# Using a Sequence Alignment Algorithm to Identify Specific and Common Code from Execution Traces

Marcelo de A. Maia, Victor Sobreira, Klérisson R. Paixão, Sandra A. de Amo, Ilmério R. Silva

Computer Science Department
Federal University of Uberlândia
Uberlândia, MG, Brazil

{marcmaia,deamo,ilmerio}@facom.ufu.br, {victor.sobreira,klerissonpaixao}@gmail.com

## Abstract

*Software product lines are an important strategy to improve software reuse. However, the migration of a single product to a product line is a challenging task, even when considering only the reengineering task of the source code, not mentioning other management challenges. The reengineering challenges are partially due to effort of identifying common code of similar features, even when we know those features in advance. This work proposes the alignment of execution traces in order to discover similar code of similar features, facilitating the reengineering task. We present the architecture of our approach and preliminary results that shows a promising direction.*

## 1 Introduction

Changes are inherent to software systems [4]. Every successful software goes through continuous evolution either to support new user expectations, hardware changes or operational changes. However, providing software evolution easily, quickly and correctly is still a major challenge for software engineers because applications are increasingly complex. This complexity is consequence of more sophisticated non-functional requirements. Most maintenance tasks are originated from new user requests, that is, perfective maintenance tasks [3, 4]. One of the major problems in software maintenance is related to program comprehension. The effort of comprehending of what will be modified is estimated in 40% to 60% of the whole effort of the maintenance phase[1]. This situation is aggravated when software documentation is either not updated, unintelligible, or simply does not exist. Another complicating issue is the software size. Reverse engineering techniques are being developed with relative success, but their scalability to large systems is still a challenge.

This work proposes a reverse engineering technique using a sequence alignment algorithm. Sequence alignment algorithms have been applied in Molecular Biology to compare two or more sequences of DNA, RNA or protein in order to find out if there exists some similarity between them. For example, if we have two sequences: `ATGGATGCCC` and `ATGCATCCC`, a possible alignment would result in the following two sequences, respectively: `ATG-GATGCCC` and `ATGC-AT-CCC`. Note that gaps are introduced in the original sequences so that an *i-th* element of the first sequence can match the *i-th* element of the second sequence. The idea of this work is based on aligning similar execution traces in order to find out where the two traces match (common code) and where they mismatch (specific code). The technique is aided by a semi-automated tool to help the identification of specific and common code of similar features. The traces should be captured from similar execution scenarios of the system, otherwise it is not expected to find common code. It is important that the developer knows what are the commonalities and variabilities between two execution scenarios from an observational point of view in order to establish adequate traces for alignment.

## 2 The Approach

In Figure 1, a general view of our approach is presented using an UML activity diagram.

The first activity is to define suitable scenarios that enables extracting relevant information when comparing two executions traces. This is a manual activity, and information used as input for this activity comprehends new user requests that defines what kind of maintenance will be performed, similar features present in the system and the avail-

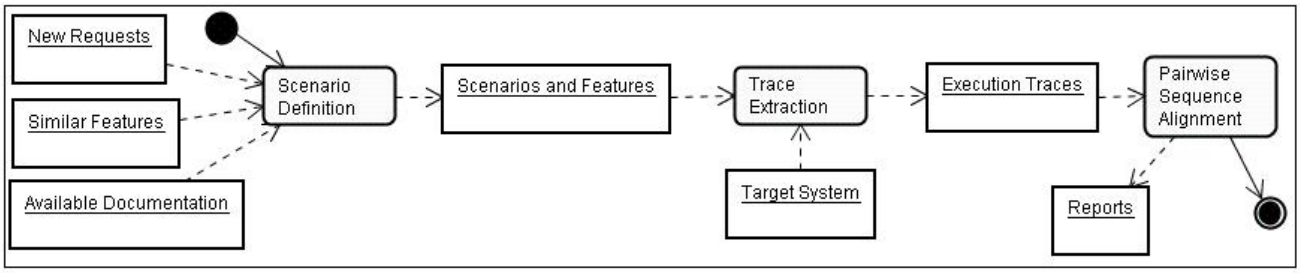


**Figure 1. The proposed approach**

able documentation of the system. The result of this activity is the definition of an execution scenario that must be performed, and consequently, the input data that enables the desired execution of the target system.

The trace extraction activity is automated. Our trace extractor is implemented in AspectJ. Currently, our target system must be implemented in Java. During the execution of the target system, the extractor intercepts method calls and writes a text file for each thread launched during the execution. Each line of the file corresponds to a method call whose content is the fully qualified name of the called method.

After the traces are collected, the next step is to perform an automatic pairwise alignment with two selected traces. The expected result of the alignment is the information of what is common to both sequences and what is specific to each sequence.

This information can be used to focus on the source code that is important to desired maintenance task.

## 3 Related Work

Some ideas of this work were inspired in other works in software maintenance.

Ding and Medvidovic proposes an incremental process for the evolution of object-oriented systems with poor or inexistent documentation. [6]. One phase of the process is the architecture recovery of specific fragments of the system. There are three assumptions for this phase: definition of the desired changes, knowledge of the application properties from the user point of view and the understanding of basic architectural features of the implementation plataform. This work was posteriorly revised with the addition of new heuristics for the phase of identifying components and with new case studies [9]. Our work relates to this, in the sense that the result of alignments helps to focus system understanding on the desired points of system evolution.

Rajlich and Silva have studied the reuse and the evolution of *orthogonal architectures*, which are code fragments organized in layers within the same abstraction level [11]. They have developed an application domain independent process aiming at adapting the system architecture to encompass a new requirement set. The authors have concluded that such process have application in small and medium-sized systems, and that the source code modularization was not effective for large scale systems. We expect that sequence alignment can be applied to large scale system in order to help focusing on the most important places to eventually modularize.

Sartipi et al. developed a work that comprehended in recovering system architecture using patterns defined in the AQL language - Architectural Query Language - and together with data mining techniques. The system is translated from source code to a graph model that is suitable for pattern-matching [13]. In other work[12], a framework that combines static and dynamic information is proposed. We also believe that we will need to combine the dynamic information extracted from sequence alignment and combine it with static information in order to achieve a more robust result.

Vasconcelos et al. [15, 16] presents a set of heuristics for class clustering in object-oriented systems from execution traces, using a similar idea of combining dynamic and static information.

Impact analysis techniques are responsible to identify the parts of the system that will be affected by a change. A well-known technique is program slicing. Binkley e Gallagher have presented a survey about this technique[5]. Our work can be used to provide the slicing criteria for understanding the impact of a software change.

Clustering is a data mining technique used for classifying related source code entities using similarity metrics. Wiggerts [17], Anquetil [2] and Tzerpos [14] shows different aspects on the clustering algorithms for source code. Feature location is a common task in software evolution activities. Marcus et al. has presented the application of an information retrieval method - *Latent Semantic Indexing (LSI)* that is used to map concepts written in natural language to relevant fragments of source code [8]. Our work also aims at locating source code fragments that are relevant in a soft-

ware evolution or software restructuring context based on external point of view of behavior.

In the Bioinformatics field, comparing sequences has became a major activity. The identification of similar regions in DNA, RNA or protein sequences can help mapping sequences to functional, structural and evolutionary characteristics. Several algorithms are presented in [7]. However, it is still a challenge to define and adapt sequence alignment algorithms for the software maintenance field. Surprisingly, at the best of our knowledge, we still could not find the use of alignment algorithms to detect similarities and commonalities of execution traces.

# 4 Sequence Alignment

Sequence alignment is a well-studied problem. Needleman and Wunsch have already proposed an algorithm for analyzing protein sequence in early seventies [10]. Several algorithms have been proposed since then. Indeed there are some issues that must be considered when adapting these algorithms for maintenance purposes.

## 4.1 Characteristics of Execution Traces

Our execution traces normally can present some patterns that can provide us with some information. For example, consider the sequences "`XaaaaY`" and "`XaaaY`". We can suspect that these sequence may be generated from the same code, and they are different just because the method "`a`" was called inside a loop that executed four times in one trace and three times in the other. Another example, consider the sequences "`XaaaaY`" and "`XaabaY`". In this case, we can suspect that some condition enabled the execution of the method "`b`", possibly inside a conditional command.

## 4.2 Global Alignment vs Local Alignment

Global alignments attempts to align every element in the sequences. These strategy is most useful when the sequences are similar and of roughly equal size. A general global alignment technique is the Needleman-Wunsch algorithm that is based on dynamic programming. Local alignments are more useful when we are trying to find a smaller sequence inside a larger one. The Smith-Waterman algorithm is a general local alignment algorithm and is also based on dynamic programming. There are also hybrid methods that attempt to find the best possible alignment that includes the start and end of one and the other sequence.

In this paper, we have chosen to study the alignment of almost similar sequences. Our goal was to choose similar features and to find out what is common and what is different between them. In such a situation, a global alignment strategy seems a reasonable alternative.

## 4.3 Pairwise Alignment vs Multiple Alignment

Pairwise alignment is used to find local or global alignments of two sequences. If it is necessary to compare several sequences, the alignment can only occurs with two sequences at a time, and the user should proceed with an integration step with another technique. Multiple sequence alignment is a generalization of pairwise alignment, in the sense that the alignment algorithm can take as input several sequences at a time. However, general multiple alignment algorithms tend to lead to NP-complete solutions, and thus are not very practical, unless you provide some heuristics or use a very small input.

In this paper, we have chosen to study the pairwise alignment because execution traces are normally large.

## 4.4 Identity and Similarity

In Bioinformatics, identity and similarity are related but different concepts. The identity is a relation of equality in which a nucleotide or aminoacid of one side must be equal to its complement to produce a match. This relation is too restrictive in Biology, so the alignment algorithm may consider to match two different elements, if these elements have some level of similarity.

In principle, considering that classes may have a reasonable cohesion, we could consider methods in the same class or in the same package to have some level of similarity, and thus apply the same principles of biology. However, in this work we have chosen to consider only the identity relationship as a prerequisite for matching two method calls.

## 4.5 Gap Penalty

Because in Bioinformatics is reasonable to accept the alignment match between two different elements, a question may arise when deciding if a match based on similarity is better or not than a gap that is inserted in one of the sequences.

In this work, we have decided not to penalize the introduction of gaps in either of the two sequences for two reasons. The first is that since we work only with identity, it seems incoeherent to accept an alignment match with two different elements instead of introducing the gap. The second reason is that the misalignment gives us also an important information: it may represent specific method calls of a sequence and thus contribute to identify specific code.

# 5 Application and Current Results

In this section, we present an application of sequence alignment to report specific and common code between two

| Traces | Length - Prefix | Length - Specific to Rectangle | Length - Suffix |
|---|---|---|---|
| Rectangle | 885 matches and gaps | 396 specific to Rectangle | 18 matches and gaps |
| Circle | 885 matches and gaps | 396 gaps only | 18 matches and gaps |

**Figure 2. Length and Characteristic of Aligned Sequences**

features in a small graphical editor shown in Figure 3. The total lines of code of the editor is 387, the number of classes is 9, and the total number of methods is 58.

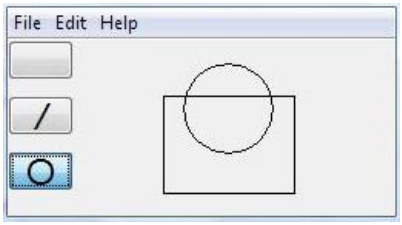


**Figure 3. The target system**

We have chosen two similar features to execute the system: drawing a rectangle and drawing a circle. The execution traces were collected and two threads were launched for each execution. Each pair of corresponding threads were aligned with a Needleman-Wunsch algorithm, considering only identities and zero gap penalty. Below we present the results of the alignments.

## 5.1 Results

The first thread was responsible for drawing the main frame and there was only 14 method calls perfectly aligned between each other.

The second thread was more interesting. Although the system is small and the execution scenarios are fairly simple, the thread for drawing a rectangle had 1040 method calls and the thread for drawing a circle had 644 method calls, and thus manual alignment seems unfairly hard. After the gap insertions each sequence had the gaps inserted and grown to 1299 elements.

In Figure 2, we show the tree main subparts of the traces and their correspondence. The interesting alignment is in the first and third part, summing 903 elements. After analyzing manually the traces, we could find out that the 396 elements in the second part, corresponds to gaps in the thread of drawing circle, because the size of the drawn rectangle was greater than the size of the circle and thus demanded more screen updates. In Figure 4, we summarize the quantitative details of the alignment.

After the alignment, we computed the set of common methods between the two features, the set of methods specific to the feature of drawing a rectangle and the set of

| Before Alignment | |
|---|---|
| Length of Rectangle Trace | 1040 |
| Length of Circle Trace | 644 |
| Difference Rect-Circle | 396 |
| **After Alignment** | |
| Length of Rectangle Trace | 1299 |
| Length of Circle Trace | 1299 |
| #Matches | 385 |
| #Gaps in Rectangle Trace | 259 |
| #Gaps in Circle Trace | 655 |
| #Real Gaps in Circle Trace | 259 |
| Length of Interesting Alignment | 903 |
| %Matches | 0.4263 |
| %Interesting Gaps in Rectangle | 0.2868 |
| %Interesting Gaps in Circle | 0.2868 |

**Figure 4. Quantitative results**

methods specific to the feature of drawing a circle. The results are shown below, respectively. False positives have arised when finding methods specific to draw a rectangle. The reason was that it was not possible to align those 396 method calls with a counterpart in the draw circle feature, as already shown in Figure 2. Nonetheless, the other results seem promising because no false negative has arised and all called methods were present in at least one of the above three sets.

```
// Common methods
graphicaleditor.MainFrame$1.paint
graphicaleditor.MainFrame.access$0
graphicaleditor.ShapeSet.draw
graphicaleditor.MainFrame.access$1
graphicaleditor.MainFrame.processWindowEvent
graphicaleditor.MainFrame$4.mousePressed
graphicaleditor.MainFrame.drawPanel_mousePressed
graphicaleditor.MainFrame.createShape
graphicaleditor.Point2D.<init>
graphicaleditor.MainFrame$5.mouseDragged
graphicaleditor.MainFrame.drawPanel_mouseDragged
graphicaleditor.Point2D.getX
graphicaleditor.Point2D.getY
graphicaleditor.MainFrame$4.mouseReleased
graphicaleditor.ShapeSet.add
graphicaleditor.MainFrame.drawPanel_mouseReleased
graphicaleditor.MainFrame.jMenuFileExit_actionPerformed

// Methods specific to draw rectangle
graphicaleditor.MainFrame$1.paint
graphicaleditor.MainFrame.access$0
graphicaleditor.ShapeSet.draw
graphicaleditor.MainFrame.access$1
graphicaleditor.Rectangle.<init>
graphicaleditor.Rectangle.setAnchor
graphicaleditor.MainFrame$5.mouseDragged
graphicaleditor.MainFrame.drawPanel_mouseDragged
graphicaleditor.Rectangle.getAnchorX
graphicaleditor.Point2D.getX
graphicaleditor.Point2D.getY
graphicaleditor.Rectangle.getAnchorY
graphicaleditor.Rectangle.draw
graphicaleditor.Rectangle.setDimension
graphicaleditor.Rectangle.getAnchor
```

```
// Methods specific to draw circle
graphicaleditor.Circle.<init>
graphicaleditor.Circle.getAnchorX
graphicaleditor.Circle.setAnchor
graphicaleditor.Circle.setDimension
graphicaleditor.Circle.getAnchorY
graphicaleditor.Circle.getAnchor
graphicaleditor.Circle.draw
```

## 6 Final Remarks

In this work, we have shown an approach to identify commonalities and variabilities in execution traces. The possibilities of usage of these information are manifold. We can help the introduction new features in the target software based on similar characteristics already present providing information of specific methods that the feature must implement. We can help extracting common components from source code based on information provied by commonalities between execution traces.

There are many questions that still persist, for instance, how the approach will scale up for larger systems, how the extracted information can be more systematically used by developers, how would be the results when working with different versions of the system, how much the trace compression would enhance the approach, and how different alignment methods behave in different situations.

**Acknowledgments**. We would like to thank CNPq and CAPES for partially funding this research.

## References

[1] A. Abran, P. Bourque, R. Dupuis, and L. Tripp. Guide to the software engineering body of knowledge (ironman version). TR, IEEE Computer Society, 2004.

[2] N. Anquetil, C. Fourrier, and T. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, Washington, DC, 1999.

[3] K .Bennett. Software evolution: past, present and future. *Information and Software Technology*, Vol. 38(11):673–680, November 1996.

[4] K .Bennett and V .Rajlich. Software maintenance and evolution: a roadmap. In *Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM Press.

[5] D. Binkley and K .Gallagher. Program slicing. *Advences in Computer*, 1(43), July 1996.

[6] L .Ding and N. Medvidovic. Focus: a light-weight, incremental approach to software architecture recovery and evolution. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 191–200, 28-31 Aug. 2001.

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

[8] A .Marcus, A .Sergeyev, V .Rajlich, and J .Maletic. An information retrieval approach to concept location in source code. In *WCRE '04: Proc. of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[9] N .Medvidovic and V .Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[10] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3), 1970.

[11] V. Rajlich and J. Silva. Evolution and reuse of orthogonal architecture. *IEEE Transaction on Software Engineering*, 22(2):153–157, 1996.

[12] K. Sartipi, N. Dezhkam, and H. Safyallah. An orchestrated multi-view software architecture reconstruction environment. In *Proc. of 13th Work. Conf. on Reverse Engineering*, pages 61–70, Oct. 2006.

[13] K. Sartipi, K. Kontogiannis, and F. Mavaddat. Architectural design recovery using data mining techniques. In *Proc. of 4th European Conf. on Soft. Maintenance and Reengineering*, pages 129–139, March 2000.

[14] V .Tzerpos and R. Holt. Software botryology: Automatic clustering of software systems. In *International Workshop on Large-Scale Software Composition*, pages 811–818, 1998.

[15] A. Vasconcelos, R. Cêpeda, and C. Werner. An approach to program comprehension through reverse engineering of complementary software views. In *1st Intl. Workshop on Prog. Comprehension through Dynamic Analysis (PCODA)*, pages 58–62, 2005.

[16] A. Vasconcelos and C. Werner. Software architecture recovery based on dynamic analysis. In *Simpósio Brasileiro de Engenharia de Software*, 2004.

[17] T. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proc. of the 4th Working Conf. on Reverse Engineering (WCRE '97)*, page 33, Washington, DC, 1997. IEEE Computer Society.