage-based, techniques that we utilize in this work rank suspicious entities using metrics like similarity coefficients or association metrics. There is a plethora of these metrics in the literature. The underlying idea is that entities of programs that have been executed more in failing executions and less in passing executions are more suspicious. Thus, developers should first investigate those entities to find the underlying faults.

Spectrum-based fault localization works as follows:

- We cut the program code into its entities at the desired level of granularity, e.g. statements, functions, or classes.

TABLE I
HIT SPECTRUM

Function	Program (P)	T1	T2	T3	T4	T5	T6
FUN1	int add(int a, int b) {int r; r=a+b; return r;}	•	•	•	•	•	•
FUN2	int sub(int a, int b) {int r; r=a-b; return r;}	•	•	•	•	•	•
FUN3	int Mult(int a, int b) {int r; r=a*b; return r;}	•	•	•	•	•	•
FUN4	int Div(int a, int b) {int r; r=a+1/b; \\bug return r;}		•	•	•	•	•
FUN5	int Cstm(int a, int b) {int r; r=(a*3)/(2+b); return r;}	•	•	•	•	•	•
FUN6	int print () {cout <<"Hi!";}	•		•			
FUN7	int main() { \\main body here! return 0;}	•	•	•	•	•	•
Execution Results (1=pass, 0=fail)		1	0	1	0	0	0

- We run the test suite and create a table of coverage information that shows which entities have been executed in each test execution and which have not. This table is called a hit spectrum. Performing step 1 above, we already have this information. Table I shows an example for a program (P) which has been cut into its functions (FUN1 ...FUN7), and its 6 test cases (T1 ...T6). Each dot in the table means that the respective function has been executed while executing the respective test.
- We measure the suspiciousness of each entity by using a similarity metric. An example is the D^4 metric [9]:

$$D^4 = \frac{(N_{CF})^4}{N_{UF} + N_{CS}}, \tag{1}$$

where N_{CF} is the number of failing executions that have covered the entity (function in the example); N_{UF} is the number of failing executions that have not covered the entity; and N_{CS} is the number of passing executions that have covered the entity.

As equation 1 shows, we use both passing and failing tests for this step.
- We rank the functions based on their suspiciousness score. The highest score gets rank 1. Table II shows the suspiciousness score and rank of the seven functions of program (P). The ranked list shows FUN4 as the most suspicious function. In this example, the fault localization technique pinpoints the faulty function at the first rank.

We utilize this technique to find the best k, or the cutting point, of the hierarchical tree. Considering Fig. 1, the question is which of the dashed horizontal lines 1 to 4 is the best cutting

TABLE II
FUNCTION SUSPICIOUSNESS RANKS

Function	Suspiciousness	Rank
Fun1	128	2
Fun2	128	2
Fun3	128	2
Fun4	256	1
Fun5	128	2
Fun6	0	3
Fun7	128	2

TABLE III
LIST OF MUTATIONS APPLIED [5], [14], [15]

Mutations
Negate condition in "for" or "while" statement
Replace integer constant by another integer
Delete a statement
Replace an operator by another operator
Assign null in assignment statements
Replace return expression with null

point. Liu and Han [10] as well as Jones et al. [4] suggest that if the failures in two clusters identify the same entities as faulty entities, they most likely failed due to the same reason and should be merged into one cluster.

This process can also be considered in a top-down manner. This technique computes the *fault localization rank* for the children of a parent to decide whether the parent is a better cluster or it should be divided into its two children. The result of fault localization result is a ranked list of entities from the most to the least suspicious. To check similarity between ranked lists, we used the Jaccard [11] set similarity as suggested by [4], defined on two sets A and B as follows.

$$Similarity(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

If the similarity of the fault localization rank of two children is smaller than a predefined threshold, they are (likely) pointing to different faults. They are dissimilar and should not be merged. Thus, the parent cluster is not a good stopping point and should be divided into its children. Otherwise, the parent is a better cluster and this is the stopping point for clustering. According to Fig. 1, the first step is to decide whether dashed line 1 is a better cutting point or dashed line 2. To answer this question, the fault localization rank at cluster c2 is compared to fault localization rank at cluster c3. If the similarity between these two sets is larger than the predefined threshold, they are similar and the parent c1 is a better clustering than dividing it into two clusters c2 and c3. As the result, line 1 is the cutting point. If line 1 is not the cutting point, the process continues to the point that no more division is needed. We somewhat arbitrarily selected 0.85 as the similarity threshold in our evaluation tests, and because of the good results, did not see a need to question this choice.

4) Selecting Representative TCs: We have already grouped TCs based on their hypothesized root causes by generating the failure tree and cutting it into clusters. Now, we need to suggest a representative for each cluster. Testers investigate only the representatives to find and report all the faults. Since it is likely that clustering is imperfect, the selection of representatives for a cluster has a great importance: We are aware that clusters are unlikely to be 100 percent pure [12].

We require our solution to make the representatives reliable. To avoid selecting an outlier (a failure which does not belong to the fault class that has the majority in the cluster) as a representative in clustering, we hence calculate the center of the cluster and find the k-nearest neighbors (KNN) [13] to the center. These KNNs are selected as representatives of the respective cluster. KNN search finds the nearest neighbors in a set of data for each data point. Based on discussion with test engineers, we considered k=1 in our experiment.

5) Utilizing SiL Execution Profiles for HiL Analysis: For analyzing failures at the HiL level, we need to perform an extra step. Developers typically generate SiL and HiL tests from the same source. With SiL simulation, the software component is tested independently from the target hardware.

However, in our context, the information used for SiL can be re-used for the subsequent HiL tests. At the HiL level, real hardware components are integrated in a simulated environment for testing. Thus, there is a common system model between SiL and HiL. The difference is the use of mocked values instead of real hardware on SiL testing. Based on this information, it seems intuitive that if we do not have access to the whole execution profile of the HiL tests (containing both software and hardware parts), we are able to analyze failures at this level utilizing execution profiles from the SiL executions. Note that in those cases where some tests pass on the SiL level but fail on the HiL level, failure analysis is more difficult and time consuming.

Consider a TC used both at the SiL and HiL levels. First, we generate its execution profile by instrumenting the code at the SiL level. Second, we run the test at the HiL-level and get the verdict (failing/passing) information. Then, we combine these two to generate the feature set for hierarchical clustering and hit spectrum for fault localization as described in step 1. The rest of the steps will be the same. Therefore, we are able to analyze SiL and HiL failures using essentially the same information.

B. Parameter Setting

We have mentioned earlier that there is a plethora of similarity metrics and methods available in hierarchical clustering and spectrum-based fault localization. We need to select metrics with the highest performance in our context. We set up a training phase that helps us select the right metrics and parameters for this reason. We had access to 25 software components for different ECUs of a car. These software components have several sub-projects. We somewhat arbitrarily selected 100 sub-projects, 4 sub-projects from each software component, and injected faults in their source codes. Table III shows the list of mutations applied. We generated 100 first-order mutated

TABLE IV
AVERAGE VALUES OF COPHENET CORRELATION COEFFICIENT ON TRAINING DATA

	UPGMA	UPGMC	WPGMC	Shortest	Ward	WPGMA	Furthest
Euclidean	0.94	0.93	0.92	0.93	0.90	0.94	0.88
CityBlock	0.89	0.89	0.89	0.88	0.87	0.89	0.80
Minkowski	0.94	0.93	0.92	0.93	0.90	0.93	0.94
Cosine	0.90	0.89	0.89	0.87	0.88	0.89	0.87
Correlation	0.90	0.89	0.89	0.89	0.87	0.89	0.87
Jaccard	0.90	0.90	0.89	0.89	0.88	0.90	0.88

versions and used these 100 faulty versions for our training data set. Although the external validity of using mutants in assessing the testing techniques has raised concerns, Andrews, Briand and Labiche [16] as well as Namin and Kakarla [14] suggest that under specific circumstances, replacing real faults with mutants has no significant impact on results. However, any analysis or generalization should be justified according to the influential factors including mutation operators, test suite size and programming languages. We applied the mutation operators previously used in the literature for fault localization [5], [14], [15]. To mitigate external threats, we then evaluated our approach using real faults.

1) Clustering Metric and Method Selection: To do hierarchical clustering, we need an algorithm for computing the distance between clusters and a metric for computing the distance between pairs of data objects, failures in our case. There are several methods and metrics in the literature. We selected seven methods: unweighted average distance (UPGMA), centroid distance (UPGMC) [17], furthest distance [17], weighted center of mass distance (WPGMC) [17], shortest distance [17], inner squared distance(ward) [17] and weighed average distance (WPGMA) [17]; and six distance metrics: Euclidean distance, City Block [18], Minkowski [18], Cosine [18] (1-Cosine), Correlation [18] (1-Correlation) and Jaccard (1-Jaccard). We utilized the Cophenetic correlation [19] to compare the performance of different combinations of the selected methods and metrics. Cophenetic correlation is a measure of "how faithfully a hierarchical tree preserves the pairwise distances between the original unmodeled data points or objects" [19].

Table IV shows the average Cophenet value for clustering on 100 programs. In this experiment we clustered all the TCs (no matter if their verdict is passing or failing) based on their execution profiles' similarity. The goal was to find the best metric and method for our context. Therefore, it was sufficient to measure how precisely the clustering tree represents the distance between TCs at this point. By looking at any clustering tree, it is possible to measure the distance between data objects. In a good clustering tree, these distances should be similar or close to the original distances between unclustered data objects. The results show that there is not a huge difference between different combinations of methods and metrics. Thus, we chose to utilize UPGMA and Euclidean distance that are showing slightly better results for our context. Higher coefficients mean more accurate hierarchical trees.

2) Fault Localization Metric Selection: There are many different approaches for fault localization in the literature. We focused on surveys that compare and evaluate different metrics for spectrum-based fault localization. Lucia et al. [20] did a comprehensive study of association measures for fault localization. Wong et al. [9] introduced a new metric named DStar and investigated the effectiveness of it in comparison with 31 other similarity metrics for locating bugs. The results of different evaluations show that there is not a single metric that has the best performance for every program. It depends on the context, programming language [9] and also the fault density [12].

For our experiment, we picked 72 metrics from these papers to find the most effective one for our context. Table V shows the list of implemented metrics.

To evaluate the effectiveness of different metrics, we measure the average percentage of code needed to be investigated to locate faults. This evaluation approach has been used in many works such as [9] and [20]. As the result of fault localization, developers receive a ranked list of entities based on their suspiciousness score. Thus, first, they check the entity with rank 1. If that is a false positive, they continue with rank 2, 3 etc. to find the faulty entity. Therefore, the percentage of the code examined to locate the fault is one indicator for the effectiveness of the metric.

To reach our goal in this step, we used 72 metrics to locate faults in our 100 faulty programs. We measured the code examined for the best and worst cases and then calculated the median for each metric. Since we had 100 programs, to make comparison of results more straightforward, we calculated the average for each metric over all programs. In Fig. 2, the x-axis shows the metrics (using their numbers presented in Table V) and the y-axis shows the average percentage of the code examined.

Our results indicate huge differences between the effectiveness of different metrics. As the graph illustrates, the minimum value belongs to metrics number 71 and 72 which are D^4 and D^5. These metrics have the same base formula. The only difference is that in D^5, the numerator is raised to the power of 5 rather than 4 in equation 1. D^4 has fewer calculations. Therefore, we chose D^4 as the fault localization metric.

Note that using the D^4 metric, a tester would be required to check about 40% of the code in average to locate the fault. This number is not a convincing result. Thus this technique cannot be used as an effective *fault localization technique* in

our case. However, it is worthwhile to notice that we are using this technique as a *similarity measure rather than a debugging technique*. Thus, we will not get a penalty if the similarity metric that we use is unable to pinpoint the exact fault location in the first rank in some cases. Nevertheless, it is important to use a good similarity metric for this reason since a bad metric may show similar rank for a great percentage of entities of the code and this makes distinguishing between different failing executions difficult.

TABLE V
LIST OF SPECTRUM-BASED FAULT LOCALIZATION METRICS IMPLEMENTED [9], [20]

Metric	**Metric**	**Metric**
Braun-Banquest (M_1)	Odd Ratio (M_{25})	Two Way Support (M_{49})
Dennis (M_2)	Yules Q (M_{26})	Two Way Support Variation(M_{50})
Mountford (M_3)	Yules Y (M_{27})	Loevinger (M_{51})
Fossum (M_4)	Kappa (M_{28})	Sebag-Schoenauer (M_{52})
Pearson (M_5)	J-Measure (M_{29})	Least Contradiction (M_{53})
Gower (M_6)	Support (M_{30})	Odd Multiplier (M_{54})
Micheal (M_7)	Confidence (M_{31})	Example and counter-example Rate(M_{55})
Pierce (M_8)	Laplace (M_{32})	Zhang (M_{56})
Baroni-Urbani (M_9)	Conviction (M_{33})	Sorensen-Dice (M_{57})
Tarwid (M_{10})	Interest (M_{34})	Anderberg (M_{58})
Ample (M_{11})	Piatesky-Shapiro's (M_{35})	Gini Index (M_{59})
Phi (M_{12})	Certainty Factor (M_{36})	Rogers and Tanimoto (M_{60})
Arithmetic Mean (M_{13})	Added Value (M_{37})	Ochiai II (M_{61})
Cohen (M_{14})	Collective Strength (M_{38})	Rogot2 (M_{62})
Fleiss (M_{15})	Klosgen (M_{39})	Hamann (M_{63})
zoltar (M_{16})	Information Gain (M_{40})	Sokal (M_{64})
Harmonic Mean (M_{17})	Coverage (M_{41})	Rogot1 (M_{65})
Simple-Matching (M_{18})	Accuracy (M_{42})	Kulczynski (M_{66})
Hamming (M_{19})	Leverage (M_{43})	Goodman (M_{67})
Scott (M_{20})	Relative Risk (M_{44})	D^1 (M_{68})
Dice (M_{21})	Interestingness Weighting Dependency(M_{45})	D^2 (M_{69})
Jaccard (M_{22})	Goodman and Kruskal(M_{46})	D^3 (M_{70})
Tarantula (M_{23})	Normalized Mutual Information(M_{47})	D^4 (M_{71})
Ochiai (M_{24})	One way support (M_{48})	D^5 (M_{72})

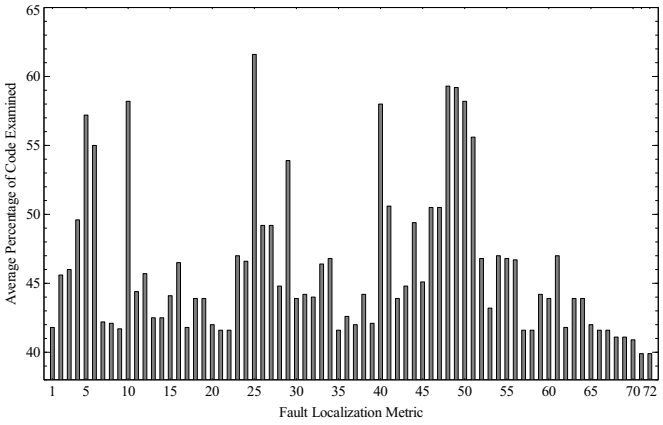

Fig. 2. Evaluation Results of Implemented Fault localization Metrics

III. EXPERIMENT

We can now refine the research questions and describe evaluation metrics, evaluation results and threats to validity.

A. Research Questions

We aim to reduce failure analysis time. To this end, we cluster failures. We need to answer the following questions to evaluate how successful our solution is in reducing analysis time:

RQ1: Can our clustering tool effectively serve as an assisting tool for testers to reduce the analysis time?

RQ2: How much reduction in time is achievable using clustering?

RQ3: Which metrics and clustering methods are most effective in our context?

Technically speaking, these questions boil down to determining how "well" our clustering schema works. We start by defining what "well" means.

B. Evaluation of the Clustering Results

Typical objectives in clustering are high intra-cluster similarity and low inter-cluster similarity. However, good scores on these criteria do not necessarily mean good effectiveness in our application. To do a proper evaluation, we need to consider the industrial objectives and research questions.

We chose *purity*, *F-measure* and *entropy* from the literature and propose two new metrics *Red* and *ARed* as our evaluation metrics. To compute purity, each cluster is assigned to the class which is most frequent in the cluster. A class is an underlying fault in our case. Then, the accuracy of this assignment is measured by counting the number of correctly assigned objects dividing by N [8] where N is the number of objects. Let $\Omega = \{\omega_1, \omega_2, ..., \omega_K\}$ be the set of K clusters and $\mathbf{C} = \{c_1, c_2, ..., c_J\}$ the set of J classes. Then purity is:

$$Purity(\Omega, \mathbf{C}) = \frac{1}{N} \sum_{k} \max_{j} |\omega_k \cap c_j| \qquad (3)$$

If the number of clusters is large, achieving high purity is easier. If each object is in its own cluster, purity is 100%.

Purity is the most important factor from a practical point of view since it means all the failures in one cluster failed because of the same reason. However, there is a trade-off between the quality of clustering and the number of clusters. Since fewer clusters mean more reduction in analysis time, which is our objective, we need a measure for this trade-off.

In the clustering literature, the F-measure [8] is proposed to this end:

$$\mathbf{F}_1 = 2 \times \frac{precision \times recall}{precision + recall}, \tag{4}$$

where precision=TP/(TP+FP) and recall=TP/(TP+FN).

In our case, a true positive (TP) means assigning two failures grounded in the same fault to the same cluster; a true negative (TN) means assigning two failures grounded in different faults to different clusters; a false positive (FP) means assigning two failures grounded in different faults to the same cluster; and a false negative (FN) means assigning two failures grounded in the same fault to different clusters. We also use the entropy metric proposed by Rogstad and Briand [21] to capture the trade-off: For each of the K clusters in Ω, they compute the number of failures of type i (failing because of underlying fault i) that belongs to cluster c_j (f_{ij}) divided by the total number of failures of type i ($|f_i|$):

$$E_F(f_i, C) = -\sum_{j=1}^{K}(\frac{f_{ij}}{|f_i|})log(\frac{f_{ij}}{|f_i|}), \tag{5}$$

and total deviation entropy is:

$$E_{F-TOT}(F, C) = -\sum_{i=1}^{J} E_F(f_i, C). \tag{6}$$

We provide these values chiefly for comparison purposes with existing work.

In order to measure how successful we are in the reduction of analysis time, we propose the Reduction (Red) metric which measures the avoided effort as complement of the ratio of the number of clusters to the number of test cases:

$$Red = (1 - \frac{K}{\#\ of\ TCs}) \times 100. \tag{7}$$

Note that Red alone is not sufficient for judging the effectiveness. To realize how successful we are in the reduction of analysis time, it is also necessary to know the ideal reduction in each case. As examples, first consider a test suite with 8 failing tests and 6 faults. In an ideal clustering, there should be 6 clusters each one pointing to 1 fault which means investigating 6 tests rather than 8 tests. Thus, the maximum possible reduction is 25%. As a second example, consider a test suite with 20 failing tests and 2 faults. In this case, the maximum possible reduction with an ideal clustering is 90%. Just considering the absolute effective reduction Red does not tell us *how good we could have been*.

Therefore, we measure how much of the ideal reduction ($IRed$) has been achieved, and name it Achieved Reduction ($ARed$):

$$ARed = \frac{Red}{IRed} \times 100 \tag{8}$$

TABLE VI
SOFTWARE COMPONENTS OF THE CASE STUDY

	Lines of Code	**Number of TCs**
SWC1-OldVersion1	$\simeq$ 145,952	1103
SWC1-OldVersion2	$\simeq$ 145,952	1103
SWC1*	145,952	1103
SWC2*	113,470	1303
SWC3	14,434	23
SWC4	59,349	890
SWC5	42,308	793
SWC6	87,984	687
SWC7	7,164	424
SWC8	29,930	234
SWC9	59,349	163

where $IRed$ is:

$$IRed = (1 - \frac{J}{\#\ of\ TCs}) \times 100 \tag{9}$$

In addition, since more clusters mean more representative tests and thus less reduction in analysis time, $ARed$ can also serve as an easily interpretable measure that shows how much finding extra clusters affects the effectiveness of our approach.

In sum, we are looking for high scores of purity, F-measure, Red and ARed and low scores of entropy. Later, we will explain why from a practical viewpoint, we consider the combination of *purity* and *ARed* optimal for analyzing results.

C. Results of the Industrial Case Study

To evaluate the effectiveness of our approach in reducing failure analysis time, we conducted a study with our industrial partner. We used the current as well as some old versions of the code and reported bugs available in the repositories of BMW. Table VI shows information of the software components under test. We have changed the names of the components for confidentiality reasons.

We performed the clustering in parallel with the partner's current process of failure analysis. This allows us to use the testers' analysis reports as ground truth for assigning failures to faults. In two of the test suites, SWC1 and SWC2, marked by *, we had access to HiL tests as well and could analyze SiL and HiL failures at the same time.

The results of the clustering evaluation are shown in Table VII. We are able to achieve high scores of purity and accuracy. In an ideal solution the number of clusters equals the number of underlying faults. The low scores of recall and thus F-measure and high scores of entropy show that the number of clusters are usually larger than the number of faults. Nevertheless, the numbers in the two last columns show that even though we do not always find the ideal number of clusters, we are able to achieve a huge reduction in analysis time anyway.

F-measure and entropy are measured and included for comparison purposes with existing work. However, we believe these numbers need to be interpreted with care. For example, 0.26 F-measure in SWC1-OldVersion2 does not sound convincing but it corresponds to a reduction of 95.41%

TABLE VII
CLUSTERING EVALUATION RESULTS

	#Failures	#Faults	#Clusters	Purity	Precision	Recall	F-measure	Entropy	Red	ARed
SWC1-OldVersion1	24	2	8	1	1	0.34	0.51	1.36	66.66	72.72
SWC1-OldVersion2	240	3	11	0.98	0.96	0.15	0.26	3.12	95.41	96.61
SWC1*	32	8	11	1	1	0.74	0.85	1.32	65.62	87.49
SWC2*	25	4	4	0.88	0.76	0.84	0.8	0.93	84	100
SWC3	7	1	1	1	1	1	1	0	85.71	100
SWC4	39	5	6	0.77	0.43	0.58	0.48	0.98	84.61	94.28
SWC5	30	3	3	0.97	0.92	0.99	0.96	0.69	90	100
SWC6	19	4	4	1	1	0.8	0.89	0.69	78.94	100
SWC7	66	1	2	1	1	0.96	0.97	0.042	96.96	98.45
SWC8	9	4	6	1	1	0.42	0.59	1.07	33.33	60
SWC9	8	1	1	1	1	1	1	0	87.5	100

(in the number of test cases, which we tacitly assume to be proportional to the time needed to debug). Also, in the same case, we were able to achieve 96.61% of the maximum possible reduction. This means that having to investigate 11 representatives instead of the ideal 3, we have lost only 3.39% (100-96.61) more reduction in time. The reason is the large number of TCs (1103) in this case. We would also conjecture that from a practical perspective, ARed may be easier to interpret than F-measure and Entropy.

When testing SWC1 and SWC2, some tests failed at the HiL but passed at the SiL level. This is not uncommon in practice. We generated the execution profiles based on SiL execution and attached the HiL verdicts to them. Then, we added these failures to the rest of the failures at the SiL level and clustered all the failures. 100% resp. 88% purity and 87% resp. 100% as achieved reductions in analysis time for these two cases show that the idea of matching SiL executions with respective HiL verdicts is successful and promising.

Based on the above results, we answer the research questions as follows:

RQ1: Considering purity and ARed scores, we believe that our clustering tool can serve as an effective assisting tool for testers. Utilizing this tool, they do not need to investigate all the failing tests one by one. Instead, they check the representative tests to find the underlying faults. In all the evaluation tests, the representative selection was successful in the sense that selected representatives were always failing due to the fault that had the majority in the cluster.

We believe that the observed high scores of purity and the proper representative selection method make our tool a reliable assistant.

Table VIII shows the number of failures per fault in each test suite. These numbers show that although in some cases we had highly imbalanced data, our clustering approach yielded high accuracy and was able to distinguish between all the distinct faults.

RQ2: Compared to the current process of debugging, utilizing our clustering tool can help in saving more than 80% of the failure analyzing time (where, once again, we equate the number of test cases with the time needed to analyze them).

RQ3: In our context, we found that the combination of "D^4 as fault localization metric" and "Euclidean distance as distance metric and UPGMA as clustering method" yielded the best performance for the chosen granularity of "functions" as entitites for execution profiling.

To answer this question, we set up the training phase, which we argue can be reproduced for a different class of systems on the grounds of existing software components. One important point in our case is that we cannot use any clustering method that needs a predefined number of clusters or an upper threshold for the number of clusters. The number of clusters means the number of underlying faults which is unknown a-priori. Depending on the software component, development stage, level of testing, granularity of the execution profile, we may get different results.

D. Threats to Validity

Our results show that we could achieve a great reduction in failure analysis time by clustering the failing tests. Nevertheless, we do not claim that our experiment setup and results will be valid for all other contexts. We offered a methodology for adapting this idea to different industrial contexts.

1) Threats to Internal Validity: Test suite quality has a great impact on fault localization [22] and therefore clustering. If all the test cases in a test suite cover the same parts (or most parts) of the code, it will be a difficult task to distinguish different failing (in the sense of different underlying causes) executions. There are results that show the threats to validity of spectrum-based fault localization [23] and its shortcomings [24]. However, we are not employing this technique for debugging and localizing faults, but rather as a similarity measure to better understand which failures happened due to the same reasons.

Another important factor is the granularity of profiling elements. We chose function-based coverage information both as object features in hierarchical clustering and for generating hit spectrum in fault localization based on the nature of our context. Using more fine-grained or coarse-grained entities may impact the performance.

Our first assumption was that each failure has only one reason behind it. Even though this assumption might not always hold true, we did not encounter any other case in our experiment. This may be because of the quality of the test

TABLE VIII
NUMBER OF FAILURES PER FAULT FOR EACH TEST SUITE

	Number of failures per fault(F)
SWC1-OldVersion1	F1:21 F2:3
SWC1-OldVersion2	F1:26 F2:202 F3:12
SWC1*	F1:14 F2:3 F3:1 F4:3 F5:1 F6:8 F7:1 F8:1
SWC2*	F1:4 F2:14 F3:6 F4:1
SWC3	F1:7
SWC4	F1:1 F2:17 F3:14 F4:1 F:6
SWC5	F1:2 F2:27 F3:1
SWC6	F1:16 F2:1 F3:1 F4:1
SWC7	F1:66
SWC8	F1:2 F2:5 F3:1 F4:1
SWC9	F1:8

suites as well. We are aware that there may be some cases where two or more faults cause the same failure.

The similarity threshold used in comparing fault localization ranks may also affect the results. Selecting a smaller number may result in a fewer number of clusters and selecting a larger number may result in a greater number of clusters. Therefore, this value can also impact the trade-off between purity and the number of clusters. We arbitrarily chose 0.85. The results show that it is a suitable threshold.

2) Threats to External Validity: Focusing on one specific domain and one specific kind of ECUs can be a potential threat to our external validity. It is unlikely that a general solution can be found for grouping failures in different contexts. To find the best solution for our case we set up a training phase. To mitigate threats to external validity, we performed the training phase on real industrial data. We used all available software components of car ECUs. Running more tests of course will help us better understand possible generalizations.

As mentioned earlier, fault injection gives rise to threats to external validity. Recent works [5], [25] suggest that real faults are replaceable with mutations if mutations are used with caution and if they are representative of real faults. In our work, we used some of the mutation operators previously applied in the literature to learn the best metric and distance measure. For the evaluation, we used real world systems and real-world faults.

IV. RELATED WORK

Dickinson et al. [26] introduce the idea of clustering execution profiles to identify failing executions from passing executions. Podgurski et al. [3] utilize a supervised classification to group failing executions with similar causes. Bowering et al. [27] use Markov models to present program executions and then cluster the models to categorize the executions. Liu and Han [10] claim that trace similarity is not enough to cluster failing executions with respect to the causes. They propose the idea of using fault localization and suggest that two failing traces are similar if they roughly point to the same fault location. Jones et al. [4] utilize the previous ideas and combine the notions of clustering based on execution traces and clustering based on fault localization ranking to propose a new approach for clustering failures. They introduce two clustering techniques, the first of which we described in section II. As second technique, they use fault localization results of each failing test and all passing tests and compute pairwise similarity of the results. Then clusters are formed based on similarities. We set up a training phase to adapt their first clustering technique to our context. As a result, our execution trace profiling and fault localization metric differ from their work. In addition, we have added a new mechanism to select some representatives for each cluster.

There is more literature that focuses on parallelizing fault localization and debugging [28], [23], [5]. In our work, we want to find failures that happened due to the same underlying faults rather than debugging and finding the underlying faults. Thus, the purpose of our work is different.

Hsueh et al. [29] apply failure clustering in the context of graphical user interfaces. They instrument the code with the aid of developers that insert probing statements. They calculate the similarity and construct a tree. Then, considering the tree, they select some representative tests to start debugging. They suggest that just showing the tree to developers is beneficial. However, they do not decide about the cutting point for the clustering tree and do not explain clearly how they choose the representatives.

DiGiuseppe and Jones [30] use semantic concepts rather than the control-flow of the program to do failure clustering. They utilize latent-semantic-analysis to group filing executions based on their semantic intent. Their work differs from ours in their use of execution semantics as opposed to control-flow data. This approach would be beneficial when execution semantics or the used language are not similar, which is not the case in our case study.

Rogstad and Briand [21] cluster deviations in regression testing for an industrial database system. Since it is unlikely to find a general solution for every context, they use context specific profiles, namely test case specifications and database manipulations, to measure the similarity between failing executions. They use the Expectation Maximization algorithm [31] for clustering which requires a prediction for the maximum number of clusters. Their clustering method, feature sets and application context differ from our work.

Our paper is a large-scale industrial case study in the automotive industry. To the best of our knowledge, this is the first paper addressing the challenge of reduction of failure analysis time in the context of automotive CPSs for both SiL and HiL levels of testing. We are not aware of previous work that has utilized SiL-level execution profiles to analyze failing tests at the HiL level. The difference between SiL and HiL may be seen as a property specific to the domain of CPS. We do not think so. In fact, it is very similar to the difference between unit testing with mocks and testing of the integrated system with actual implementations. Thus, we believe that this idea is more general and can be applied in many domains.

V. CONCLUSION AND FUTURE WORK

One challenge in testing automotive CPSs is reducing failure analysis time. We built on the idea to cluster failing tests

to reduce the effort needed by testers and investigated its effectiveness in an industrial case study with ca. 850 KLOC. We focused on the SiL and HiL levels of testing. Results show that we can group failures based on their underlying faults with very high purity. The clustering tool can effectively reduce the effort needed by testers. Utilizing this tool, testers save more than 80% in failure analysis time, at least in our study.

We believe that our results add to the growing body of evidence about the potential usefulness of fault localization techniques if multiple faults are present. Complementing earlier work, our study indicates that these techniques may be more useful for organizing the debugging process than for actually locating faults. To the best of our knowledge, we are the first to provide this evidence with a large industrial case study for embedded automotive systems.

Our methodology is meant to be instantiated to a specific domain or class of applications. We have shown how to learn the relevant parameters (e.g., clustering method, distance measure) as a significant part of the initialization process. While we cannot report on a second study in a different domain here and clearly see the threats to external validity, we ourselves are sufficiently confident to repeat our own study in a different context.

In the future, we will specifically focus on HiL-level testing. We will add more hardware related information as a basis for clustering such as input/output signals. It will be also helpful for testers to understand the reasons behind failures. Therefore, discovering causal dependencies from execution traces would be a possible future addition to our work.

REFERENCES

[1] A. Pretschner, "Defect-Based Testing," *Dependable Software Systems Engineering*, pp. 224–245, 2015.

[2] E. Bringmann and A. Kr, "Model-Based Testing of Automotive Systems," *2008 International Conference on Software Testing, Verification, and Validation*, pp. 485–493, 2008.

[3] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. S. J. Sun, and B. W. B. Wang, "Automated support for classifying software failure reports," *25th International Conference on Software Engineering, 2003. Proceedings.*, vol. 6, pp. 465–475, 2003.

[4] J. a. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in Parallel," in *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA '07*, 2007, p. 16.

[5] W. Hogerle, F. Steimann, and M. Frenkel, "More debugging in parallel," *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, pp. 133–143, 2014.

[6] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982.

[7] L. Rokach and O. Maimon, "Chapter 15 Clustering methods," *The Data Mining and Knowledge Discovery Handbook*, p. 32, 2010.

[8] C. D. Manning, P. Raghavan, and H. Schütze, "Introduction to Information Retrieval," *2008*, vol. 1, no. c, p. 496, 2008.

[9] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[10] C. Liu and J. Han, "Failure proximity: A fault localization-based approach," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE-14. New York, NY, USA: ACM, 2006, pp. 46–56. [Online]. Available: http://doi.acm.org/10.1145/1181775.1181782

[11] T. Pang-Ning, M. Steinbach, and V. Kumar, *Introduction to data mining*, 2006.

[12] N. DiGiuseppe and J. a. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Software Engineering*, pp. 928–967, 2014.

[13] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967.

[14] A. S. Namin and S. Kakarla, "The use of mutation in testing experiments and its sensitivity to external threats," *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, p. 342, 2011.

[15] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the Accuracy of Fault Localization Techniques," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 76–87.

[16] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 402–411, 2005.

[17] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, no. 1973, p. 29, 2011.

[18] B. McCune, J. B. Grace, and D. L. Urban, *Analysis of Ecological Communities*, 2002.

[19] S. Saraçli, N. Dogan, and I. Dogan, "Comparison of hierarchical cluster analysis methods by cophenetic correlation," *Journal of Inequalities and Applications*, vol. 1, no. 203, pp. 1–8, 2013.

[20] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[21] E. Rogstad and L. C. Briand, "Clustering Deviations for Black Box Regression Testing of Database Applications," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 4–18, 2016.

[22] B. Baudry, F. Fleurey, Y. L. Traon, and Y. Le Traon, "Improving Test Suites for Efficient Fault Localization," *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pp. 82–91, 2006.

[23] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, 2013, p. 314.

[24] W. Masri, "Automated Fault Localization. Advances and Challenges," in *Advances in Computers*, 2015, vol. 99, pp. 103–156.

[25] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pp. 654–665, 2014.

[26] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 339–348. [Online]. Available: http://dl.acm.org/citation.cfm?id=381473.381509

[27] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, p. 195, 2004.

[28] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "Localizing software faults simultaneously," *Proceedings - International Conference on Quality Software*, pp. 367–376, 2009.

[29] C. H. Hsueh, Y. P. Cheng, and W. C. Pan, "Intrusive test automation with failed test case clustering," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2011, pp. 89–96.

[30] N. DiGiuseppe and J. A. Jones, "Concept-based Failure Clustering," *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 29:1–29:4, 2012.

[31] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum Likelihood from Incomplete Data via the EM Algorithm," *Journal of the Royal Statistical Society. Series B*, vol. 39, no. 1, pp. 1–38, 1977.