On the Use of Execution Trace Alignment for Driving Perfective Changes

Luciana Lourdes Silva, Klérisson Ribeiro Paixão, Sandra de Amo, Marcelo de Almeida Maia
Computer Science Department
Federal University of Uberlândia
Uberlândia, Brazil
luciana.lourdes@gmail.com, deamo@ufu.br, marcmaia@facom.ufu.br

Abstract — **Perfective changes in well-established software systems are easier to perform when the development team has a solid understanding of the internals. However, it is reasonable to assume that the use of an open source system to incorporate new features and obtain a new software product is an appealing approach instead of coding a new product from scratch. Considering this scenario, and considering that it is not uncommon that systems are poorly documented, there is no widely accepted approach to guide the perfective maintenance for developers with low understanding of the system. This work proposes a new method based on the analysis of execution traces for locating evolution points in the source code where changes should be performed. The proposed method was evaluated with three open source systems and the conclusion suggests a significant impact on effort reduction.**

Keywords – software evolution; execution traces; reuse; software maintenance

I. INTRODUCTION

The available source code for development teams is a rich asset for developing new products, either in open-source environments or in a proprietary industrial context. Nonetheless, it is also well-known that most of this software lacks adequate documentation, or when it exists there is no guarantee that it is either updated, or completed [8].

Most of the effort in changing software systems to add a new feature (perfective maintenance) relies on the comprehension of requirements and code artifacts. One of the most significant problems in current experiences of software evolution is the difficulty of the maintainer to find parts in the source code where new features have to be inserted or where changes have to be done. The traceability among requirements and code artifacts still is a major challenge in perfective maintenance [2].

Several attempts to facilitate understanding software systems have emerged based on the hypothesis that the source code itself is the only reliable source of information about the available software [15]. However, techniques for program comprehension are rather insufficient to developers rapidly grasp the implementation of a software feature. Recent IDEs provide powerful debugging tools for comprehending the system behavior but provide only simplified views to find easily locations where the implementation of new feature should be located in the source code.

This work intends to contribute in the following scenario. *Suppose the existence of a software system with available source code. Suppose the precondition that the typical developers responsible for introducing a new feature in this software have not participated in the development process and have no previous knowledge of its internals. Suppose also that either the documentation is inexistent or developers do not want to use the existing documentation because it is not reliable. For instance, suppose that developers want to evolve a UML modeling tool with more than one hundred thousand lines of code to enable the creation of a new kind of diagram. This scenario imposes a major challenge to the team. One would not expect to carry out this task with one or two man-day effort.*

Currently, the traditional approaches to perform this evolution task include the understanding of the system by means of documentation analysis and/or debugging similar features as an alternative for initial understanding of the system. One hypothesis of this work is that the insertion of a new feature can be performed with low effort with the analysis of similar artifacts existent in the system. The hypothesis supposes that it is possible to discover where the new artifacts will be inserted or changed and also the initial form of these artifacts.

The approach proposed to discover this information is based on the sequence alignment of summarized execution traces. This technique enables the separation of common parts of source code from specific parts related to important features that will drive the addition of the new one. The main contribution of this work is the evaluation of this approach to verify if it helps to locate potential elements of code that can guide the development of a new feature. The evaluation was conducted with real-world systems and with meaningful evolution tasks.

This paper is organized as follows. Section II presents an overview of the approach. This section describes how the execution traces are captured, represented, summarized and aligned. It also shows how the result of the alignment should be used. Section III presents the evaluation of the approach in the three different systems in order to determine how the approach improves the development practice when performing perfective changes in a software system. Section IV presents the discussion about the results. Section V presents the related works. Finally, the last section presents the conclusion and future work.

II. THE PROPOSED APPROACH

The goal of the approach is to retrieve important information from the source code that could guide the comprehension of systems in order to reduce the effort in evolution tasks. For this purpose, an alignment approach of

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.28

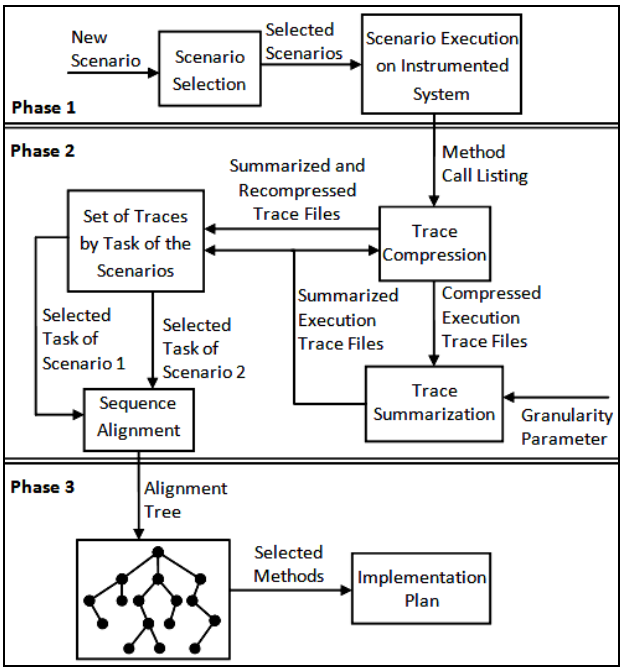

Figure 1. The proposed approach.

execution traces is executed in target systems to find common and variant points in the source code. Thus, the approach is applicable to similar features in a system.

Figure 1 provides an overview of the proposed approach, which has three main phases:

A. *Definition and Execution of Scenarios*
B. *Summarization and Alignment of Execution Traces*
C. *Definition of the Implementation Plan*

A. *Definition and Execution of Scenarios*

In the first phase, the definition of execution scenarios is based on searching features in the system that have some relationship to the new feature, but that should have some specificity among them. The information obtained from this specificity should guide the implementation plan.

1) *Choosing Suitable Pairwise Features*

In order to reuse, modify, or insert new features in a system, firstly the developer needs to know which basic features the system executes. For instance, suppose that the system is an email client. The developer needs to know that *send message*, *receive message*, *save draft*, and *login with a user id* are basic features of the system. The developer can define which features are related to each other. Consider the following example of a simple *figure editor*. Suppose that the features *Draw Rectangle* and *Draw Circle* are already implemented in the system and the developer needs to implement the feature *Draw Triangle*. Figure 2 shows an example of a pair of execution scenarios *A* and *B* in order to detect common and specific code which implements the features *Draw Rectangle* and *Draw Circle*. Some common method calls, those used to create the item menus, are expected to be related to the task *Initialize the System*. Also, during the execution of *Draw a Rectangle* and *Draw a Circle*, there may be some methods calls that are common to both tasks. Specific method calls to draw each kind of figure are expected to be related to the respective tasks in scenarios *A* and *B*. Figure

Scenario A	Scenario B
1. Initialize the System	1. Initialize the System
2. Draw a Rectangle	2. Draw a Circle
3. Copy Rectangle	3. Copy Circle

Figure 2. Example of scenarios to detect common and specific code that implement the features

2 also shows tasks, *Copy Rectangle* and *Copy Circle*, that may help to understand how to introduce *Copy Triangle* in the system.

Nonetheless, it is possible to try to compare the trace data that contains the execution of a feature F1 of interest with the trace data that contains the execution of a totally different feature F2 used just for control. A important decision in the definition of the execution scenarios is that the developer introduces common tasks in the same order in both scenarios, possibly surrounding the execution of F1 and F2 to control where those traces have methods related to F1 and F2, respectively, that can be captured by the alignment algorithm that will be described in the next subsection.

2) *Instrumentation of the System*

To capture execution traces, the target system should be woven with an internal instrumentation tool based on AspectJ that collects the executed methods. The developer needs to mark the beginning of a new task, and when the execution of this task ends, the developer also marks such event. This process repeats until all the features in the scenario have been executed. During the execution of each scenario, a trace file is created for each started thread. Each line of the trace file corresponds to a method call, which has the qualified method name and the corresponding level of the call in the execution stack. The collected information enables to reconstruct the method call tree for each feature executed in the defined scenario.

B. *Summarization and Alignment of Execution Traces*

In the second phase, the traces are compressed, summarized and aligned. An execution trace is a large file because it contains loops and recursive method calls. An alternative to reduce this large size is the elimination of these repetitions [4]. Nonetheless, even after the compression of the trace, preliminary results have shown that the alignment result of the execution traces can result in a quantity of information extremely large that may make the approach unfeasible [9, 12].

1) *Trace Summarization Algorithm*

The goal of this algorithm is to summarize the execution traces that are typically very large, even after the compression process using the algorithm [4]. To make feasible the application of the alignment approach [9, 12], we propose the combination of trace compression with a summarization approach based on the elimination of method calls with low granularity. The granularity of a method call is the total number of other calls that have occurred during the execution of that call. In other words, considering a method call tree, the granularity of a call node *n* is *N-1*, where *N* is the size of the sub-tree with root *n*. So, method calls with greater granularity are expected to be more relevant. The user has to select the minimum granularity of method calls in the summarization process.

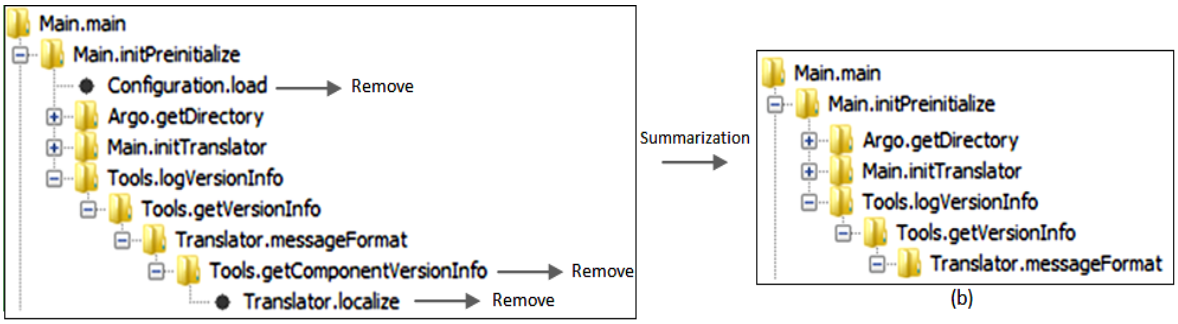

Figure 3. (a) A compressed trace, where the loops and recursion were eliminated and (b) the respective summarized trace.

The lower is the chosen granularity, the lower is the number of removed methods from the trace, producing possibly more useless information to analyze. If the selected minimum granularity is 1, then only method calls with no nested method calls (leaf nodes) will be removed from the trace. On the other hand, low granularity discards less methods calls and the chance of discarding relevant methods is lower. In order to find the most suitable granularity, the user can start the summarization process with granularity 1 and analyze the corresponding result. If there is still too much information to analyze, the granularity can be increased, and so on. If a greater granularity eliminates relevant method calls identified with a lower granularity then the previous granularity is preferred. The Figure 3.a shows as example a fragment of a compressed real trace. The gray arrows show the method calls that will be pruned during the summarization algorithm if the minimum granularity is 2. Figure 3.b shows the summarized trace.

2) Alignment of Summarized Execution Traces

The algorithm proposed in [11] was adapted for our special purposes. The first adaptation is about the characteristic of an execution trace. The traces can contain noises. For instance, in the sequences "XaaaaY" and "XaaaY", one can suppose that these sequences were produced by the same source code. The difference could be caused by a loop that executed the method "a" four times in the first sequence and three in the other. That is the reason for compressing the traces. After the compression process, then the summarization algorithm is performed on compressed traces, as shown Figure 3.

In the trace alignment process, the traces are already compressed and summarized, that is, loops, recursions and method calls with low granularity were removed from the traces. The input to the alignment tool is two traces of possibly different sizes. At the end of alignment execution, two new traces with the same size are produced, containing blanks inserted such that aligned methods are exactly at the same position in both traces.

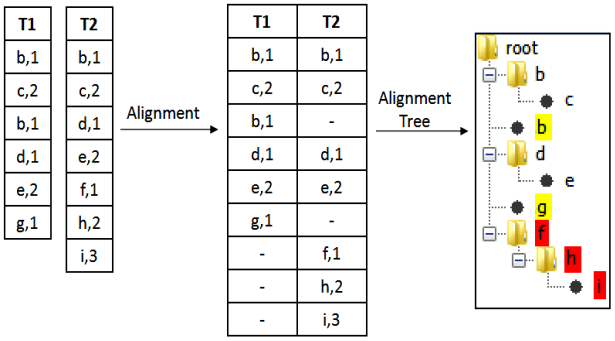

Figure 4. The trace alignment process.

A pair of aligned traces can be analyzed in segments. Trace segments are groups of consecutively aligned elements and also groups of consecutively misaligned elements. The aligned result of a pair of traces is better represented in a tree view, because it shows the nesting of method calls. The misaligned segments are shown in different colors, where nodes with the same color are from the same sequence. Figure 4 illustrates an example of the trace alignment. Two traces are presented containing several method calls *(b, c, d, e, f, g, h, i)* and their respective level in the execution stack. The sequence <(b,1), (c,2)> indicates that the method *c* was called from the method *b*. In the alignment tree, the first segment presents the calls *b* and *c*, where the call *c* is a child of the call *b*. The second segment contains a misalignment indicating that the call to method *b* in trace T1 does not have a counterpart in trace T2. The next alignment segment presents the calls *d* and *e*, where the method call *e* is a child of the call *d*. After that, the misaligned segment, containing only the method call *g* in trace T1, does not have the counterpart in trace T2. Finally, the last misaligned segment indicates that the method call sequence *f*, *h,* and *i* in trace T2 does not have the counterpart in trace T1 at same position.

One hypothesis of this work is that misaligned segments in the sequences indicate method calls that are specific to the different features executed in each trace. Therefore, misaligned segments suggest methods that should be analyzed to complete the implementation of the new feature. Nonetheless, the misalignments also can represent noise in the execution trace such as mouse events.

It also may be useful to analyze aligned method calls that were executed just before or after the misaligned calls because those methods have possibly driven the variation.

C. Definition of the Implementation Plan

The third phase consists of a visual analysis of the alignment trees obtained from the previous phase. Some threads and some misaligned segments in which method names are not closely semantically related with the execution scenario can be discarded. During the scenario execution, the developer marked the beginning of each scenario step. This information allowed producing the

summarized trace files separated for each scenario task. If the scenario execution starts more than one thread then it is possible to know which threads executed a task. Thus, in order to understand how a task is implemented, the developer selects only the threads that contain method calls executed for the current task being analyzed.

The design of the implementation plan starts by selecting methods that would help to understand how a feature is implemented in the source code. Relevant methods to execute a task are selected from the alignment trees in order to guide in the implementation of a new feature. The methods are elected as important by a simple inspection of package, class and method names in the alignment tree.

An important question in the evaluation of this approach is the quality of the list of methods for the plan. In our study, we defined the control to assess the quality of the list of methods, the Actual Positive (AP) methods, during the actual implementation of the new feature.

The methods included in the set AP are those that were necessary to understand the required change and those that were effectively changed. Using the methods in the set AP and in the implementation plan it is possible to know the following values:

TP: True Positives. Methods pointed out by our approach that were effectively used.

FP: False Positives. Methods pointed out by our approach that were not used.

FN: False Negatives. Methods that were necessary but were not pointed out by our approach.

These values are used to evaluate the approach in terms of precision and recall.

III. EVALUATION

The goal of the evaluation is to determine how the approach improves the developer practice when performing perfective changes in the software system. As a matter of fact, a reasonable measure that would assess the hypothetical improvement is the necessary effort to perform the task of understanding the target system and implementing the required changes. This option would require an experimental study with a reasonable number of developers performing the task using a traditional approach and using our approach. As a first study we have decided to understand how the proposed approach would help the developer in potentially providing precise and sufficient information that would guide the execution of change tasks. The choice should analyze the precision and recall of the information provided by the proposed approach. Three studies with different systems have applied the execution trace alignment approach to drive the evolution of them.

The selected systems and perfective changes are described below:

1. ArgoUML 0.30 is a UML modeling tool (161 KLOC) with 92 packages, 2122 classes, 156 interfaces, 2314 attributes, 1914 static attributes, 13918 methods, and 669 static methods. In the above numbers, libraries such as, the framework GEF - *Graph Editing Framework* and the OMG metamodel are not included. The chosen perfective change for this system is the addition of a new kind of diagram: *Object Diagram*.
2. Llama Chat is a chat server/client pair for use on the web (2260 KLOC). The chosen perfective change for this system is the addition of *Speech Synthesis,* possibly, for accessibility purposes.
3. Columba is an email client written in Java (101 KLOC) with 336 packages, 1475 classes, 225 interfaces, 2963 attributes, 957 static attributes, 7924 methods, and 453 static methods. The chosen perfective change for this system is the addition of a new way of sending messages: *Send Scheduled Message.*

The selected systems are open source software available in a public repository in reasonably different application domains. The authors did not contribute in the past in the maintenance of these systems. The perfective changes were selected to be meaningful ones and not just cosmetic changes, such as, changing a property of the GUI.

The studies include several phases to define a set of suggested methods that the developers should consider in order to get the perfective change done. The assessment of this set of methods requires effectively implementing the changes and recording the methods that were actually used in the implementation, and then the result set of methods can be evaluated in terms of precision and recall.

A. ArgoUML- Experimental Results

The following 3 phases were performed in order to get a suggested set of methods to plan the implementation of the changes.

1) Definition and Execution of Scenarios

The scenarios were specified searching for existing diagrams that were similar to the required new diagram. The class diagram and the component diagram are similar because they are both based on nodes and links between them. Nonetheless, the kind of nodes and links may be specific for each one. The selected scenarios, shown in Table I, have a direct relation between their steps.

2) Summarization and Alignment of the Traces

During the execution of each scenario, six trace files were collected, one for each started thread. The compressor was executed to remove loops and recursive calls. Then, the trace summarizer reduced the size of compressed traces, eliminating method calls with low granularity. Methods with granularity 2 were removed. An attempt to remove methods with granularity greater than or equal to 3 was discarded because the resulted traces had a major influence in the results of the alignment, pruning specific methods that were clearly related to the

TABLE I. PAIR OF SCENARIOS FOR ARGOUML

A. Draw Class Diagram	***B. Draw Component Diagram***
1. Initialize the system	1. Initialize the system
2. Create Class Diagram	2. Create Component Diagram
3. Add class C1	3. Add component D1
4. Add class C2	4. Add component D2
5. Insert an association between C1 and C2	5. Insert a dependency between D1 and D2

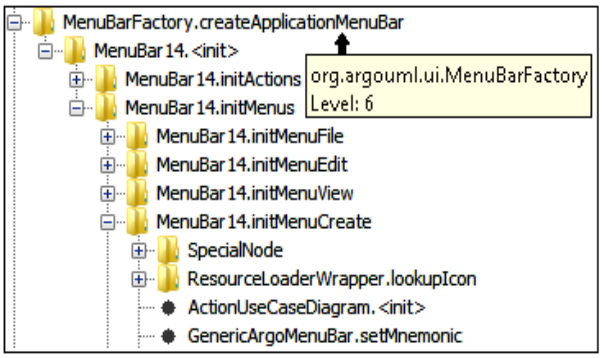

Figure 5. Fragment of the tree that contains the initialization.

executed task. An adequate selection of the granularity parameter will be discussed in the next section. Finally, the compressor was applied again on summarized traces to remove possible loops that were not detected on the first compression. Table II shows the result of each step to reduce the traces of the selected scenarios. The trace reduction ratio is related to the original size. During the scenario execution, the developer must inform the beginning of each scenario step. This information allows producing summarized trace files separated for each scenario step. For instance, 2 trace files have been created for the step "Create Class Diagram", one that was obtained from the trace of Thread 2 and another from the trace of Thread 6. These 2 trace files contain only method calls executed to create the class diagram.

After summarization, the alignment algorithm was executed in the summarized traces obtained from the previous phase. Each alignment was obtained between i^{th} thread of the scenario A and i^{th} thread of the scenario B. The result of each alignment was represented in one separated alignment tree used in the next phase.

3) *Definition of the Implementation Plan*

The first step in this phase is a visual analysis of the alignment trees in order to define the selected methods that would be relevant to the implementation of the feature *Object Diagram*. The methods were selected by a simple inspection of package, class and method names.

Before starting method selection, some threads and some misaligned segments whose methods are not closely semantically related to the execution scenario are elected to be discarded. The main semantics of the threads could be analyzed by visual inspection of the alignment tree. For instance, the result alignment of the Thread 2 between the scenario A and B has shown 73 misalignment segments. After analyzing these segments, 4 misaligned segments were discarded. For instance, `mouseExited` is a mouse event that is not closely semantically related to any important action of the execution scenario, so its corresponding segment was discarded. Indeed, there is some room for subjectivity in this criterion and it is possible that the developer could have an aggressive

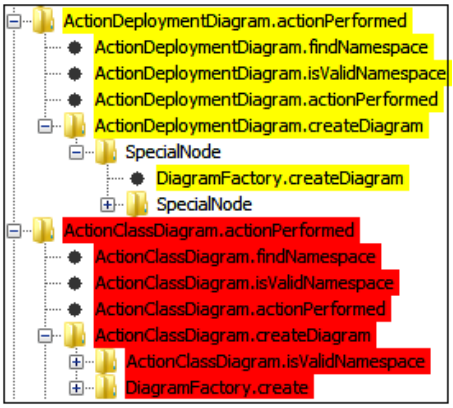

Figure 6. Fragment of the tree that illustrate the specific call sequence for each scenario.

discard criterion, eliminating important methods. Instead, our approach was quite conservative because only 5% of misaligned segments were discarded. Thread 1 and Thread 2 contained method call names related directly with the initialization of the system as shown in Figure 5. Thread 2 also contained method call names related to the creation of the class diagram, the addition of classes in the diagram and the addition of an association. Figure 6 shows a fragment of the tree, illustrating a misalignment occurred during the creation of class and component diagram. The attentive reader may notice that the methods called when drawing a component diagram are called from the class `ActionDeploymentDiagram`. The reason is that the deployment diagram palette was used to create the component diagram. Thread 3 contained method calls of classes which draw class figures (`FigClass`) and component figures (`FigMNode`). The semantics of these classes was inferred by their names, but confirmed only with source code inspection. The conclusion was that Thread 3 was responsible for rendering graphics. Since, Thread 2 also contained `FigClass` and `FigMNode`, then Thread 3 was discarded because it had no other relevant task. After this point, we concluded that a class named `FigObject` was necessary. A search for classes with the name `Fig*` has revealed that this class already existed and should be considered for reuse, if possible. Thread 4 contained method calls related to critics. For the sake of simplicity, we did not include critics in the implementation of the feature *Object Diagram*, so this thread could also be discarded. Thread 5 has been discarded because all method calls have been aligned, and they were also related to critics. Finally, the methods of Thread 6 were related to OCL and critics, and so, the thread was discarded for the same reason.

The implementation plan of the new diagram has been designed by just analyzing the method calls in Thread 1 and Thread 2. The implementation plan has been guided by the several features that need to be added in the

TABLE II. COMPRESSION AND SUMARIZATION RESULTS OF THE ARGOUML'S EXECUTION TRACES

Scenarios	*Task*	*Trace Size (Method Calls)*	*Compressed Trace Size/ Compressing Ratio / Elapsed Time (ms)*	*Summarized-Compressed Trace Size / Summarization Ratio / Elapsed Time (ms)*	*Compressed-Summarized-Compressed Trace Size / Comp-Summ Ratio / Elapsed Time (ms)*
A	5	238.107	181.608 / 23,73% / 26607	43.146 / 81,88% / 4531	36.733/ 84,57% / 3305
B	5	277.720	216.277 / 22,12% / 43087	52.701 / 81,02% / 5375	41.305 / 85,13% / 4157

software. A feature is an element observable from the user point of view. The designed features are 1) menu item to *Create Object Diagram,* 2*)* buttons to *Add Objects in Diagrams* and c) events to *Add an Association between 2 Objects*. The implementation plan consists in the following 3 steps describing how each feature has to be implemented.

Step 1: Create Object Diagram. The first point is to find where menu items are created and then introduce a new menu item for creating object diagrams. The hypothesis is that the aligned segments in thread 1 show common elements of the initialization. Therefore, some of these elements, for instance, the menubar, should be adapted to support the new diagram. The search for the relevant method calls started with the expansion of the first node of the tree, which has two children. The method call `initializeSubsystems` of the class `Main` was the one selected. This node was expanded – to level 2 – and the method `initializeGUI` of the class `Main` was selected among 11 nodes because the goal is to understand the GUI initialization. The process of navigation was the same until level six, where we found the method `createApplicationMenuBar` of the class `MenuBarFactory` – the first node in Figure 5. In this method call, the constructor of the class `MenuBar14` – level 7 – was selected after a confirmation of its semantics by inspection of the source code. Changes that should to be done to insert the menu "*New Object Diagram*" were supposed to be in method `MenuBar14.initMenuCreate`, also shown in Figure 5. This method has been inserted in the plan for creating the item menu. As shown in the tree, there is an instantiation of the class `ActionUseCaseDiagram.<init>` for the use case diagram, so this class and method have also been added to the plan because we assumed that a class/method `ActionObjectDiagram.<init>` should also be implemented.

The necessary methods to handle the event "*Create Object Diagram*" have been found in the alignment between the scenario A and B (Thread 2). The hypothesis is that the misaligned segments during drawing each diagram contain specific fragments of the class and component diagrams and, these fragments should be also added for the new diagram. The Figure 6 shows a piece of the tree, distinguishing the specific calls of each scenario. The methods of each scenario were presented in the tree with different colors, being easier to locate them. For instance, the class `ActionDeploymentDiagram` contains the methods `actionPerformed`, `findNamespace`, `isValidNamespace`, `createDiagram` and the class `ActionClassDiagram` contains the same method names. So, we inserted these methods in the plan for creating the object diagram. The method `create` presented in the `ActionClassDiagram's` segment has been inserted as well.

Step 2: Add Objects in Diagrams. The selection of methods that would help to understand how to add objects in diagrams was based on the hypothesis that misaligned segments that contain method calls related semantically with "*3. Add Class C1*" and "*3. Add Component D1*", should provide relevant information to *Add Object*. The search for these methods has been achieved in the same way as in step 1. The methods that were evaluated to be semantically related to the addition of the object in a diagram were `addNode`, `addElementListener`, `createDiagramElement`, `getFigNodeFor`, `canAddNode`, `createDiagram`, and `doesAccept`.

Step 3: Add Association between 2 Objects. The hypothesis for this step was the same as for the previous step. The misaligned segments containing method calls related with "*4. Insert Composition between C1 and C2*" and "*4. Insert Dependency between D1 and D2*" were analyzed because the misalignment should provide relevant information to *Add Association between Object O1 and O2.* The methods selected in the plan were `addEdge`, `canAddEdge`, `getFigEdgeFor`, `createAssociation`, `connect`, `createAssociationEnd`, `createDependency`, `buildConnection`, `paint`, `paintClarifiers`, and `buildAssociation`, `buildDependency`.

TABLE III. NUMBER OF ELEMENTS INSERTED INTO ARGOUML

Kind of Element	*Quantity.*
Packages	2
Classes	6
Methods	13
Properties	3
Total Inserted LOC	606

4) ArgoUML – Analysis of Results

After the design of the implementation plan, a set of 31 methods were listed by our approach as being important in conducting of the perfective change.

The control to assess the quality of this set is the Actual Positive (AP) methods that were discovered during the implementation of the new diagram. A set of 32 methods were really necessary to add the feature. The methods included in this set are those that were necessary to understand the required change and those that were effectively changed. The object diagram was implemented in ArgoUML with approximately 606 lines of code, including insertions made by the Eclipse IDE such as imports, stub constructors, added methods automatically after inserting the clause implements, and so on. Table III shows the number of inserted elements in the source code.

Table IV shows an analysis of the result in terms of TP (True Positives: methods pointed out by our approach that were effectively used), FP (False Positives: methods pointed out by our approach that were not used) and FN (False Negatives: methods that were necessary but were not pointed out by the approach).

B. Llama Chat's Experimental Results

This study evaluates the approach with a case where the perfective change also includes the reuse of one system to enhance a feature of the other. In this case, the

TABLE IV. PRECISON AND RECALL IN ARGOUML

System	*TP*	*FP*	*FN*	*Precision%*	*Recall %*
ArgoUML	19	6	12	76	61.29

speech synthesizer system FreeTTS[1] has been reused to enhance the Llama chat with speech synthesis capability. The fundamental API of FreeTTS was understood using the documentation. The following 3 phases were performed in order to get a suggested set of methods that would help to understand where the feature of FreeTTS should be introduced in Llama Chat.

1) Definition of the Execution Scenarios

This case is different from the ArgoUML approach because we could not find two similar features that would be useful to implement a third feature. In this case, the scenarios can be specified with the search for two features, such that one or both would be related to the methods that potentially should be updated to support the new feature. In this case study, the goal was the composition of two systems. In Llama chat, a user A sends a private text message to a user B and this user B receives and listens the speech of the text message. Thus, the relevant point of change is the part of code that receives a message, which should be modified to introduce speech synthesis. The selected scenarios, shown in Table V, contain the features Receive Message and Send Message.

TABLE V. PAIR OF SCENARIOS FOR LLAMA CHAT

C. Send a Private Message	***D. Receive a Message***
1. Initialize the system	1. Initialize the system
2. Change Channel	2. Change Channel
3. Send Message to User B	3. Receive Message From User A

2) Summarization and Alignment of Traces

During the execution of each scenario, four trace files were captured, one for each started thread. After, the process of compression and summarization was started. This process was the same as the previous study. Table VI shows the result of each step to reduce the traces of the selected scenarios.

The alignment method has been performed on summarized traces obtained from the last step. Alignment trees were obtained in the same way as in the previous study.

3) Definition of the Implementation Plan

Firstly, a visual analysis of alignment trees helped to select methods that would be relevant to the composition of the independent systems. The methods were selected by a simple inspection of package, class and method names. The first study of alignment trees showed which threads contained method calls related to *send private messag*e and *receive messag*e. Threads 2 and 3 could be discarded because Thread 2 contained only method calls to initialize the system and Thread 3, methods to queue the messages. The implementation plan was specified

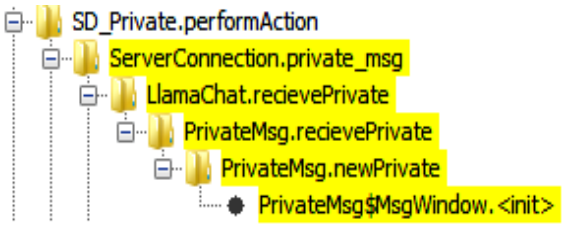

Figure 7. Fragment of the tree that illustrates the specific call sequence to receive a private message in Llama Chat.

with the analysis of alignment trees of Thread 1 and Thread 4. The implementation plan consisted in the following 2 steps.

Step 1: Send Message. The selection of methods that would help to understand how a message is sent was based on the single misaligned segment of Thread 1. The selected methods, semantically related to the *"3. Send Message to User B"*, were `sendPrivate` and `newPrivate`. They were inserted in the plan for adding the new feature.

Step 2: Receive Message. The hypothesis for this step was the same as in the previous step. The alignment of the thread 4 contained three misaligned segments. The selected methods, semantically related to "*3. Receive Message From User A*", were `privateMessage`, `recievePrivate` shown in Figure 7. They were inserted in the plan.

4) Llama Chat – Analysis of Results

The implementation plan was based on a set of 4 methods. The Actual Positive (AP) methods were discovered during the implementation of communication of the two systems. A set of 5 methods were really necessary to insert the new feature, including all the four methods of the implementation plan. The methods included are those that were necessary to understand the required change. Only one method was effectively changed. Twelve lines of code were necessary in order to incorporate the speech recognition feature of FreeTTS into Llama Chat. Table VII shows an analysis of the result in terms of TP , FP, and FN.

TABLE VII. PRECISON AND RECALL IN LLAMA CHAT

System	***TP***	***FP***	***FN***	***Precision%***	***Recall %***
Llama Chat	4	0	1	100	80

C. Columba's Experimental Results

This study evaluates the approach with a case that required the approach to be applied twice: to understand where the maintenance should take place, and how to reuse other parts of the system, instead of replicating the feature. The 3 phases of the approach were applied as follows in order to retrieve a suggested set of methods for

TABLE VI. COMPRESSION AND SUMARIZATION RESULTS OF THE LLAMA CHAT'S EXECUTION TRACES

Scenarios	***Task***	***Trace Size (Method Calls)***	***Compressed Trace Size/ Compressing Ratio / Elapsed Time (ms)***	***Summarized-Compressed Trace Size / Summarization Ratio / Elapsed Time (ms)***	***Compressed-Summarized-Compressed Trace Size / Comp-Summ Ratio / Elapsed Time (ms)***
C	3	262	145 / 30,29% / 106	27 / 89,69% / 41	20/ 92,37% / 30
D	3	116	93 / 19,83% / 60	22 / 81,03% / 41	21 / 81,89% / 21

[1] http://freetts.sourceforge.net/docs/index.php

TABLE VIII. PAIR OF SCENARIOS FOR UPDATING COLUMBA

E. Send Message Now	*F. Send Message Later*
1. Initialize the system	1. Initialize the system
2. Write New Message	2. Write New Message
3. Send Now	3. Send Later

guiding the implementation of the new feature *Send Scheduled Message*.

1) *Definition of the Execution Scenarios*

Two scenarios were specified in order to search for existing features that were similar to the new feature. One can notice a close correspondence between the steps of the selected scenarios presented in Table VIII.

Another pair of scenarios has been selected in order to reuse some implemented tasks in the system, shown in Table IX. The scenario "New Appointment" has been selected to guide how to reuse some methods that insert a new appointment in the Calendar.

TABLE IX. PAIR OF SCENARIOS FOR COLUMBA FOR REUSING

E. Send Message Now	*G. New Appointment*
1. Initialize the system	1. Initialize the system
2. Write New Message	2. Create Calendar
3. Send Now	3. Insert New Appointment

2) *Summarization and Alignment of Traces*

During the execution of each scenario, seventeen trace files have been captured, one for each started thread. The process of compression and summarization were the same as the previous studies. Table X shows the result of each step to reduce the traces of the four scenarios selected.

The alignment method on summarized threads was executed in the same way as in the previous studies to produce the alignment trees.

3) *Definition of the Implementation Plan*

After a visual analysis of the alignment trees, methods were selected by examining carefully their package, class and method names. Threads that contained method calls related to "Send Now", "Send Later", and "Insert New Appointment" were identified and the others were discarded. Several misaligned segments and threads could be discarded because they had no method names semantically related to any of their features. For instance, the methods `getBorderInsets`, `mouseEntered` were not considered semantically related to *Send Now* or *Send Later* features. Indeed, subjectivity is a matter and this decision could reduce the recall metric. We adopted a conservative approach, such that, in the case of doubt we preferred not to discard.

The implementation plan of the new feature has been specified analyzing the method calls in the threads 3, 4, 5, 8, 16, and 18. The implementation plan is guided by the required features to be added and reused in the software. The implementation plan consists of the following 3 steps describing how each feature has to be added.

TABLE XI. SET OF METHODS OF THE IMPLEMENTATION PLAN FOR THE TASK SEND MESSAGE IN COLUMBA

Class Name	*Method Name*
SendAction	actionPerformed
SendLaterAction	actionPerformed
SendMessageCommand	Process
SendMessageCommand	Execute
SaveMessageCommand	Process
SaveMessageCommand	Execute
CachedMboxFolder	addMessage

Step 1: Add Button in the Tool Bar. The selection of methods that would help to understand how a button is added in the toolbar was based on the misaligned segments of Thread 3. This thread contained the method calls executed to write a new message, and so, it was a candidate to inform where the new button had to be added. The buttons and panels in Columba are built using XML files. Then, we should use some specific methods recovered from the alignment tree to toggle breakpoints in a debug session in order to identify which file was read during the creation of the new message window. The methods considered to be semantically related to the creation of the frame and toolbar were `createCustomViewItem`, `initComponents`, `createButton`, and `actionPerformed` belonging to the class `NewMessageAction`. These methods were inserted in the plan for adding the name of the new button in the XML file.

Step 2: Send Message. The selection of methods that would help to understand how a message is sent was based on the hypothesis that misaligned segments containing method calls semantically related to "*3. Send Now*" and "*3. Send Later*" should provide relevant information. The methods that were evaluated to be semantically related to sending a message were considered in the implementation plan as shown in Table XI.

Step 3: New Appointment. The selection of methods that would help to comprehend how an appointment is created was based on the misaligned segments that contain methods semantically related to "*Insert New Appointment*". The selected methods, semantically related to the feature, were inserted in the implementation plan. These methods were `createButtonPanel`, `createPanel`, `actionPerformed` of the class `NewAppointmentAction`, and `CalendarPicker.init`.

4) *Columba – Analysis of Results*

A set of 14 methods and 1 class were selected and inserted in the implementation plan by our approach as being important for conducting the perfective change.

The Actual Positive (AP) methods were discovered during the implementation of the new feature in the

TABLE X. COMPRESSION AND SUMARIZATION RESULTS OF THE COLUMBA'S EXECUTION TRACES

Scenarios	*Task*	*Trace Size (Method Calls)*	*Compressed Trace Size/ Compressing Ratio / Elapsed Time (ms)*	*Summarized-Compressed Trace Size / Summarization Ratio / Elapsed Time (ms)*	*Compressed-Summarized-Compressed Trace Size / Comp-Summ Ratio / Elapsed Time (ms)*
E	3	132.872	29.521 / 77,78% / 5376	7.025 / 94,71% / 904	5.788/ 95,64% / 571
F	3	89.871	25.107 / 72,06% / 3464	6.012 / 93,31% / 778	5.218 / 94,19% / 645
G	3	116.825	30.887 / 73,56% / 3831	6.785 / 94,19% / 834	5.788/ 95,22% / 358

TABLE XII. NUMBER OF ELEMENTS INSERTED INTO COLUMBA

Kind of Element	*Quantity.*
Classes	2
Methods	7
Insertion in Properties and XML files	3
Total Inserted LOC	256

system. A set of 11 methods were really necessary during the process of adding the new feature. The scheduled message has been implemented in Columba with approximately 256 lines of code. Table XII shows the number of inserted elements in the source code. Table XIII shows an analysis of the result in terms of TP, FP, and FN.

TABLE XIII. PRECISON AND RECALL IN COLUMBA

System	*TP*	*FP*	*FN*	*Precision%*	*Recall %*
Columba	11	3	0	78.6	100

IV. DISCUSSION

One fundamental step of the approach proved to be the summarization process. The summarization process is based on filtering methods with low granularity, i.e., methods with primitive operations or methods that call other few methods with low granularity. The greater the granularity value, the greater the number of pruned methods of the trace. Excessive pruning is undesirable because the alignment would not provide enough information for helping in the definition of the maintenance plan, so a more conservative approach would be preferable at the price of having more false positives. If the result of alignment provides excessive information, then the developer can use greater granularity values to reduce the size of traces and consequently, the number of alignment segments. Previous experiments have shown that the alignment approach without the summarization of execution traces results in a large number of misaligned segments [9, 12]. This situation would introduce a large number of false positives in the approach, especially if we consider a conservative approach to discard unrelated method names.

An important question that we should try to answer is "Can our approach be considered actually effective in reducing the effort necessary to perform perfective maintenance?". Suppose that a developer needs to insert a diagram in ArgoUML by debugging the source code. How many thousands of lines should be debugged? Of course, there is no definitive answer to this question because this would depend on the ability of the programmer in "*stepping over"* uninteresting method calls until they reach the important methods that are necessary to understand and implement the changes. However, the tables that depict the size of the execution traces show that even for summarized traces the number of method calls is still too large and would impose a significant challenge on the ability of the programmer in finding the relevant information. On the other hand, if we consider the results of our approach, we have reached high levels of precision in the three studies presented, 77%, 100% and 79%, respectively. These results mean that almost all methods indicated to be important in the perfective maintenance were actually important. Besides that, the recall of the studies, 63%, 80% and 100%, respectively were also significant, because even if not all information is provided, the missing important methods would be easier to find after the developer has gained better understanding of the system using the true positives methods suggested by the approach.

V. RELATED WORK

To best of our knowledge, there is no application of sequence alignment algorithms to analyze execution traces, except those developed by the authors [9]. They have shown and introduced the use of execution trace alignment. However, in that study they did not assessed the approach with a detailed application in software evolution.

Other approaches to analyze execution traces have been extensively used in the comprehension of software [1, 17]. A seminal work is the Software Reconnaissance approach that also compares code executed in traces with and without the features [16] but without an alignment algorithm. Some authors [10, 14] suggest the integrated use of static and dynamic views of the software system. The dynamic views are obtained by means of profiling of the most used features in the system. Nonetheless, the primary goal is to obtain the system architecture to reduce the effort of comprehension of the system. In our approach, we have not focused on the most used features for comprehending the system in an overall manner. We have focused on well chosen features in order to provide direct information customized for the process of software evolution. This requires less effort to make needed changes in the source code.

In [5], another approach to summarize traces was designed using fan-in and fan-out metrics, which are different from the granularity metric proposed in this work.

The IDE's debugging and search tools help to understand how that objects interact with each other. Nonetheless, usually it is not possible to extract a precise view of interdependency among several classes involved in the implementation of a feature, as our approach has demonstrated to extract. The approach of software evolution presented in this work guides the developer in a precise manner. Consequently, this reduced drastically the effort of implementation.

A work that in certain way is similar to ours is based on the premise that the insertion of new similar features in a system is seen as the practice of include clones [7]. Other studies show that from 7% to 23% of source code in typical systems are cloned [13]. Partly, these studies corroborate to our hypothesis of that evolving source code using similar fragments is useful.

VI. CONCLUSIONS

In this paper, we have proposed an approach based on the analysis of summarized execution traces in order to reduce the effort of software evolution tasks. Our work has shown how to retrieve important information from the source code to help to conduct perfective maintenance.

One of our working hypotheses that we worked was that the misalignment of execution traces are points of specific implementation of a particular feature. Based on this hypothesis, we have evaluated our approach in the evolution of three different open source software systems. Firstly, we implemented in the UML modeling tool ArgoUML the object diagram. Secondly, we implemented the speech recognition in the chat server/client pair for use on the web Llama Chat. Thirdly, we implemented the schedule message in the email client Columba. The experimental results have shown a considerable quality of the list of methods produced by our approach as being important to conducting the perfective changes. Our results suggested that the example scenario described in Section I, in which a developer unfamiliar with the target system, should introduce a new kind of diagram in a UML editing tool with a one or two man-day effort is actually feasible, as one could hardly believe.

ACKNOWLEDGMENT

We acknowledge the Brazilian agencies FAPEMIG, CNPq and CAPES for partially funding this research, and the members and previous members of the Software Engineering Lab at the Computer Science Department of Federal University of Uberlândia for the collaboration on Reverse Engineering projects.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. van Deursen, A Controlled Experiment for Program Comprehension through Trace Visualization, *IEEE Trans. on Software Engineering*, vol. 99, no. PrePrints, 2010

[2] A. de Lucia, R. Oliveto, G. Tortora, Assessing IR-based traceability recovery tools through controlled experiments, *Empirical Software Engineering*, 14 (1), Springer, Netherlands, 2009, pp. 57-92.

[3] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[4] A. Hamou-lhadj and T.C. Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In *Proc. of IWPC*, 2002, pages 159-168.

[5] A. Hamou-Lhadj, T. Lethbridge. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System, In *Proc. of IEEE ICPC*, pp.181-190, 2006.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. An Overview of Aspect J. In *Proc. of ECOOP*, 2001.

[7] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. *In IEEE Intl. Symp. on Empirical Software Engineering*, pp. 83–92, 2004.

[8] T. C. Lethbridge, J. Singer, and A. Forward, How Software Engineers use Documentation: The State of the Practice, *IEEE Software,* vol. 20, no. 6. CA, USA: IEEE Computer Society Press, 2003, pp. 35–39.

[9] M. Maia, V. Sobreira, K. Paixão, S. de Amo, I. Silva. Using a Sequence Alignment Algorithm to Identify Specific and Common Code from Execution Traces. *4th Intl. Workshop on Program Comprehension through Dynamic Analysis*. Antwerp, pp. 6-11. 2008.

[10] M. Mit and M. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proc. of ICSE*, pp 24-27, 2003.

[11] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology,* (48):443–453, 1970.

[12] Paixão, K.R. *Using execution trace alignment to understand variation points in source code (in portuguese)*. Master´s Dissertation. Federal University of Uberlândia. 2009. 78pp.

[13] C.K. Roy and J.R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems, in: *Proc. of the 15th Working Conference on Reverse Engineering, WCRE 2008*, pp. 81-90, 2008.

[14] K. Sartipi, and N. Dezhkam. An Almalgamated Dynamic and Static Architecture Reconstruction Framework to Control Component Interactions. In *Proc. of WCRE*, pp 259-268, 2007.

[15] V. Sobreira, and M. Maia. A Visual Trace Analysis Tool for Understanding Feature Scattering. In *Proc. of WCRE*, pp.337-338, 2008.

[16] Wilde, N., and Scully, M.. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*. 7 (1), pp. 49–62, 1995.

[17] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. *In Proc. of ICSE*, pp. 470-479, 2004.