Toward Extended Change Types for Analyzing Software Faults

Billy Kidwell
Department of Computer Science
University of Kentucky
Lexington, KY
bill.kidwell@uky.edu

Jane Huffman Hayes
Department of Computer Science
University of Kentucky
Lexington, KY
hayes@cs.uky.edu

Allen P. Nikora
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA
allen.p.nikora@jpl.nasa.gov

***Abstract*—This research extends an existing source code change taxonomy that was designed to analyze change coupling. The extension expands change types related to statements in order to achieve more granular data about the type of statement that is changed. The extended taxonomy is evaluated to determine if it can be applied to software fault analysis. We found that the extended change types occur consistently and with high frequency in fault fixes for Eclipse 2.0 and 3.0. Faults were then clustered according to the source code changes and analyzed. We found that the types and sizes of clusters are highly correlated, indicating some consistency in the patterns of the fault fixes. Finally, we performed an initial investigation to determine whether faults in the same cluster have similar characteristics. Our results indicate that many of the change types can be used to characterize the type of fault that has been fixed. However, some of the change types obfuscate the true nature of the fix. Ideas for improving the taxonomy based on these findings are provided.**

***Index Terms*—Fault classification, change taxonomy, clustering, source code analysis.**

I. INTRODUCTION

Software fault classification schemes have been used to guide process improvement [1]–[3], prevent defects [4], improve software interface design [5], [6], improve testing [7]–[9] and analyze security breaches [10]. Despite the benefits of using a software fault classification scheme, it is not yet common practice in industry. Practitioners find that some faults are difficult to classify, and that training and discipline are necessary to get accurate results [11]. Tool support could improve the accuracy and efficiency of the software fault classification process.

This research investigates the extension and application of fine-grained source code changes to the analysis of software faults. Fluri et al. introduced ChangeDistiller, a tool that can identify the fine-grained source code changes from two versions of source code [12]. The algorithm and change taxonomy implemented in ChangeDistiller are designed to analyze change couplings [12], [13]. A version of ChangeDistiller is available under an open source license[1]. The change taxonomy consists of more than forty change types. Four of these change types identify the insert, update, delete, or re-ordering of a statement. In order to extend the taxonomy, we expand these four change types by appending the type of statement that was changed.

The first contribution of this research is an extension to the change taxonomy developed by Fluri and Gall [13] that allows the taxonomy to be applied to the analysis of software faults. We demonstrate that the new change types occur often in fault fixes for two versions of the Eclipse software project. We also find that the frequency of occurrence is correlated, indicating a consistency in the types of fixes that are applied to the two versions of the software.

A second contribution of the study is a technique to cluster faults according to the syntactic similarity of the fix in order to validate that the taxonomy. A vector of the extended change types and their frequency is used as input. Cosine similarity is used as the internal similarity measure. The resulting clusters occur consistently in two versions of Eclipse and provide groups of fault fixes with similar syntax. The clustering method chosen also reduces noise in the data by creating a single, low similarity cluster with data that does not match other clusters. This allows a random selection of faults from each cluster for manual analysis.

A third contribution of this study is the identification of limitations in the extended change taxonomy. We find that some clusters contain faults that are well characterized by the dominant change types for that cluster. For example, the faults that we randomly sampled in the condition expression change cluster were logic faults. On the other hand, some clusters contain faults where the change type obfuscates the primary characteristics of the fault. As an example, the update variable declaration change type does not differentiate between an update to a declaration where a variable is initialized with a method call and an update to a declaration where the variable is initialized with a constant. We report our findings and suggest future work to improve the change taxonomy in the context of software fault analysis.

A minor contribution of this work is additional data on the nature of software fault fixes. Pan et al. introduced bug fix patterns that cover 45.7%-63.3% of the total fault fix hunk pairs in seven open source java projects [14]. Their study found the frequency of these patterns to be surprisingly consistent, concluding that developers have difficulty with specific code situations at a highly consistent rate. Hamill and Goševa-Popstojanova reported that requirements faults and coding faults represent 33% of the total faults each [15]. Furthermore,

[1] https://bitbucket.org/sealuzh/tools-changedistiller/

they grouped projects by the number of releases and compared their results with three other studies. They conclude that coding faults are significant, that interactions between components cause problems, and that the remaining defect types are not major causes of problems and may be domain specific. Our investigation uncovered and validated a strong consistency in the syntax used to correct software faults in two major versions of Eclipse. The occurrence of particular syntactical change types and the patterns of syntax changes that were clustered both exhibit this consistency.

IEEE defines a software *fault* as an "incorrect step, process or data definition in a computer program" [16]. The term defect is used synonymously with fault by some researchers. A *fault* leads to a *failure* when the software does not perform to specifications. The term bug is ambiguous, and may refer to a *fault* or the resulting *failure*. We avoid the use of the term bug as much as possible, but it may be used to refer to the documentation of a fault in fault tracking databases such as Bugzilla[2].

The paper is organized as follows. Section 2 presents our motivation to improve software fault classification. Section 3 explains the research approach for this study. In Section 4 we present our validation of the extended change taxonomy. Threats to validity are presented in Section 5. Related work is discussed in Section 6. Future work is discussed in Section 7.

II. MOTIVATION

Despite multiple advantages that have been documented for software fault classification, the classification of faults is not a mainstream practice. Fault classification data could also be useful in software engineering research. For example, which types of faults are most easily predicted by fault prediction models? How does the prediction of specific fault types vary with different features? What is the relationship between the fault type, and the type of component where a fault may occur (aka Fault Link[17], [18])? Research questions such as these are all faced with a significant barrier to entry due to the lack of data sets with classified fault information.

Our proposed solution to this problem is a decision support system (DSS) that can aid a developer or researcher with the classification of software faults. This section describes our vision for the DSS and relates the current research to this effort.

The fault classification process begins when the developer commits a fix for a software fault. The abstract syntax tree for the source code before and after the fix is instantiated and compared. This comparison results in a collection of fine-grained source code changes that can be used as input to the system, as depicted in Fig 1. The extended change types in this research form a basis for the input to the decision support system.

The user is presented with a view to show the differences in the source code alongside suggestions for the most appropriate fault type. The fault type suggestions are provided by a machine learning algorithm that is trained from historical data. The resulting selection is stored and may be used as additional training data.

[2] http://www.bugzilla.org/

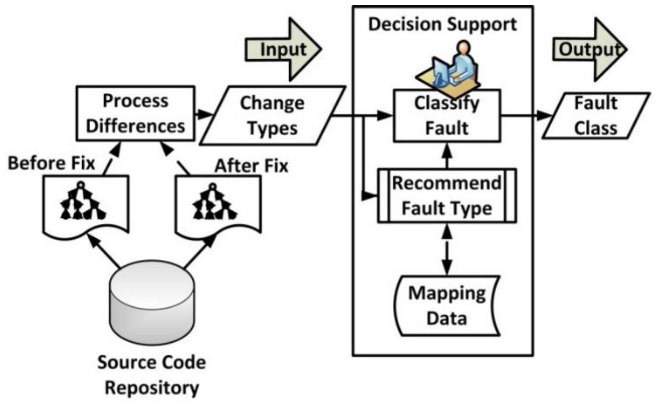


Fig. 1. A Decision Support System for Fault Classification

III. APPROACH

The approach is divided into data collection and clustering.

A. Data Collection

This study builds on the Eclipse fault data that was used by Krishnan et al. to evaluate change predictors in a software product line [19]. Eclipse 2.0 and Eclipse 3.0 were selected for this study. Each fault is processed according to a simple workflow. File revisions before and after each fault fix are retrieved from the CVS source code repository and stored locally.

The fine-grained source code changes are extracted for each pair of files using the ChangeDistiller tool [20]. Fluri et al. describe the change distilling process, where the abstract syntax tree of each revision of the source code are compared and source code changes are extracted [12]. These source code changes are stored in a database with all of the contextual information that is provided by ChangeDistiller. Contextual information includes the name and type of the source code entity that was changed with its location in the file, the parent entity of the changed entity, and the parent entity of before the source code change.

The contextual information collected by ChangeDistiller allows the extension of the *statement delete*, *statement insert*, *statement update*, and *statement ordering change* change types. We use the *changed entity* information available from the ChangeDistiller API to identify the type of statement that was altered, such as an *if statement* or *method invocation*. All of the information for each change is recorded in an SQL database and the extension is performed through the use of an SQL script. A database trigger is used to append the changed entity's type to the change type. This occurs for the statement insert, statement update, statement delete, and statement ordering change types. For example, a record with a change type of *statement insert* and a changed entity of *method invocation* will result in an extended change type of *statement insert method invocation.* We translate this value to *insert method call* for readability. Once the database is populated with all of the source code changes, a query is used to collect the type and count of source code changes that are recorded for each fault in the dataset.

B. Clustering

The input to the clustering process is a vector. The features of the vector are the extended change types. One hundred and one extended change types were present in the Eclipse 2.0 dataset and one hundred and nine change types were present for Eclipse 3.0. Space limitations do not allow the enumeration of all features, but the most significant features are shown in the legend of Fig 4. The extended change types are shown in bold.

The CLUTO clustering toolkit is used to perform clustering of the data [21]. CLUTO was selected based on its inclusion of cosine similarity as a distance measure and the visualization features that aid in the analysis of the clusters. CLUTO creates a hierarchical clustering solution when the repeated bisection approach is used [22]. The hierarchical solution provides views of the data at different levels of granularity, and in our case allows us to compare hierarchies in data from multiple datasets.

All documents are initially partitioned into two clusters. One of the clusters is selected and bisected. This process is repeated k-1 times to arrive at k clusters. CLUTO provides seven different criterion functions that can be used to guide the clustering process. A simple, greedy scheme is used to optimize the selected criterion function [23]. During multiple iterations of refinement, each instance in a cluster is visited in random order and moved to the cluster that improves the criterion function's value. This iterative refinement is repeated until no instances are moved. In order to avoid the selection of a local maximum or local minimum, the entire process is repeated multiple times and the best solution is selected. The default value of ten iterations was used in this study.

CLUTO offers multiple criterion functions that can be used in clustering. Our purpose in clustering is to group faults with similar syntactic changes. The I1 and I2 criterion functions maximize internal similarity, so we limited our evaluation to these criterion functions. I1 maximizes the sum of the average pairwise similarities between the instances in the cluster. I2 maximizes the similarity between each instance and the centroid of the cluster, similar to the k-means algorithm [21].

CLUTO provides metrics to aid in cluster analysis. For each cluster, the internal similarity (iSim) and external similarity (eSim) are reported, along with their standard deviations (iSDev and eSDev). The internal similarity is the average similarity between all objects of the cluster. An internal similarity near one represents a "tight" cluster. We focus our evaluation of clusters on the internal similarity since we are trying to group software fixes with similar syntax. The external similarity is the average similarity between the objects of each cluster with the rest of the objects. An external similarity near zero represents a cluster that is well-separated from other clusters in the data set. We report the external similarity but do not use it for evaluation.

CLUTO reports a number of features that account for the internal similarity of a particular cluster. These are referred to as *descriptive* features [21]. A percentage is provided with each feature. This allows us to make statements such as "Condition expression changes account for 94.7% of the similarity for instances in cluster 1." The descriptive features are used in this study to characterize and label each of the clusters and make a conjecture about the types of faults that belong to the group. Labeling of the clusters is entirely based on the statistical prominence of the features in the cluster, and not based on subjective evaluation of the results.

IV. VALIDATION

We perform four main tasks during our analysis of the extended change types. For the first task, we investigate the most common change types in fault fixes for Eclipse 2.0 and 3.0. We then compare the percentage occurrence of the top 12 change types in both versions of the software to determine if the occurrence is consistent. In the second task we evaluate the I1 and I2 criterion functions to determine the most appropriate criterion function for clustering the fault data. Next, we compare the clusters of Eclipse 2.0 and Eclipse 3.0 to determine whether there is consistency in the clustering of faults. Finally, we perform a manual inspection of a subset of the faults to investigate the usefulness of the clusters for analyzing faults. Although our manual inspection includes too few faults to make any strong conclusions, we identify several areas of improvement based on the evaluation.

A. Frequency of Change Types in Fault Fixes

In this section we evaluate the frequency of extended change types in software fault fixes as compared to the original change taxonomy. The top twelve change types that are extracted from fault fixes in Eclipse 2.0 and 3.0 are the same, and are presented in Table I with frequency of occurrence.

TABLE I. TOP TWELVE CHANGE TYPES IN FAULT FIXES

Change Type	Eclipse 2.0		Eclipse 3.0	
	Commits	**Percent**	**Commits**	**Percent**
Insert If *	1512	52.39%	3415	52.21%
Insert Method Call *	1391	48.20%	3039	46.46%
Insert Var Decl *	1145	39.67%	2637	40.31%
Statement Parent Chg	1098	38.05%	2555	39.06%
Add Functionality	979	33.92%	2205	33.71%
Update Method Call *	958	33.19%	2095	32.03%
Insert Assignment *	937	32.47%	2238	34.21%
Delete If *	934	32.36%	2239	34.23%
Delete Method Call *	861	29.83%	1883	28.79%
Insert Return *	777	26.92%	1750	26.75%
Update Var Decl *	734	25.43%	1850	28.28%
Cond Expr Change	731	25.33%	1853	28.33%

The first column indicates the change type. Change types that were introduced by our extension to the taxonomy are denoted by an asterisk (*). The second and fourth columns provide the number of commits that are associated with a fault fix that contained at least one instance of the change type for each version of the software. The third and fifth columns provide a percentage of the total number of commits that include the change type.

The total number of extended change types in this list provides evidence that the extended change types provide additional granularity that is useful in the analysis of software fault fixes. The change types occur with surprising consistency between the two versions. This led us to question whether the frequency between the two versions is consistent. The following hypotheses are used for investigation.

H_0: There is no significant correlation in the frequency of extended change types in Eclipse 2.0 and Eclipse 3.0 (α=0.05).

H_A: The frequency of extended change types in Eclipse 2.0 and Eclipse 3.0 are correlated (α=0.05).

The data is not normally distributed, so the non-parametric Wilcoxon signed rank test is performed to test the hypothesis. The test was performed against the number of commits for each extended change type in the dataset. The test indicates that there is no significant difference in the frequency of the change types, with a *p*-value of 0.0005. We reject *H0* in favor of the alternative and conclude that the occurrence of change types is consistent in these two versions of the software.

B. Evaluation of Criterion Functions

In order to proceed with the clustering and inspection of the faults, we must choose the most appropriate criterion function. We limit our selection to the I1 and I2 criterion functions, since these functions maximize the internal similarity of the clusters. Clustering is performed for fault data for Eclipse 2.0 and Eclipse 3.0. We repeat the clustering for all values of k from 2 to 20. The number of fault types in a fault taxonomy should be manageable and not too large [24]. Based on this recommendation, we expect there to be seven to ten fault types. We choose a broad range of numbers to be inclusive. We use the following hypotheses for investigation.

H_0: There is no difference in the mean internal similarity of clusters when using the I1 and I2 criterion functions (α=0.05).

H_A: The mean internal similarity of clusters when using the I1 criterion function is greater than the mean internal similarity of clusters when using the I2 criterion function (α=0.05).

The mean internal similarity for each of these methods is presented in Table II. The number of clusters, *k*, is shown in the first column. The remaining columns report the internal similarity for each method, for each version. A graph of these values for the Eclipse 2.0 dataset is presented in Fig. 2. A similar graph for Eclipse 3.0 is displayed in Fig. 3.

We perform a one-tail paired samples Wilcoxon signed rank test on the similarity data for I1 and I2 to evaluate the hypothesis. A paired t-test was considered, but the data does not pass a test for normality, and thus the non-parametric test is used. We perform the test independently for both versions of Eclipse. For Eclipse 2.0, the p-value = 3.815e-06 and for Eclipse 3.0, the p-value = 3.624e-05. In both cases we are able to reject the null hypothesis in favor of the alternate hypothesis.

Zhao and Karypis provide an analysis of document clustering solutions using the I1 and I2 criterion functions in their comparison of criterion functions [23]. In general, all criterion functions have different sensitivities based on the tightness of the clusters and the degree of balance in the resulting solution. Zhao and Karypis analyze the I1 and I2 functions to explain how the I1 criterion function can lead to several pure, tight clusters and a single large, poor quality cluster. This poor quality cluster is referred to as a "garbage collector" and results from the function's tendency to exclude peripheral documents from the pure clusters.

TABLE II. MEAN INTERNAL SIMILARITY

	Eclipse 2.0		**Eclipse 3.0**	
k	**I1**	**I2**	**I1**	**I2**
2	0.292	0.282	0.297	0.289
3	0.329	0.317	0.333	0.322
4	0.404	0.401	0.412	0.415
5	0.475	0.429	0.439	0.443
6	0.497	0.449	0.526	0.468
7	0.517	0.462	0.546	0.494
8	0.535	0.487	0.551	0.510
9	0.561	0.495	0.566	0.528
10	0.567	0.499	0.571	0.531
11	0.577	0.506	0.584	0.539
12	0.580	0.503	0.591	0.571
13	0.584	0.511	0.601	0.569
14	0.593	0.514	0.612	0.574
15	0.597	0.521	0.614	0.576
16	0.602	0.543	0.617	0.580
17	0.606	0.549	0.601	0.585
18	0.607	0.555	0.604	0.587
19	0.621	0.561	0.615	0.599
20	0.624	0.567	0.622	0.630

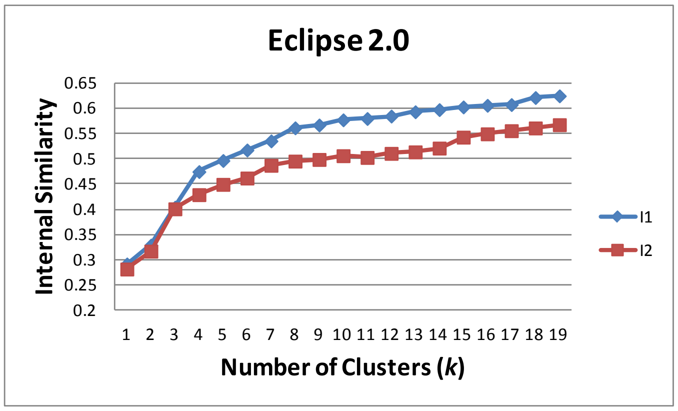


Fig. 2. Mean Internal Similarity for Eclipse 2.0

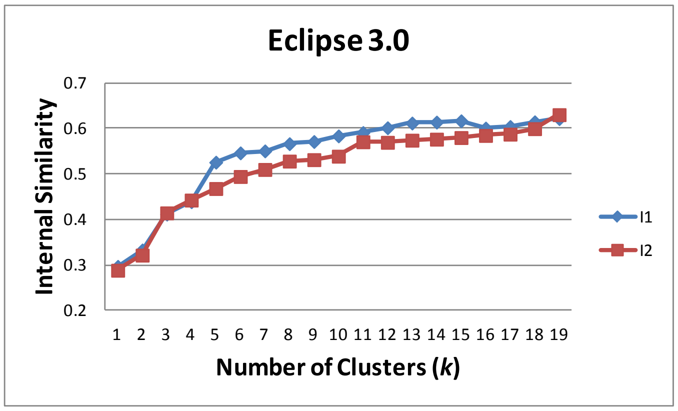


Fig. 3. Mean Internal Similarity for Eclipse 3.0

Zhao and Karypis provide an analysis of document clustering solutions using the I1 and I2 criterion functions in their comparison of criterion functions [23]. In general, all criterion functions have different sensitivities based on the tightness of the clusters and the degree of balance in the resulting solution. Zhao and Karypis analyze the I1 and I2 functions to explain how the I1 criterion function can lead to several pure, tight clusters and a single large, poor quality cluster. This poor quality cluster is referred to as a "garbage collector" and results from the function's tendency to exclude peripheral documents from the pure clusters.

Zhao and Karypis conclude that this property of the I1 criterion function may be useful in noisy data sets [23]. This helps explain the superiority of the I1 criterion function in our experiment, and suggests that more analysis of the instances in the "garbage collector" may help refine analysis for less-frequent faults.

C. Consistency of Clusters for Eclipse 2.0 and 3.0

In this section we analyze the consistency of the clustered fault fixes for Eclipse 2.0 and Eclipse 3.0 at k=10. We choose this value of *k* due to similarities in the descriptive features across the two versions of Eclipse. The groups appear to stabilize at this value of *k*. Ten is also on the high end of the number of fault classifications that are recommended by best practices [24]. We label each cluster based on the descriptive features reported by CLUTO. Since the top five descriptive features of each cluster are reported, we use a threshold value of 10% for determining whether a feature is significant.

The cluster features, sizes, and similarities are reported in Table III. The first row reports on the clusters that are described by the update of a variable declaration. In Eclipse 2.0, this cluster included 94 faults, while in Eclipse 3.0 the cluster includes 261. The last row of the table contains totals for the number of faults in each data set.

TABLE III. COMPARISON OF CLUSTERED FAULTS

	Eclipse 2.0		**Eclipse 3.0**	
Cluster (Descriptive Features)	**Size**	**iSim**	**Size**	**iSim**
Upd Var Decl	94	0.789	261	0.724
Cond Expr Chg	139	0.708	244	0.834
Add Func	132	0.678	441	0.599
Upd Method Call	266	0.663	494	0.654
Ins If + Ins Return	164	0.58	0	-
Ins If + Stmt Parent Chg	446	0.57	908	0.584
Ins Meth Call	434	0.566	756	0.582
Del Meth Call + Ins Meth Call	279	0.525	669	0.513
Ins If + Ins Meth Call + Ins Var Decl	554	0.504	1049	0.515
Ins Assign + Upd Assign	376	0.084	706	0.128
Ins Assign + Ins If	0	-	567	0.579
Total	**2884**		**6095**	

Notice that Eclipse 2.0 has a cluster described by the insertion of *if* and *return* statements, while Eclipse 3.0 has a cluster that is described by the insertion of *assignment* and *if* statements. In order to compare the clustering solutions, we treat these as empty clusters in the versions where they do not occur. We use the following hypotheses for investigation.

H_0: There is no significant correlation in the clustering solutions of Eclipse 2.0 and Eclipse 3.0 at k=10 (α=0.05).

H_A: The clustering solutions of Eclipse 2.0 and Eclipse 3.0 at k=10 are correlated (α=0.05).

To test the hypothesis, Pearson's correlation coefficient is calculated. A Shapiro-Wilk test for normality was performed to verify that the data is normally distributed. The value of *r* for the data is 0.778, with a *p*-value = 0.004, allowing us to reject the null hypothesis and conclude that the cluster types and sizes are correlated.

This correlation in the patterns of the faults is consistent with the work completed by Pan et al. in their analysis of bug fix patterns [14]. Our hope is that the syntax of the fix can be used to determine a fault type. Fault taxonomies are sometimes utilized to measure where faults are injected for process improvement. Chillerage et al. demonstrated this use with ODC [1]. If the syntax of the fix is an indicator of the fault type, this consistency could also support the findings of Hamill and Goševa-Popstojanova [15]. They found consistency in where faults were introduced across multiple studies, and concluded that coding faults were a significant source of faults.

D. Manual Inspection of Faults in Each Cluster

In this section we present clustering results on Eclipse 2.0 fault fixes using the I1 criterion function and setting *k*=10. Only Eclipse 2.0 is considered in this section. The Eclipse 2.0 dataset consists of 101 fine-grained source code change types after expanding statement insert, update, delete, and ordering change types and eliminating changes to comments and source code documentation. There are 2884 faults in the dataset with Java source code changes. Faults with zero Java source code changes, e.g., those requiring only changes to properties or xml configuration files, are not included in the analysis. CLUTO reports a number of metrics for the clusters. These metrics are presented in Table IV.

TABLE IV. CLUSTER SUMMARY

Cluster Id	**Size**	**iSim**	**iSDev**	**eSim**	**eSDev**
0	94	0.789	0.124	0.077	0.052
1	139	0.708	0.134	0.112	0.073
2	132	0.678	0.125	0.129	0.058
3	266	0.663	0.136	0.118	0.069
4	164	0.58	0.084	0.212	0.073
5	446	0.57	0.093	0.203	0.065
6	434	0.566	0.091	0.208	0.066
7	279	0.525	0.09	0.207	0.084
8	554	0.504	0.082	0.246	0.059
9	376	0.084	0.057	0.083	0.081

The CLUTO manual provides a full description of these metrics [21]. A summary is presented here. The Cluster Id is a zero-based integer assigned to each cluster. The Size is the number of faults that were assigned to the cluster. The column labeled iSim is the mean internal similarity of the faults in the cluster. The column labeled iSDev is the standard deviation of the mean internal similarities. Similarly, the eSim column is the mean similarity of the faults in the cluster with the faults that are not in the cluster, or the external similarity. The eSDev column is the standard deviation of the mean external similarity for the faults in the cluster. The clusters are ranked by subtracting the external similarity from the internal similarity and arranging them in decreasing order. This positions tight, distinct clusters at the top of the list.

CLUTO also reports descriptive and discriminating features for each cluster. Descriptive features are reported with the feature name and a percentage score. The percentage indicates the amount of similarity in this cluster that can be attributed to the descriptive feature. Likewise, discriminating features report a percentage that describes the amount of the dissimilarity with other clusters that can be explained by the discriminating feature [21].

The CLUTO toolset provides tools to visualize clustering results [21]. A modified version of the cluster plot visualization for the results that we manually analyzed is presented in Fig. 4. The columns in the visualization are the clusters, with the size of each cluster in parentheses. The tree structure aids in understanding the relationships between clusters. For example, cluster 6 and 7 are very similar clusters, and contain similar source code changes. The rows of the visualization provide a subset of the 101 source code changes that were used as features during the clustering process. The darkness of the cells is based on the intensity of the feature within each cluster. For example, in the first column we see that cluster 5 is described by the *statement parent change* and *insert if statement* change types. The label for descriptive features is repeated to the left of each occurrence. As an example, Cluster 1, on the far right of the illustration, is described by conditional expression changes (COND EXPR CHG).

For each cluster we present internal clustering metrics, features that explain the clusters, and then conduct a manual inspection of five to eight faults. We randomly selected the faults that were manually inspected from each of the clusters. The fault reports for these faults are available on the Eclipse foundation Bugzilla web site.[3] A description of our manual assessment of each individual fault is available online for interested readers (http://selab.netlab.uky.edu/Kidwell-Hayes-Eclipse_2-0-Fault-Inspection.pdf).

1) Cluster 0 – Update Variable Declaration

Cluster 0 is the tightest and smallest cluster in the selected solution. The *update variable declaration* change type explains over 98% of the similarity of the faults in the cluster. We expect faults in this cluster to represent faults where a variable is either uninitialized or incorrectly initialized.

Two of the five faults in this category fell in this expected category (10483 and 16828). In Bug 11110, a condition expression change is edited to check for null references. A portion of the change is provided in Fig. 5. The change requires the intermediate variable **window** to be added on the new line

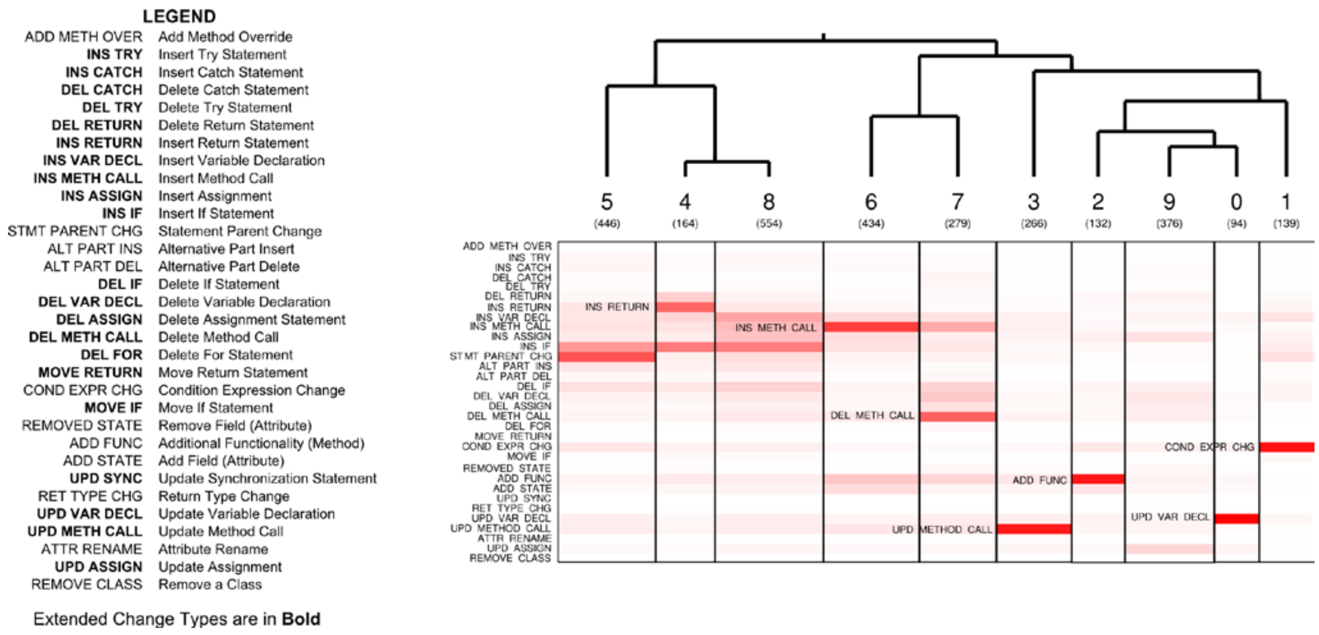
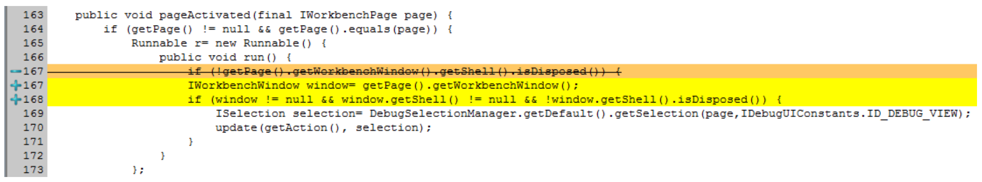


Fig. 4. Visualization of Clusters for Eclipse 2.0 where k=10 using I1 criterion function.



```
 163      public void pageActivated(final IWorkbenchPage page) {
 164          if (getPage() != null && getPage().equals(page)) {
 165              Runnable r= new Runnable() {
 166                  public void run() {
-167                      ~~if (!getPage().getWorkbenchWindow().getShell().isDisposed()) {~~
+167                      IWorkbenchWindow window= getPage().getWorkbenchWindow();
+168                      if (window != null && window.getShell() != null && !window.getShell().isDisposed()) {
 169                          ISelection selection= DebugSelectionManager.getDefault().getSelection(page,IDebugUIConstants.ID_DEBUG_VIEW);
 170                          update(getAction(), selection);
 171                      }
 172                  }
 173              };
```

Fig. 5. Fault fix to check for Null Pointer Exception in Bug 11110

167. The window variable is used in the new condition on the new line for 168. This change is obfuscated because it occurs in a variable declaration for an anonymous class, an instance of **Runnable** that is declared on line 165.

The fix for Bug 18923 includes variable name changes that cause this fault to belong to this cluster, but do not characterize the fault. Bug 23824 appears to be an interface fault due to a cast or incorrect argument. In this case the call occurs in a variable declaration and is not detected.

The unexpected faults in this cluster indicate that updates to variable declarations can hide important syntactical details that are necessary for fault analysis.

2) Cluster 1 – Conditional Expression Changes

The presence of a *conditional expression change* in faults that belong to Cluster 1 explain 94.7% of the similarity values for these items. Simple logic errors are expected to belong to this cluster. More complex algorithmic faults that require extensive logic changes may also be represented here. Four of the five faults we inspected were logic errors, while the fix for Bug 18787 was a more complex logic change.

Logic problems are a common cause for software faults and the source code changes are often small and contained. This results in faults that are easily characterized by these changes.

2) Cluster 2 – Additional Functionality

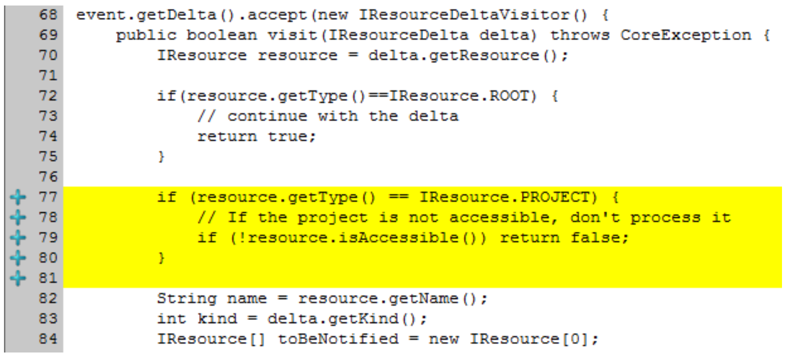
The similarity in Cluster 2 is explained by the addition of one or more new methods (95.2%). We expect faults in this cluster to include additions of new features and functionality. We investigated six faults in this cluster.

Five of the faults met our expectations for this category. The sixth, Bug 15513, was fixed by overriding a method of the base class. This type of fault logically belongs to the group, so we add it as an additional consideration for this cluster.

3) Cluster 3 – Update Method Call

The faults in Cluster 3 are characterized by the update of a method call (95.4%). The faults in this cluster are expected to be interface faults that involve the incorrect use of methods. Five faults in this cluster were manually inspected.

The most unexpected finding in this cluster was the impact of anonymous classes. Three of the five faults that we manually inspected in this cluster had methods updated where the argument was an anonymous class. The changes to the anonymous class were logic changes. An example is shown in Fig. 6 from Bug # 20421. Similar to the anonymous class that was encountered in cluster 0, the true nature of the change was hidden. The addition of lines 77-81 check a precondition and return false if it is not met. However, it occurs within the anonymous class that is passed to the accept method on line 68.



```
68  event.getDelta().accept(new IResourceDeltaVisitor() {
69      public boolean visit(IResourceDelta delta) throws CoreException {
70          IResource resource = delta.getResource();
71
72          if(resource.getType()==IResource.ROOT) {
73              // continue with the delta
74              return true;
75          }
76
77          if (resource.getType() == IResource.PROJECT) {
78              // If the project is not accessible, don't process it
79              if (!resource.isAccessible()) return false;
80          }
81
82          String name = resource.getName();
83          int kind = delta.getKind();
84          IResource[] toBeNotified = new IResource[0];
```

Fig. 6. Additional condition check and exit within an anonymous class from Bug #20421

4) Cluster 4 – Insert If and Return Statements

Cluster 4 is the first cluster that is primarily explained by two features. The addition of a *return* statement explains 47.3% of the similarity and the addition of an *if* statement explains 36.4% of the similarity. We expect simple faults in this cluster to be checking faults. More complex faults with multiple instances of *if* statements and/or multiple instances of *return* statements may represent more complex logic faults or algorithmic faults. Five faults were manually inspected in this cluster and all of them met expectations. Two of the five were checking faults. Two of the fixes were minor logic changes. Bug 14061 had extensive changes to the program logic.

5) Cluster 5 – Insert If Statement and Parent Change

The faults in Cluster 5 are characterized by a *statement parent change* (63.1%) and the insertion of one or more *if* statements (22.7%). Similar to Cluster 4, we expect logic faults that range from checking faults to more complex logic faults. We manually inspected five faults in this cluster. Bug 14025 was the only fault in this cluster that did not meet our expectations. The change required logic changes, but included new functionality as well.

6) Cluster 6 – Insert Method Call

The similarity of faults in Cluster 6 is explained primarily through the insertion of method calls (78.5%). A small part of the similarity is explained due to the addition of methods (6.7%). We expect this cluster to contain faults due to missing functionality and misuse of methods. Seven faults from this cluster were manually inspected.

Three of the faults addressed missing functionality (10823, 11308, and 18067). Three of the faults were interface faults (17490, 17981, and 21654). The fix for Bug 16160 was unexpected in this cluster. It repaired a dependency problem. Additional analysis may be necessary to automatically classify issues of this type.

7) Cluster 7 – Delete Method Call

The faults in Cluster 7 are explained by the removal of method calls (56.6%) and partially explained by the insertion of new method calls (16.2%). We expect the faults in this cluster to include the removal of extraneous code and moving method calls to new locations. Since the changes imply restructuring of the code, functional defects and refactoring may also be present in these faults. Five faults from this cluster were manually inspected. Three of the five fell into the category of extraneous method calls or functionality (14800, 16051, and 16445). The other two fixes in this cluster involved extensive changes to current program flow, and included refactoring.

8) Cluster 8 – Insert If, Variable Declaration, Method Calls and Assignments

The faults in Cluster 8 are explained by the insertion of if statements (40.3%), variable declarations (19.5%), method calls (11.1%), and assignment statements (9.0%). Given the nature of these changes, the faults in this cluster are expected to be algorithmic or functional changes to behavior. Seven faults in this cluster were manually inspected.

Five of the faults that were manually inspected fall into this broad category of changes. Bug 15506 was fixed by adding a busy indicator. The CVS commit for Bug 19270 included changes for another bug, which makes automated analysis challenging. The large number of faults that met our expectations in this cluster is encouraging, but further analysis is needed.

9) Cluster 9 – Garbage Collector

As mentioned previously, the last cluster acts as a "garbage collector" when the I1 criterion function is used. The descriptive features for this cluster were an update to an assignment (24.6%), addition of an assignment (12.7%), removal of a variable declaration (8.0%), update of a return statement (6.6%), and removal of a function (6.5%). The variation in change types and the scores for each feature support previous findings about the nature of the last cluster when I1 is used [23].

We expect this cluster to have varied faults that are either uncommon or simple faults that are obfuscated by implementation details. These may represent a set of faults for which automated classification is not possible or is not warranted due to their infrequent nature. A total of eight faults from this cluster were manually inspected.

There was no discernible pattern to these changes. Some of the changes were large, while others were small and infrequent. It is important to note that the fix for Bug 16027 includes some changes that were hidden because they were part of a return statement.

V. Threats to Validity

Wohlin et al. describe four areas where the validity of the results may be threatened [25], we discuss threats in each of these areas.

Conclusion validity concerns the statistical significance of the result. CLUTO allows the clustering of data to be repeated a number of times to avoid incorrect results due to a local minimum or maximum. We used a value of ten for the number of iterations, commonly used in clustering experiments with this tool [22], [23]. We undertook statistical analysis to test our hypotheses, and checked the assumptions of the tests that were used.

Internal validity is concerned with our ability to correctly measure the influence of the independent variables on the dependent variables and the elimination of possible confounding variables. We use publicly available data to construct our dataset. In addition, we are in the process of repeating the experiment on multiple versions of the Eclipse software. Initial results indicate that our clustering results are consistent across versions.

Construct validity refers to how well the independent and dependent variables in the study measure what is intended. It is difficult to measure the correctness of clustering when a correct answer is unknown. Purity and entropy measures require a classified dataset, and the manual classification of faults after the fact is time consuming and error prone.

External validity refers to the ability to generalize the results of the study. This paper presents early work that requires additional validation before generalization is possible. To achieve results that can be generalized we will extend the study with multiple versions of Eclipse and other software systems. The study also depends on tools that are currently available only for Java source code. We cannot claim that our results can be generalized outside of this particular dataset.

Our manual inspections of faults within each cluster are subjective. In order to obtain statistically valid results, independent reviewers should be used to classify the faults manually and inter-rater agreement should be measured. The number of faults inspected should also be increased to a statistically significant sample. Despite these limitations, the manual inspection in this study did find specific issues that can be used to guide improvement to the change taxonomy.

VI. Related Work

DeMillo and Mathur present a syntactical classification of software faults [8]. DeMillo and Mathur define the notion of a syntactic transform that captures the difference in two versions of the software's abstract syntax tree. The classification is automatable, but results in a large number of software fault types. Our research was motivated by this approach, and the idea that "syntax is the carrier of semantics" [8]. DeMillo and Mathur validate their work by classifying 291 software faults from TeX. Faults are classified in a hierarchy to handle cases where there are multiple changes involved in a single fix. The change types used by DeMillo and Mathur were designed for Pascal, and include changes more granular than statement level changes (e.g. wrong operator).

Pan, Kim, and Whitehead describe 27 automatically extractable bug fix patterns for Java source code. A bug fix pattern describes a syntactical pattern that occurs as one part of a software fault fix, such as changing the parameters of a method call [14]. Pan et al. consider a change to the source code as a collection of "hunks," and refer to the hunk in the faulty and fixed version of the code as a "hunk pair." The bug fix patterns cover 45.7%-63.3% of the total fault fix hunk pairs in seven open source java projects [14]. Pan et al. found the frequency of these patterns to be surprisingly consistent, concluding that developers have difficulty with specific code situations at a highly consistent rate. Our research has similarities, but differs in two major ways. First, our work does not look for pre-defined patterns of changes. The extended change taxonomy can be used as a basis to do this, but additional applications are possible. Secondly, we look at a fault fix as a whole, rather than individual "hunk pairs" in the fault fix. This is an important distinction when the fault that is being examined is a large, complex change.

Thung et al. categorize defects into three super-categories of ODC defect types, control and data flow, structural, and non-functional [26]. Text from the bug reports is pre-processed to extract features based on the bug report. The code is pre-processed to extract counts for additions and deletions of nodes in the abstract syntax tree. Thung et al. empirically validated their approach on 500 defects from three software systems, with a 77.8% average accuracy using a SVM multiclass classification algorithm [26]. Our method currently focuses on source code changes alone. Our study builds on the change types that are extracted from the Evolizer toolset, and thus uses a more sophisticated algorithm for the comparison of abstract syntax trees [12], [20].

VII. Conclusions

In this study, we extend the change taxonomy developed by Fluri and Gall [13], as implemented in ChangeDistiller [12], [20], and evaluate its use for analyzing software fault fixes. First, we extend the change taxonomy by expanding *statement delete*, *statement insert*, *statement update*, and *statement ordering change* change types to include the type of statement that was changed. We record the occurrence of these change types in two versions of Eclipse. We find that the extended change types occur more frequently than the original change types, and that their occurrence in the two versions is consistent.

In order to further validate the extended change types, and their patterns of occurrence, the CLUTO clustering toolkit is used to cluster the fault fixes. Using the repeated bisection clustering method and the cosine similarity, the I1 criterion function performs better than the I2 criterion function with respect to the average internal similarity of the clusters in the resulting solution. The ability of I1 to create tight clusters and one cluster that acts as a "garbage collector" in a noisy data set aids the investigation [23].

The results of clustering where k=10 are analyzed. The similarity of the cluster is explained by one to four features that are shared by the faults in the cluster. These descriptive features are used to automatically label the cluster. The clusters for Eclipse 2.0 and 3.0 and their sizes were compared. We found that the occurrence and size of the clusters were correlated, indicating that the clustering of these change types is consistent in these two versions of the software.

A subjective analysis of a subset of faults in each cluster provides guidance on the types of faults characterized by different source code change types. Many fault fixes are in agreement with our expectations based on the syntactical changes that were made to the fault. For example, faults fixed with changes to condition expressions that are inspected in this study are in line with expectations.

Several of the faults that were inspected exposed limitations in the taxonomy. ChangeDistiller stops the comparison of the abstract syntax trees at the statement level due to its intent in analyzing change couplings. As a result, update changes to variable declarations, assignments, or return statements do not provide the granularity necessary for fault analysis. We saw a surprising number of problems with anonymous classes as method parameters, and within variable declarations, that also require more granular information about the change. These findings indicate that we must extend the comparison beyond differences in statements, to differences in arguments and expressions. Based on our results, we make the case that is necessary to have a taxonomy that can indicate changes to operators and literals in addition to structure.

Faults that fix dependency problems provide additional challenges. Adding change types for the addition and removal of import statements may aid in this effort, but more sophisticated dependency analysis for build and run-time dependencies may be necessary.

We encountered a number of common software repository mining problems during our manual inspection. Code refactoring that is included in a commit for a bug fix can make analysis difficult. A simple change, such as renaming a variable for readability, should be handled at the semantic level of analysis. More complex refactoring changes will still make automated analysis difficult. Developers sometimes include multiple bug fixes in a single commit, as evidenced by Bug #19270. Bug #18468 was mislabeled as Bug #18486, which can be problematic when bug database information is cross-referenced with the syntactical changes.

We conclude that the current taxonomy may be helpful for analyzing a subset of fault types, but that the taxonomy must be further extended for syntactical fault analysis. Improving the change taxonomy and broadening the validation of the results are the focus of our future work.

VIII. Future Work

Our first step to extend this study will be to expand the change taxonomy to provide more granular information about source code changes that occur within a statement. *Update Statement* changes, even in their expanded form, can vary widely for variable declarations, assignments, and return statements. We must also account for anonymous classes when used within variable declarations and as method parameters.

In future studies we will expand this work to include additional datasets from subsequent versions of the Eclipse software. In particular, we would like to study the results for Eclipse version 3.3 (Europa) through 3.6 (Helios). This will provide data on four versions of Eclipse over a four year period. We also plan to compare the change characteristics of fault fixes with those of other changes, such as new features and refactoring changes.

Automated or semi-automated fault classification opens the door to large scale studies on software faults in open source systems that are currently cost prohibitive. In future work we hope to apply our fault classification technique to study *fault links*. Hayes et al. define a fault link as "a relationship between the type of module being developed or changed and the fault type" [17].

In a case study with the Apache web server and the Mozilla web browser, Hayes et al. found evidence that suggests there is a relationship between the module type and the fault type [17]. In a second study, Hayes et al. empirically validated that the application of fault link data to code inspections can improve the number of faults found and improve the detection of faults that are difficult to locate [18]. Fault links have potential applications in many verification and validation (V&V) techniques. The application of automated fault classification and automated module classification will provide access to significantly more data for the exploration of fault links.

Acknowledgment

This research is partially sponsored by NASA under grant NNG05GQ58G.

References

[1] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, “Orthogonal Defect Classification-A Concept for In-Process Measurements,” *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992.

[2] C. B. Seaman, F. Shull, M. Regardie, D. Elbert, R. L. Feldmann, Y. Guo, and S. Godfrey, “Defect categorization: making use of a decade of widely varying historical data,” in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, Kaiserslautern, Germany, 2008, pp. 149–157.

[3] M. Leszak, D. E. Perry, and D. Stoll, “A Case Study in Root Cause Defect Analysis,” in *Software Engineering, International Conference on*, Los Alamitos, CA, USA, 2000, p. 428.

[4] W. D. Yu, “A software fault prevention approach in coding and root cause analysis,” *Bell Labs Technical Journal*, vol. 3, no. 2, pp. 3–21, 1998.

[5] D. E. Perry and W. M. Evangelist, “An Empirical Study of Software Interface Faults,” 1985.

[6] D. E. Perry and W. M. Evangelist, “An Empirical Study of Software Interface Faults - An Update,” in *Proceedings of the Twentieth Annual Hawaii International Conference on System Sciences.*, Hawaii, 1987, vol. II, pp. 113–126.

[7] B. Beizer, *Software Testing Techniques, 2nd Edition*, 2 Sub. International Thomson Computer Press, 1990.

[8] R. A. Demillo and A. P. Mathur, “A Grammar Based Fault Classification Scheme and its Application to the Classification of the Errors of TEX,” Software Engineering Research Center; and Department of Computer Sciences; Purdue University, Technical Report, 1995.

[9] N. Li, Z. Li, and X. Sun, “Classification of Software Defects Detected by Black-box Testing: An Empirical Study,” in *Proceedings of 2010 Second World Congress on Software Engineering (WCSE 2010)*, Wuhan, China, 2010, pp. 234–240.

[10] W. Du and A. P. Mathur, “Categorization of software errors that led to security breaches,” *In Proceedings of the 21st National Information Systems Security Conference*, 1998.

[11] A. A. Shenvi, “Defect prevention with orthogonal defect classification,” in *Proceeding of the 2nd annual conference on India software engineering conference*, Pune, India, 2009, pp. 83–88.

[12] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, “Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction,” *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[13] B. Fluri and H. C. Gall, “Classifying Change Types for Qualifying Change Couplings,” in *14th IEEE International Conference on Program Comprehension (ICPC’06)*, Athens, Greece, 2006, pp. 35–45.

[14] K. Pan, S. Kim, and E. Whitehead, “Toward an understanding of bug fix patterns,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, Jun. 2009.

[15] M. Hamill and K. Goseva-Popstojanova, “Common Trends in Software Fault and Failure Data,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 484–496, 2009.

[16] “IEEE Standard Glossary of Software Engineering Terminology,” IEEE Computer Society, IEEE Std 610.12-1990, Dec. 1990.

[17] W. Farrr, “Software reliability modeling survey,” in *Handbook of software reliability engineering*, 1996, pp. 71–117.

[18] C. W. Holsappe, “DSS Architecture and Types,” in *Handbook on Decision Support Systems 1*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[19] S. Krishnan, C. Strasburg, R. R. Lutz, and K. Goševa-Popstojanova, “Are change metrics good predictors for an evolving software product line?,” in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, New York, NY, USA, 2011, pp. 7:1–7:10.

[20] H. C. Gall, B. Fluri, and M. Pinzger, “Change Analysis with Evolizer and ChangeDistiller,” *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[21] G. Karypis, “CLUTO: A Clustering Toolkit,” University of Minnesota, Department of Computer Science, Minneapolis, MN, Technical Report #02-017, Nov. 2003.

[22] Y. Zhao and G. Karypis, “Evaluation of hierarchical clustering algorithms for document datasets,” in *Proceedings of the eleventh international conference on Information and knowledge management*, New York, NY, USA, 2002, pp. 515–524.

[23] Y. Zhao and G. Karypis, “Empirical and Theoretical Comparisons of Selected Criterion Functions for Document Clustering,” *Mach. Learn.*, vol. 55, no. 3, pp. 311–331, Jun. 2004.

[24] B. G. Freimut, “Developing and Using Defect Classification Schemes,” Fraunhofer IESE, IESE-Report 072.01/E, Sep. 2001.

[25] C. Wohlin, P. Runeson, and M. Höst, *Experimentation in Software Engineering: An Introduction*, 1st ed. Springer, 1999.

[26] F. Thung, D. Lo, and L. Jiang, “Automatic Defect Categorization,” in *Reverse Engineering, Working Conference on*, Los Alamitos, CA, USA, 2012, vol. 0, pp. 205–214.

[27] J. H. Hayes, I. R. C.M., V. K. Surisetty, and A. Andrews, “Fault Links: Exploring the Relationship Between Module and Fault Types,” in *Dependable Computing - EDCC 2005*, 2005, pp. 415–434.

[28] J. H. Hayes, I. R. Chemannoor, and E. A. Holbrook, “Improved code defect detection with fault links,” *Software Testing, Verification and Reliability*, vol. 21, no. 4, pp. 299–325, Dec. 2011.