Mining Object-Oriented Software Execution Traces to Discover Patterns for Automated Testing

Obaid ur Rehman Malik
Mining Object-Oriented Software Execution Traces to Discover Patterns for Automated Testing

Sammanfattning
Ingen svensk sammanfattning finns då denna uppsats skrivits av en engelskspråkig student. Se "Abstract" för mer detaljer (på engelska).
Summary

With the evolution of new software technologies, the requirements for automated testing are becoming more and more stringent. With increasing size of software projects, manual testing is becoming less efficient. For automated testing one of the most important question is, what to focus upon while testing?

For a large number of functions along with large number of possible call sequences, it is very hard to generate test cases that cover all possible paths of control flow. By finding patterns in the calling sequences we will be able to identify more defects by focusing our testing efforts on those patterns.

In this paper, we have described our work on tracing call sequences using Aspect Oriented Programming methodology and discovering those patterns in call sequences using data mining techniques.
Acknowledgments

I would like to thank from the bottom of my heart Dr. Robert Feldt and Richard Torkar for giving me the opportunity and guiding me at every step of this research; to Thomas Gil for his support in issues regarding AspectDNG; to Zeeshan Ejaz for providing help in modifying AspectDNG and cross checking the obtained results.

I thank Allah Almighty for giving me the strength and courage to do what at first seemed to be an impossible task. Finally I would like to thank my parents and my fiancé for believing in me and continuously pushing me in achieving the best.
Innehållsförteckning

Sammanfattning ................................................................. i
Summary ........................................................................... ii
Acknowledgments .......................................................... 3
Innehållsförteckning ......................................................... 4
1. Introduction ................................................................. 5
2. Background ................................................................. 6
   2.1 Aspect Oriented Programming ................................ 6
   2.2 Data Mining and Clustering .................................... 7
3. Experimental Setup/Methodology .............................. 7
   3.1 Tracing OO Software Execution with AOP ............ 7
   3.2 Mining OO Execution Traces Data ......................... 8
     3.2.1 Distance Computation .................................. 8
     3.2.2 Data Transformation ................................... 10
     3.2.3 Discovery of similar patterns ......................... 11
4. Results ........................................................................ 12
5. Discussion ..................................................................... 14
6. Conclusion ..................................................................... 16
7. Future Work .............................................................. 16
8. References ..................................................................... 17
Abstract

With the evolution of new software technologies, the requirements for automated testing are becoming more and more stringent. With increasing size of software projects, manual testing is becoming less efficient. For automated testing one of the most important questions is, what to focus upon while testing?

For a large number of functions along with large number of possible call sequences, it is very hard to generate test cases that cover all possible paths of control flow. By finding patterns in the calling sequences we will be able to identify more defects by focusing our testing efforts on those patterns.

In this paper, we have described our work on tracing call sequences using Aspect Oriented Programming methodology and discovering those patterns in call sequences using data mining techniques.

1. Introduction

Testing a software system has always been a challenging task. It is indeed difficult to cover all the aspects of any given software system. If we are able to get all the call sequences of a software system then we can put more focus on the sequences that have been used the most while testing and this technique can help in the generation of automated test cases.

Some work has already been done on using call sequences for automated testing of object oriented software, like in Buy et al's [8] work where they use data flow analysis, symbolic execution and automated deduction to produce sequences of method invocations of class under test. But generation of call sequences was not fully successful in every attempt and there have been no efforts to analyze those call sequences themselves. We have adopted a different approach towards finding the control flow paths, i.e. aspect oriented programming (AOP) which provides a reliable framework for tracing call sequences. Moreover we attempt to discover frequent and similar paths in the call sequences, which can help to minimize the required testing effort.

Aspect oriented programming is currently focused on software design implementation and its verification. A very limited effort has been put into software testing despite of its acknowledged importance, like the work by Jianjun Zhao [3] on unit testing. Our scope in this paper is control flows and call sequence analysis.

We understand the importance of discovering patterns in call sequences because we know that in object oriented programs; an object can be in different states and behave differently on the basis of the sequence of function calls [8].

Moreover we know that ensuring that a software system is completely free of defects is a very cost consuming process and reducing the number of identified bugs (faced by users) is a big leap towards software quality (defect based software quality measurement). Also it might be possible, with the help of call sequences patterns, to devise intelligent test cases that summarize the call sequence and hence we test more with lesser test cases.

This paper focuses on object oriented programming paradigm only. Object oriented programming is one of the modern programming methodologies, which belong to the group of high-level languages. Microsoft .NET is a platform for object oriented programming in languages such as in C#, VB.NET, J#, VC.NET etc, and is selected as platform on which we will be working on for this research.

Our work describes three main steps in reaching the final goal (i.e. discovering patterns in call sequences). (1) Getting the software execution traces of a .NET assembly, (2) discussion on comparing two call sequences and computing the difference between them using edit distance algorithm (an algorithm we create by modifying an existing algorithm called
“approximate string matching algorithm”) which is required in distance based clustering for (3) mining these traces to discover patterns of similar call sequences.

The rest of the paper is structured into different sections. Section 2 starts with the related work that has already been done in this area and some of the concepts those are required in order to go through the rest of the material. Section 3 presents the experimental setup that is used to get the patterns in call sequences. Section 4 contains the results and our findings. Section 5 discusses the results and summarizes overall our research. Section 6 concludes our findings and section 7 describes our proposed future work.

2. Background

This section starts with the brief discussion of the previous related work that has been done in this area and in the later half of this section, we provide user with some background information and concepts required to understand rest of the material. Our discussion in this section is mainly related to three lines of work, i.e. mining and analysis of the software data, analysis of tracing of the software and automated testing.

A lot of work has been done in the analysis of software development data. Some have worked in mining the software development data i.e. mining CVS archives [19, 20].

Ramly and Stroulia’s [6] demonstrate that sequential data mining algorithms can be applied to discover patterns of user activities from system-user interaction traces and they developed a process for identifying special types of patterns called interaction patterns for web-based systems and legacy systems. They used these discovered patterns for interface reengineering and personalization.

Xie and Pei’s [5] work on mining API usages so that developers can understand the API usage and write client code more effectively. For that they developed an API usage mining framework and its supporting tool MAPO. It searches the source code search engines (like Koders [32], gonzui [31]) and mine to extract frequent used APIs for developers to inspect.

Pauw et al [4] gives an execution pattern view that identifies execution patterns in object-oriented visualization so that a programmer can visualize and explore a program’s execution at various levels of abstraction. But this work limits to only visually explore the program execution so it cannot be helpful in finding similar patterns of program execution. There has also been considerable amount of work done by the dynamic analysis community on dynamically analyzing execution traces of object-oriented systems and aiding the development of robust and reliable large-scale systems [21, 22, 33].

A lot of work has also been done in the analysis of tracing of the software like Lencevicius et al [34] proposed a generic approach that gives the design coverage of the executed software so that the test suites are developed easily, they use the design coverage from tracing and trace analysis framework.

An extensive research has also been done in the field of automated testing. There have been numerous publications on automated testing. Like Xie and Notkin’s [23] work on automatically identifying unit tests for object-oriented program without requiring any specification. So the developer can use them to augment with their existing test. There approach was based on statistical algebraic abstraction and program properties. In our approach, we have identified patterns in the call sequences that can later be helpful in generating automated test cases. Xie et al developed a framework, Wrasp that generates automatic test for AspectJ programs [24]; their work on automatically identifying special and common tests from a large number of automatically generated tests without requiring specifications [25].

To the best of our knowledge no work has been done in mining software execution traces to identify patterns that can be later used for automated testing.

Our work is different from the previously done work. We have used Aspect Oriented Programming approach to generate call sequences and then we have applied data mining techniques to identify the patterns of similar call sequences. These patterns can be later used to develop automated test cases.

For a better understanding of the topic, we have explained the concepts of Aspect Oriented Programming (AOP), Data Mining and Clustering in the following sub-sections.

2.1 Aspect Oriented Programming

In Aspect Oriented Programming (AOP), the different aspects of a system are programmed separately and then these separate programs are combined together to produce executable code [1, 2]. AOP makes it easier to slice programs on the basis of separation of concerns thus making it easier to maintain [1, 2].

Following concepts of Aspect Oriented programming might help understanding rest of our discussion.

Cross-cutting concerns: classes need to interact with each other even though each of them is performing different tasks. There are cases where two aspects of a systems behavior seem to get mixed together in the
code. In such cases we say the two aspects cross-cut each other with respect to the program [2].

**Advice:** it is the supplementary code that is added to an existing model that is executed upon each interception of function call or field access. In our case we have logged software execution traces with the help of an “Around Call” advice, which intercepts all calls taking place during execution.

**Point-cut:** this is the well-defined point in execution at the time of interception.

**Aspect:** Advice and Point-cut when combined enable us to add new functionality or aspect to our existing software. Like in our case it provides execution-tracing aspect to our target software.

### 2.2 Data Mining and Clustering

Data mining can also be termed as information mining from databases for knowledge extraction and data pattern analysis. Data mining algorithms are applied on the data to extract patterns that can be recognized by a human. There are mainly three types of data mining algorithms [7]:

- Classification / Regression
- Basket and Sequence Analysis
- Clustering Methods

Clustering is used to identify clusters or groups of objects that have high similarity within a group and they are dissimilar from other groups. The similarity and dissimilarity is based on distance measure. The distance measure can either be a numerical value or a conceptual difference that can be used to distinguish an object from other objects.

Identification of similar patterns was performed using the well-established concept of data mining called clustering. We have applied distance-based clustering methods on our data as it suits to our requirements. Distance-based clustering methods group objects in the cluster on the basis of distance criteria [7].

### 3. Experimental Setup/Methodology

This section describes the experimental setup that is used to identify patterns in software execution traces. We divided our experimental setup into two modules namely,

- Execution tracing
- Pattern discovery

Execution tracing was performed using AOP technique of aspect weaving. For that we selected an open source tool named “AspectDNG” [10]. Execution tracing module provides us with the data that we can later use for pattern discovery.

Pattern discovery was performed using clustering algorithms of data mining paradigm. Open source implementations of clustering algorithms were converted and modified to suit our process. In the following text we talk about these modules in detail.

#### 3.1 Tracing OO Software Execution with AOP

We define *Call Sequence* as an ordered set / sequence of *function calls* that an *object* makes in its lifetime. These calls can be to any function that exists in its *scope*, whether belonging to same *class type* or any other *class type*. Also this call can be to an *instance function*, *static function* or a *constructor* of some *class type*. By definition, we will not consider calls from *static functions of the class* if they are not referenced using any *object*, i.e. referenced using *class type* name. (.NET does not allow calling *static functions* using *object instances* where as Java does).

Unlike the previous work done on execution tracing, we have managed to generate call sequences for this data and later on used it to discover similar patterns using the AOP methodology.

We used aspect oriented programming to weave a tracing aspect in our software that helped us intercept every function call that occurs, and logs it thereby providing us with a detailed execution trace. This trace can provide us very helpful information, i.e. interaction between objects of different classes in the component under examination.

There are many existing tools for aspect weaving in existing object oriented software, most of them in their infancy, for instance *AspectJ* [27], *JBOSS AOP* [28], *LOOM.NET* [29] etc. With a little experiment we decided to use an open source Aspect Weaver called “*AspectDNG*” [10]. This utility can weave any .NET assembly without needing to have its source code. Its author does not yet claim stability of the said tool.

Initially the “*JoinPoint*” [10] parameter we received at the interception module did not contain some of the information we were interested in. Since *AspectDNG* [10] is an open source implementation, we added few more members in “*JoinPoint*” [10] to make it suitable for our needs.

---

1 This paragraph requires knowledge of concepts of *Object Oriented programming model.*
Required information was:

- The “Source” object of the call.
- The “Source” function that was calling.
- The “Source” class type name.
- The “Target” object being called. (The target object was already available in a class member named RealTarget, unlike other items in this list)
- The “Target” function being called.
- The “Target” class type name.

After the code modification, the aspect call handlers now receives the object of modified “JoinPoint” [10] type, which contains the new members and extended information as compared to the original “JoinPoint” [10], we might have deviated from the standard implementation of AOP though. By the end of the execution, the information intercepted was stored in a Data Repository (In this case it is Microsoft Access database), which would be later used to mine for patterns in call sequences.

The data repository is primarily a single relation with a number of attributes. The purpose and description of each attribute is described here.

- **ID**: Primary key, used for uniquely identifying records.
- **SequenceID**: is an incrementing value that is used to sort the function calls and hence generate call sequence in the right order, i.e. the order they took place. This information will not be accurate for multithreaded applications in general, but if we are looking for the calls an object makes and assume that one object will only be contained by one thread (like in our case); it succeeds to provide the correct order of the call sequence.
- **CallerType**: This is the class name of the source from where the function call is made. It is a fully qualified name of the type that includes namespace and class stored as a string.
- **CalleeType**: This is the class name of the target to which the function call is made stored. It is again a fully qualified name of the type that includes namespace and class stored as a string.
- **CallerFunction**: This is the name of the function that is making the function call during its execution. It belongs to the class CallingType.
- **Calleefunction**: This is the name of the function that is being called. It belongs to the class CalledType.
- **CallerObjectHashCode**: Any project can make more than one instances for a given class. To keep the record that which object actually made the call/sub-call we store its hash code, which is unique within the execution scope. This is a very important attribute since it separate call sequences made by two different objects of the same class.
- **CalleeObjectHashCode**: This is the hash code of the target object.
- **DateTime**: This contains the Date and Time when the call was intercepted.

**Hash Code** of the called object can be “NA” (not applicable) in two cases. First, call is made to a static function of a class, hence does not contain an object reference so its hash code is not applicable. Second, Call made to a constructor, in this case the target object is not yet created hence the hash code is not applicable. By our scope we are not interested in calls made from a static function; hence calling object's hash code will always be available for our gathered records.

### 3.2 Mining OO Execution Traces Data

In order to understand how clustering can be performed on call sequences, we first have to define some new terminologies and extend some existing concepts.

#### 3.2.1 Distance Computation

Clustering works on criterion of difference between objects. We will refer to this difference between objects as “distance” throughout the text. Existing distance computation methods are Euclidean distance [14], Manhattan metric [16], Minkowski metric [15] etc. These methods to compute the distance are not applicable to the call sequences(since all these distance measures are graphical and are computed using the coordinate system, call sequences on the contrary have no graphical representation), instead we need some conceptual distance. In order to understand concept of distance between call sequences we first have to define the difference between call sequences.

There have been attempts to compute distance between objects that do not have any graphical representation. Like in the work of Cilibrasi and Vit'anyi [18] who worked on discovering a universal distance measure. This universal measure is not suitable for call sequences; neither are the string distance measures since they can be problematic. Even if the function names are descriptive and the functions that are semantically similar tend to differ less in their names, we will be able to find examples that rules out the string distances. Consider this example.

C1, C2 and C3 are three call sequences, each involving three functions.
C1  Dance > Party > Fun
C2  Dancer > Patty > Gun
C3  Dance > Eat_Drink_Be_Happy > Fun

If we use string distance for grouping, C1 would be more similar to C2 (even though there are different functions involved), because only three characters are different. Where as in C3 there are several different characters. Which ever string distance function we use C1 will be more similar to C2 because of the content (characters) of the strings.

We have to compare each function call of a call sequence with the function call of the other call sequence. We have to understand that a function call can be matched with another function call resulting only Boolean result, i.e. function "Dance" is not similar to function "Dancer".

Function calls cannot be similar, they are either same or different, like characters in a string can either be same or different not similar. But call sequences (sequence of function calls) can be similar (if involve same function calls in similar sequence) just like strings that can be similar.

For our distance measure we should get that C1 is more similar to C3, because they have two functions common and both functions are in same sequence.

For clustering call sequences, we need to define difference criteria. This criterion should return us a numeric value called “Distance Measure”. For lower value of distance measure for two rows we will say they are more similar that the set of rows with high distance measure.

Our distance measure is a scalar quantity, and defined in a way that the commutative property holds.

\[ \text{EditDistance} (C1, C2) = \text{EditDistance} (C2, C1) \]

Distance between two call sequences is the extent to which call sequence differs from another call sequence with respect to function calls that take place, and in the order they take place in the call sequence, i.e. two call sequences are different if they involve different function calls or different order of same function calls. This mean two call sequences are equal only if they involve same set of function calls in the same order, as they exist in either of the call sequence.

Also function names can be same in different classes so we use fully qualified names of the functions to uniquely identify them (name space and class name appended to function's name). In overloaded functions it is safe to assume that they should be considered as a single function since all the functions should do same job principally, with different input and/or output parameters.

To measure the extent by which two call sequences differ from each other, we modified an existing algorithm called “Approximate String Matching” algorithm [9], which computes Levenshtein Distance [13] between strings. We desire to compute Levenshtein distance between call sequences. Levenshtein distance is also commonly called Edit Distance in literature. Hence we will refer to this modified algorithm as “Edit Distance” algorithm throughout the text.

“Approximate string matching” is a dynamic programming algorithm and works on strings and compares the extent to which a string differs from the other, i.e. computes the Edit Distance between two strings. Edit Distance is defined as minimum number of string operations (Insert, Delete, Replace) required, in order to make one string equal to the other. We have to extend the definition of “Edit Distance” and do some modification in existing algorithm to make it work on call sequences.

Edit Distance for call sequences is defined as number of operations (Insert, Delete, Replace) required for making one call sequence equal to other. Primarily the operations listed above are performed on call sequences for insertion, deletion or replacement of a function call in the call sequence.

Strings are logically an ordered sequence of elements; each of the elements is a character. We extend this concept by defining call sequences. Call sequences, like a string, is an ordered sequence of elements; instead of characters, each element represents a function call.

String = “abc”
a, b, c are characters

Call Sequence = “α > β > γ”
a, β, γ are function calls

Instead of representing each function call by a unique symbol we modify the existing algorithm to work on string arrays. Each index of the string array stores textual representation of the function call (i.e. fully qualified name of the called function) and the algorithm instead of comparing characters now compares strings at specified indices. Note that the comparison between function calls is not approximate hence two functions with similar names are treated different when involved in a function call.

Edit-Distance Algorithm created by modifying an existing algorithm named “Approximate String Matching” [9]
P and T are now string arrays (textual representation of call sequences)

**EditDistance** \((P, T)\)

- number of function calls to iterate on
  \(n = |P|\)
  \(m = |T|\)
- initialization
  For \(i = 0\) to \(n\) do \(D[i, 0] = i\)
  For \(i = 0\) to \(m\) do \(D[0, i] = i\)
- iteration over function calls
  For \(i = 1\) to \(n\) do
    For \(j = 1\) to \(m\) do
      \(D[i, j] = \min(D[i - 1, j - 1] + \text{matchcost}(P[i], T[j]), D[i - 1, j] + 1, D[i, j - 1] + 1)\)
  return \(D[n, m]\)

- \(p\) and \(t\) are strings (textual representation of a function call)
- **matchcost** \((p, t)\)
  - if both function calls are same
    If \((p = t)\)
      return 0
    Else
      return 1

The order of the modified distance computation algorithm is \(|P| \times |T|\), where \(P\) and \(T\) are textual representation of two call sequences and \(|P|\) and \(|T|\) are number of function calls involved in \(P\) and \(T\) respectively.

**Examples:**
Consider another set of call sequences as example. In the examples given below see the impact of difference in order of function calls as well as difference of actual functions called.

**Example 1**
Figure 1 exhibits two call sequences that are different in only one function call. This accounts of one replace operation and hence the distance between them is one.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>m2</td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1: Distance between CS1 & CS2 = 1**

Example 2, Figure 2 shows two call sequences that only differ in the order of function calls. But the difference between them accounts for two replacement operations.

![Figure 2: Distance between CS1 & CS3 = 2](image)

3.2.2 Data Transformation

The **Data Repository** that contains the execution traces now undergoes a transformation process to generate call sequences. Up till now we have all the information required, stored in a data repository. We can select a specific class type that we are currently interested in and filter out the records, which are out of our scope, i.e. calls made by objects of other class types.

The function calls made by different objects of the desired class type are now to be stored in a data structure named “Call Sequence”. Every instance of the selected class type corresponds to one call sequence each.

From examples shown in figure 1 and figure 2 we can generate three call sequences, one for each object of the class type source. Here \(o1, o2\) and \(o3\) are objects of the class type we are interested in (Source).

CS1 corresponds to \(o1\).
CS2 corresponds to \(o2\).
CS3 corresponds to \(o3\).

Loading the execution traces in to data structure involves sorting of records with respect to **SequenceID**, thereby preserving the order of the function calls. Also function calls are grouped together with respect to the **CallingObjectHashCode**, which ensures the call sequences generated are corresponding to one object of the selected class type each.

This data structure primarily stores the call sequences in order in a vector and has a capability of presenting textual representation (string array) of the call sequence, which can be the input to our distance computation algorithm. Hence we converted execution traces to a specific data structure, which now can be clustered with respect to the conceptual distance they have with each other.
3.2.3 Discovery of similar patterns

We defined difference criteria among call sequences and also modified an existing algorithm to perform distance computation. Also the execution trace data we collected was loaded to data structures, which can be processed by the distance computation algorithm. We are now ready to go ahead with clustering.

Our target platform for proof of concept is .NET framework. Being unable to find a decent open source collection of clustering algorithm implementations, we decided to convert Weka’s [11] source code from java to C#. Weka [11] is an open source well renowned data-mining library and is programmed in java. Its GNU license gave us liberty to modify its code and convert it to C#.

Only a subset of Weka’s [11] code which was in our scope was converted to C# (i.e. clustering algorithms and core classes). Major work on this conversion was performed by Microsoft® java to C# conversion utility. Conversion issues were resolved with the help of borrowing syntax from two open source utilities vbWeka .NET [12] (a subset of Weka [11] converted from java to VB.NET, only classification algorithms are implemented), and RTools [30] (provides a java-like string tokenizer for .NET).

Currently only two distance-based algorithms are converted for our use, namely

- **Simple K Means**, tries to partition data in K clusters by minimizing the internal error of the cluster iteratively.
- **Farthest First**, works on the principle of partitioning the farthest item first and computing that to which partition other items would go.

With the help of multiple algorithms we can cross validate the results and look for any other interesting information when the results of the two clustering algorithms are compared.

These algorithms worked on **Numeric, Nominal** and **String** type attributes (attribute is a constant value that is used to identify the type of a given data instance) and did not have support for processing call sequences. A new attribute type called “CallSeq” was defined in the core classes of Weka [11] and now the core of Weka [11] can also process our previously defined data structure “call sequences”.

Before clustering the call sequences we added the support of handling call sequences in the clustering algorithms. This modification of code involved removing the “Missing Value Filter” from the clustering algorithms. Filter is a concept in Weka [11] that pre-processes the data that is to be processed by an algorithm. Clustering cannot work on data with missing values so the missing values are replaced with some other values, generally mean value of the records to over come this limitation. Since our call sequence data will never have a missing value, we can eliminate this step in clustering.

Weka [11] implements each clustering algorithm in a separate class. In order to minimize the modification, we inherited these classes into our new classes and we simply override the methods related to distance computation. These methods were

- **Distance computation**: replaced by our algorithm for distance computation.
- **Computation of Cluster’s Mean item**: SimpleKMeans, computes the mean of the cluster in each iteration, which would be later used to find cluster assignments of the call sequences.

**Computation of the Mean item of the cluster** is required in SimpleKMeans algorithm. Mean item is the representative item for the cluster. For numeric attribute this value is simply the arithmetic mean of the attribute. For nominal and string attributes, it is the most frequent item, called Mode.

For call sequence type attribute we find the medoid of the cluster that will be used as the mean item for cluster assignment. For this purpose we compute the sum of distance of an item with all other items in the same cluster. The item, which has the least sum of distances with all the other items of the same cluster it belongs to (the item that has the maximum similarity with other items in the same cluster), is chosen as the mean item for that iteration.

**Pseudo code for finding medoid of a cluster**

```plaintext
► iteration over all clusters
For i = 1 to c do

► iteration over all call sequences in the cluster
For j = 1 to m do

Sum(j) = 0
  ► Compute sum of distance with all items
  For k = 1 to m do
    Sum(j) = Sum(j) + EditDistance
    (Clusters (i).Items(j),
```
Clusters($i$.Items($k$))

$MinIndex = 1$

$\triangleright$ Searching for minimum sum of distances

For $j = 2 \rightarrow m$
do
  If $\sum (MinIndex) > \sum (j)$ then
    $MinIndex = j$

$\triangleright$ Setting medoid of the cluster

$Medoid (i) = Clusters (i).Items (MinIndex)$

Computation of medoid is an expensive operation but it was optimized by the use of distance table, which helps us to avoid computing distance between two specific call sequences over and over (yet it can be considered as drawback of using K-Means for call sequences). This mean item is later used in the iteration for assignment of clusters to the items. Distance is computed for all items against all the mean items of every cluster. The item goes to the cluster for whose mean item has the minimum distance with the item, among all the distances with all other mean items of all the clusters.

**Distance Table and complexity.** Distance computation can be an expensive operation, especially when call sequences involve large number of function calls. More over, distance between same set of call sequences is computed over and over (i.e., in every iteration of clustering). To optimize the usage of results of distance computation, we can store the distance information in a distance table.

This distance table can be populated upon the start of clustering operation or on need to know basis. The distance table is primarily a two-dimensional integer array. Every row and every column corresponds to a call sequence. For $N$ number of call sequences, the size of the table will be $N \times N$.

Distance of a call sequence from itself is defined to be Zero; hence the diagonal values of the table will have a value Zero. We can initialize these values without needing to call the distance computation algorithm. Also we defined that distance between two call sequences is commutative; the distance table will be symmetric. Hence we can compute the upper triangle of the distance table and fill the lower triangle with symmetry and vise versa.

From the examples in figure 1 and figure 2, we can compute the following distance table.

<table>
<thead>
<tr>
<th></th>
<th>CS1</th>
<th>CS2</th>
<th>CS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CS2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>CS3</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

In the table above, we put the diagonal values during table initialization and the values in bold text are put against symmetric nature of the table. Thus the total number of times the distance computation algorithm will be called is,

$$C = \lceil \frac{(N \times N) - N}{2} \rceil$$

### 4. Results

In search of a suitable target for our experiment we discovered that smaller applications does not tend to create numerous objects of same class type thereby limits the number of call sequences generated. We require a rather sufficient number of call sequences so that we can mine for similarity in them and hence later on discover patterns. To provide our experimental setup a suitable input, we decided to write our own application that does create number of class sequences and hence is a good candidate of patterns in call sequences.

We wrote a simple test application in .NET, which creates random number of instances of class “InterestingClass” with in a range of 10 to 100, which would later on determine the number of call sequences we actually get. Afterwards each object of the said class creates a call sequence of random length with a number of different function calls and their permutation. The call sequence length is not fixed, i.e. length of call sequence depends on the outcome of the random number generation; minimum length of the call sequence is 6. The purpose of random call sequences is only to describe our ability to discover patterns; call sequences traced from any application should work with the same principle.

During our examination of the target application, we weaved a call execution tracing aspect in it and logged the call execution. These traces were then transformed to call sequences. A specific trial run of the application generated eighty-six call sequences, which we later on clustered to find patterns.

For representation purpose, each call sequence is represented using a simple notation, “CS” followed by the calling object’s hash code for the corresponding call sequence. Also, each cluster is represented a numeric cluster id, call sequences with same cluster id are grouped together because of their higher similarity.
Call sequences can be very long for larger programs; presenting large sequence diagrams may prove to be poorly readable. We present some call sequences, which are not very long yet fairly large to express our hypothesis. More over call sequences involve fully qualified names of the functions called; we present a smaller representation of a function (function names only) for further readability purpose, even though the actual process uses fully qualified names. Also function names in our target project were intentionally kept small (one lettered function names).

In order to understand our notation of call sequences and the function calls occurring in it, following table may be helpful.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Actual Name / Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Function Call to <code>InterestingClass::E</code></td>
</tr>
<tr>
<td>N</td>
<td>Function Call to <code>Random::N</code></td>
</tr>
<tr>
<td>R</td>
<td>Function Call to <code>InterestingClass::R</code></td>
</tr>
<tr>
<td>A</td>
<td>Function Call to <code>clsAnotherClass::A</code></td>
</tr>
<tr>
<td>B</td>
<td>Function Call to <code>clsCompanyClass::B</code></td>
</tr>
<tr>
<td>C</td>
<td>Function Call to <code>clsOtherClass::C</code></td>
</tr>
<tr>
<td>D</td>
<td>Function Call to <code>clsSomeClass::D</code></td>
</tr>
<tr>
<td>&gt;</td>
<td>Notation for next function call in the call sequence</td>
</tr>
</tbody>
</table>

A typical call sequence is presented in figure 3 in our notation of call sequence along with its UML interaction notation for clear understanding of our representation scheme.

The call sequences were clustered using different available clustering algorithms with different parameters. When clustered using Simple K-Means [Figure 4] we were able to get a distinct cluster assignment for the call sequences, grouping those, which are similar together.

![Figure 3: A call sequence representation](image)

![Figure 4: Summary of call sequence clusters obtained using Simple K-Means](image)

When the same call sequence data was clustered using other clustering algorithms, similar results were found although not exactly the same (this is discussed more in subsequent section). For instance the summary of the call sequence clusters obtained using Farthest First algorithm is shown in figure 5.

![Figure 5: Summary of call sequence clusters obtained using Farthest First](image)
Note that clusters of items are labeled different in both algorithms, which is acceptable because cluster labels are arbitrary and insignificant. The slight difference between the results of one algorithm from the other is because of the limitation of the algorithms, such as their partitioning strategy and the parameter values they depend on.

Figure 6 shows two call sequences from the same cluster; note the remarkable similarity between them (See the boxed portions).

![Figure 6: Two call sequences from the same cluster](image)

Figure 7 shows two call sequences from different clusters, the similarity between them is clearly lesser than the similarity they have with items of their respective clusters.

![Figure 7: Two call sequences from different cluster](image)

Another way of looking at these results can be looking at the differences of the call sequences rather than their similarity. This view is more justified since we performed distance based clustering that works on the difference of the call sequences. We can see that call sequences in figure 6 are less different than each other (see the unboxed portions) since they belong to same cluster. Similarly call sequences in figure 7 are more different since they belong to separate clusters.

5. Discussion

In order to demonstrate our experimental setup, we weaved tracing aspect in the target application. The aspect weaving was done using “AspectDNG” [10], an open source aspect-weaving tool that works upon .NET assemblies. .NET like its competitor java is coming up as a very robust application development framework. With the establishment of Mono project [26] (a project with motivation of running and compiling .NET assemblies on Linux platform), .NET has provided a confronting argument to the java’s platform independence.

The selection of “AspectDNG” [10] was made because of its ease of use and ease of modification. Moreover, unlike its competitors, “AspectDNG” [10] does not require any change in the source of the target application, which (changing the source of the target application may cause the application to lose its integrity). For instance “LOOM.NET” [29] (a powerful competitor of “AspectDNG” [10] requires users to create the objects, that are to be weaved, using a provided application programming interface (API). That means some objects the same class can exist in the application context with a new aspect while others in their unmodified form. Also the objects that are created at runtime without using the API may not be weaved. On the contrary, “AspectDNG” does not even require the source code of the target application. Information about the internal design of the application is a value adding information.

“AspectDNG” [10] in its original form does not fulfill our requirements. We require a lot more information than just the target object. We require information that can vividly describe a function call that is being made (and traced by our newly weaved aspect). A complete description of a function call can be acquired by answering following questions:

- Which object is making the call?
- Which function was calling?
- Which class type is making the call?
- Which object is being called?
- Which function was being called?
- Which class type is being called?

For that purpose we altered the source of “AspectDNG” [10] under the GNU license. Now at a point cut we get all the information to answer above-mentioned questions. This can also help us achieve scalability. Since the “JointPoint” object now contains all the information, it can be stored in an object database like “db4objects” [35] and be processed later during the call sequence construction phase. Currently this information is extracted and stored in a data repository, which can affect the scalability and performance of the target application. Since our focus was on smaller applications, this issue was ignored and will be catered in future when we work on real life applications that are much bigger in size and complexity.

We used version 0.9.53 that was newly released at that time and its stability was not claimed by its author [10]. This version had some limitations (like many other aspect weaving tools in their infancy) and it fails to weave bigger applications and also had some issues with inheritance. Although later release of “AspectDNG” [10] solved these issues but they weren’t so easy to alter to fit our needs. This limits us
to weave only small applications that don’t use inheritance extensively.

We presented results of the experiment in the earlier section, which was performed on an application that creates random call sequences. Random call sequences provide us with the added information that whether our implementation of the call sequences discovery process is biased to any factor or not (i.e. length of call sequences), which could have been difficult to check in execution traces of a real application.

An important issue to discuss is that why did we use call sequences from software that is designed to generate random call sequences. The answer lies in the limitations of the aspect-weaving tool we used (“AspectDNG”) [10] and the modifications we did to it. Since the pattern discovery module does not assume any limitation of the call execution tracing aspect, we can say we can find patterns in data of call execution tracing of a real life application.

Although random call sequences can be used to verify our hypothesis yet it cannot verify one important concept, for a different execution parameters of a software (different order of execution of use-cases), do we still get the same call sequences and similar patterns or not. This would be looked into in more detail once a more stable version of “AspectDNG” is altered to our specific needs thereby enabling us to trace bigger applications.

Execution tracing ended with a large amount of data related to function calls and their order of execution. This data included enough information to vividly describe a function call and its order against other function calls being made in the system. This information was then converted to call sequences. Note that we did not consider overloaded functions as different functions and they are treated as if the same function is being called. Since overloaded functions are supposed to perform similar functionality with different input and/or output parameters it is safe to neglect overloading. Our future work may analyze this topic in detail.

The call sequences hence obtained were passed through the clustering module to discover any existing patterns in the data. The clustering was performed with two available algorithms namely K-Means and Farthest First. These algorithms were used to group call sequences together on the basis of their similarity. The difference measure was defined as number of function calls required to insert, delete or replace in a call sequences to make it a replica of the other call sequences it is being compared to. This difference was computed by the edit-distance algorithm, which was created by modifying an existing algorithm called approximate string matching algorithm.

The results achieved were shown in figures 4 and 5 respectively. From these figures we observe a very close relation between the results of two algorithms. Table 3 provides a similarity insight into comparison of the results of the two algorithms.

Table 3: Similarity between cluster assignments two different algorithms

<table>
<thead>
<tr>
<th>Clusters acquired</th>
<th>Farthest First</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>42</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>46</td>
<td>3</td>
<td>29</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

The slight difference between the results of the two algorithms is due to the limitations of the algorithms and the parameters values they depend on. For instance K-Mean requires the number of clusters to partition data into, in advance. Secondly it requires max number of iterations so that it could stop even if the results do not converge i.e. items keep changing clusters due to their similarity with more than one cluster.

An attempt to overcome the shortcoming of K-Means (that it requires number of clusters beforehand) was made in an algorithm called X-Means [11]. This algorithm repeatedly calls K-Means algorithm on the same data with an increasing value of K. For a value of K, which does not bring a change in the result of clustering assignments, is considered to be appropriate for the given data. For the acquired set of call sequences we performed an experiment which was principally same as what X-Means intend to do, i.e. we executed K-Means on the same call sequence data for incrementing value of K and found the results shown in Table 4.

Table 4: Results for different values of K in K-Means

<table>
<thead>
<tr>
<th>K</th>
<th>Sum of Squared errors</th>
<th>Iterations performed</th>
<th>Clusters acquired</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>430</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>383</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>366</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>366</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

From these results we can clearly see a converging trend when we increase the value of K from 2 to 5. These results also show that for a given value of K, it not mandatory for all clusters to contain call sequences, i.e. call sequences tend to group together on the basis
of similarity. When we move from $K=4$ to $K=5$ we do not see any change in the clustering with respect to clustering assignments and iterations performed. Although this does not mean that for any given value of $K > 4$, K-Means will result in the same cluster assignments, yet this does show that $K=4$ is a good value to represent clusters of the given data.

Similarly Farthest First also requires number of clusters to partition data into. Note that the number of clusters provided in advance may not be real representation of the similarity groups existing in the data but a little experiment with different parameter values can provide us sufficiently optimal result.

Another difference between the results of two algorithms is the cluster labels assigned to each cluster. Cluster labels are insignificant and assigned arbitrarily. If the same algorithm is used on the same data again, different cluster labels are expected. More items on diagonal mean more similarity against mapping of one cluster of one algorithm to another cluster of the second algorithm.

Further more from the results in Table 3 we can see a major group of similar call sequences (K-Means cluster 2, Farthest first Cluster 1). Although we compared the results of two different clustering algorithms for cross validating the results yet such an examination of results also gave us more depth into the results we achieved, such as the items, which do not belong to same similarity group, when results from the two algorithms are mapped on to each other, exhibit such a behavior because of their similarity with both clusters. In our future work we might give fuzzy clustering [17] a try in order to acquire groups with high similarity while catering the similarity of a call sequences with multiple clusters. After acquiring such similar patterns in call sequences, we should focus on such patterns for automated testing to acquire better results.

6. Conclusion

In large software systems, there exists large number of functions and calling orders. It is very hard to touch all the boundaries when testing such type of a software system. We have done an experimental study that helps put more focus on objects that have been used the most.

For our experimental study, we have added some features in an aspect-weaving tool "AspectDNG" [10] that when applied to any .NET assembly helps in generating complete call sequences end to end. We established conceptual distance criteria between call sequences and computed this distance by modifying an existing algorithm i.e. “Approximate String Matching Algorithm” [9]. The modified distance measuring algorithm compares all the call sequences, two at a time, and fills the distance table based on the there similarity. We have developed a tool that applies data mining algorithms to get clusters of similar call sequences.

Summarizing the whole discussion, the paper demonstrated that software execution traces of a .NET assembly were generated, the difference between call sequences was created using Edit Distance algorithm and then by mining these call sequences we get patterns of similar call sequences. Thus, it helps to identify execution patterns with in a large number of call sequences so that more time and effort should be spent on testing such objects. This helps reduce time and effort in testing the software system and developing automated test cases.

7. Future Work

In our current scope we only discovered patterns in call sequences of single threaded applications or at least the object should be contained by a single thread, in future we aim to work on multi threaded application by catering the thread identifiers (process identifiers).

We defined call sequences as the ordered set of function calls an object makes in its life time (call sequences are defined with respect to the source object) and we plan to work on a new definition of call sequence, i.e. set of ordered function calls that are made to a specific object (thinking in term of defining call sequences with respect to the target object).

Also we assumed that overloaded functions perform similar function with a different set of input and/or parameters. In our future scope we aim to incorporate parameters in our call sequences thereby looking call sequences involving overloaded functions more closely. Another direction we can divert concentration of our research is on discovering patterns within parameter values. Clustering parameters will require clustering of objects in their specific states, which requires a lot of work on our side.

For our current research, we developed a tool to discover the patterns in execution traces. We might work on it and release it as an extension to some development environment. The converted code of Weka [11] is helpful to other research scholars who are working on .NET platform and will be provided in accordance to the license of Weka [11].

The results of our experimentation made us ponder on another possibility of pattern identification technique, i.e. fuzzy clustering [17] that might acquire groups with high similarity and cater the similarity of a call sequences with multiple clusters both at the same
We also aim to contribute new concepts in aspect oriented programming that were introduced by us for our research work, i.e. source object, calling and called function names. Future work might require us to add support for actual parameters in aspect oriented programming concept joinpoint, so that we may trace executions in more detail.

Further more we would want some actual automated test case generating tool to work on the results we derived from our research and analyze the benefits gained from it. Our research is merely the first step towards numerous dimensions that can be explored.

Acknowledgments

The author is very grateful to Thomas Gil for his support in fixing issues regarding AspectDNG; to Zeeshan Eejaz for providing help in modifying AspectDNG according to the specific needs; to Robert Feldt and Richard Torkar for guiding at every step of this research.

8. References


