An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension

Heidar Pirzadeh, Akanksha Agarwal, Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montréal, Québec, Canada
{s_pirzad, a_agarw, abdelw}@ece.concordia.ca

***Abstract* – Understanding the behavioural aspects of a software system is an important activity in many software engineering activities including program comprehension and reverse engineering. The behaviour of software is typically represented in the form of execution traces. Traces, however, tend to be considerably large which makes analyzing their content a complex task. There is a need for trace simplification techniques that can help software engineers make sense of the content of a trace despite the trace being massive. In this paper, we present a novel algorithm that aims to simplify the analysis of a large trace by detecting the execution phases that compose it. An example of a phase could be an initialization phase, a specific computation, etc. Our algorithm processes a trace generated from running the program under study and divides it into phases that can be later used by software engineers to understand where and why a particular computation appears. We also show the effectiveness of our approach through a case study.**

Index Terms – Program comprehension, dynamic analysis, trace analysis, trace phase detection.

I. INTRODUCTION

Software maintenance is perhaps one of the most difficult tasks in software engineering. Existing systems usually suffer from poor to non-existent documentation which complicates the task of software maintainers. They often need ways to understand the system before they can make any modification that preserves the system reliability. Research in the area of program comprehension and reverse engineering aims to reduce the impact of this problem by investigating techniques that can help software engineers understand large systems which has undergone several years of ad hoc maintenance [1].

There exist two types of system analysis techniques: static and dynamic analysis. Static analysis relies on the source code to recover high level views of the system that can be used by a software engineer to understand what the system does and why it does it this way. The second approach, which is the focus of this paper, is based on the analysis of the behavioural aspects of a system by first executing it and then analyzing the generated run-time information. Run-time information is typically represented in the form of execution traces. There are several techniques for generating execution traces including instrumenting the system or the execution environment. Traces, however, have been historically difficult to work with since they can be overwhelmingly large. Recently, there has been an increase in the number of trace analysis tools that can help engineers make sense of large traces. These tools rely on some sort of abstraction techniques that vary significantly from one study to another but with a common objective being to extract high-level views from raw traces (e.g. [2, 3, 4]). These techniques suffer from many drawbacks including the fact that they rely heavily on users to distinguish important trace content from noise. This is usually a complex task, especially when applied to large traces.

In this paper, we present a novel technique for simplifying the analysis of execution traces by devising an algorithm that can divide a trace content into various fragments that correspond to the execution phases of a program. We define an execution phase as part of a program that implements a specific task such as initializing variables, performing a particular computation, etc. It is expected that a typical run of a program will include several phases that correspond to the task being performed. An execution phase has also been defined in other areas such as in program optimization as any stable period in which the execution of a program uses the same amount of resources [10, 11, 12].

In this paper, we focus on traces of routine (method) calls, which are commonly used for the purpose of program comprehension [5, 6]. We use the terms method, routine, and function to mean the same thing. Our phase detection algorithm is based on the fact that a phase shift within a trace appears when a certain set of methods responsible for implementing a particular task and which are prevalent in one phase starts to "fade" as the program enters a new phase, where new methods start taking place. The algorithm operates on the trace while it is generated (i.e. on the fly), which is usually preferable to offline processing since it

eliminates the need to save the entire trace even if only part of it is needed.

We have not come across any study in the area of trace analysis for program comprehension that deals with execution phase detection. Perhaps, the closest studies to ours are the ones proposed by Cornelissen et al. [7] and Reiss et al. [13, 14] where the user can visually discern major phases in the execution scenario in execution mural views that help outline the system's general functionality. These techniques can be further explored to detect phases and where phase transitions appear.

The remaining part of this paper is organized as follows: The phase detection algorithm is presented in Section 2. In Section 3, we show the effectiveness of our approach when applied to a case study. We conclude the paper in Section 4.

II. PHASE FINDING

The idea behind our phase detection approach is to detect when and where during the execution of a program, the methods that implement a particular phase start to disappear as new methods begin to emerge, indicating the beginning of another phase. Our proposed phase detection algorithm operates on a trace while it is generated (i.e., on the fly). This is contrasted with an offline approach, where the entire trace is first collected before applying the algorithm. Online processing of traces is usually more desirable than an offline approach since the users can see the results early and may need to make decisions based on this early feedback without having to wait until the entire trace is generated.

Our algorithm is composed of two key steps which are:

1. Phase Change Detection: In this step, we detect if there an execution phase shift. In other words, we detect when a large number of methods of the first phase disappear and that new methods are invoked.

2. Phase Shift Location: Once a phase change is detected, we need to know the exact location in the trace where the phase shift has taken place, i.e., at which point in the trace the methods of the previous phase start to disappear and that new ones start to appear.

A. Phase Change Detection

In order to detect the phase changes of a program, it is required to capture a set of distinct methods that have been invoked as the program is executing. We refer to this set as a *working set (WS)*. We sort in an ascending manner the methods of a working set based on their prevalence. This is used to detect the phase change (Step 1) by detecting when the most frequent methods become less frequent as the program enters a new phase. It is also used to indicate when the phase change has started to take effect (Step 2). As the program executes, its working set is subsequently updated so that it can reflect the changes in program's behavior. Updating the working set after each new method invocation can be computationally expensive. Instead, the update can be done after a certain number of method calls (i.e., a *chunk* of methods calls). This way, the chunk size can specify the update rate of the working set. The chunk size is provided as input. Determining an appropriate chunk size that balances the computation overhead with the accuracy of the approach is a topic that we intend to tackle in the future. As shown in Figure 1 (lines 1 - 15), at the beginning of our phase finder algorithm, we make a new working set and keep a snapshot of the working set every time we update it until a phase change is detected.

```
1  phaseFinder(Chunk_i: chunk of methods, T:threshold)
2  {
3   if (i == 1)
4     WS = new workingset()
5   for each m in Chunk_i
6   { // Building working sets (WS)
7      if WS.contains(m)==False
8      {
9          WS.add(m)
10     }
11     WS.rank_methods()  // ranking method within a WS
12  }
13  Snapshot_i = WS
14  if (i == 1)
15     Snapshot_o = Snapshot_i
16  Distance = compare (Snapshot_o, Snapshot_i)
17  if (Distance < T)
18  {
19    for each candidate m
20    { // Detection of the shift location with voting
21      for chunk correspond to (Snapshot_o . . Snapshot_i)
22         if m.rank(chunk) is close to mid-rank
23            chunk.vote()
24       return (chunk with maximum votes)
25     }
26    Snapshot_o = Snapshot_i
27   }
28 }
```

Fig. 1. The pseudo code of phase finding algorithm

Figure 2 shows an example of a routine call trace. Assuming that the chunk size is set to 3, the working set WS will contain the first three methods *A*, *B*, and *C*, sorted in a descending order based on their ranking.

The rank of a method directly related to its prevalence. That is, the rank of each method is the number of methods with a better prevalence plus one. The prevalence function takes into account the frequency of a method in part of the

trace that has been processed so far (*frequency(m)*), the chunk number in which the method was first introduced (*first_chunk(m)*), the current chunk number (*curr_chunk*), and the chunk size (*chunk_size*) as shown in the following equation:

$$P(m) = \frac{frequency(m)}{(current\_chunk - first\_chunk(m) + 1) * chunk\_size}$$

The rationale behind this function is to keep track of the prevalence of the methods as the algorithm goes through the trace chunks. If a set of methods keep appearing relatively at the similar rate after each chunk is processed, then this is a good indicator that the program is still in the same phase. On the contrary, if some of the methods start "fading" (i.e. appearing less frequently) according to a certain threshold, then this suggests that a new phase is taking place. We anticipate that the threshold is application specific and that a tool that supports our approach should allow enough flexibility to vary the threshold.

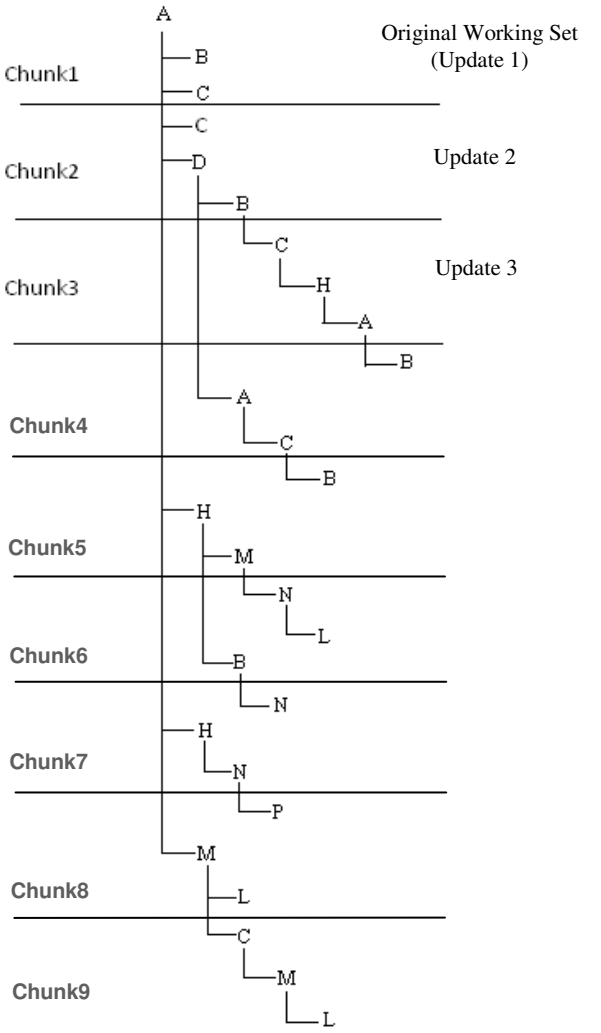

Fig. 2.An example of a routine (method) call trace

Applying the prevalence function to the methods of Chunk 1 in the trace of Figure 2 returns P(A) = 1/{(1-1)+1}*3 = 1/3. Similarly, the prevalence of *B* and *C* is also the same, i.e. 1/3, since *A*, *B*, and *C* are invoked at the same prevalence within the first chunk. Since for each of these methods in the first chunk there is no method with a better prevalence they are all ranked 1. The working set that is formed after processing the first chunk is {A, B, C}. We call the content of the first working set, an original working set and we will use it as a baseline against which new updates of the working sets are compared.

As the algorithm processes the upcoming chunks, it updates the working set by adding the newly encountered methods and by computing their prevalence. For example, the next trace chunk of the trace in Figure 2 is composed of the methods *C*, *D*, and *B*. This triggers the update of the working set by adding the method *D* and recomputing the prevalence of all methods. The new frequencies are computed as follows:

P(A) = 1 / {(2-1) + 1}*3 = 1/6.
P(B) = 2 / {(2-1) + 1}*3 = 2/6 = 1/3.
P(C) = 2 / {(2-1) + 1}*3 = 2/6 = 1/3.
P(D) = 1 / {(2-2) + 1}*3 = 1/3.

This way, the content of the working set is updated to {B, C, D, A}. This shows that the method *A*, which occupies now the last position, has gradually started to fade, whereas *B* and *C* are still present. Each time we update the working set, we compare it with the original snapshot of the working set and if there is a significant change between the original and the current one, then this suggests that there is a new phase.

In order to detect this change, we compare the methods of the current working set with the ones contained in the original working set (lines 16-18 of the algorithm). If less than a certain threshold, *T*, of the methods of the original working set appears in the current new working set, then this suggests that a phase change has taken place. Determining the threshold *T* in advance might not be possible since it might be application specific. We anticipate that a tool that supports this technique will allow enough flexibility to the users to vary the threshold according to their needs.

However, instead of comparing all the methods of the current working set, one possible optimization is to compare only a few of them that have high ranking (i.e. the ones that appear in the beginning of the set). The number of methods can be equal to the chunk size since, in the worst case scenario, the number of new distinct methods that can be found in a new chunk is less than or equal to the chunk size.

When applied to the previous example and by having the threshold randomly set to 20%, we can see that the first three methods of the original working set also appear in the beginning of the working set formed after processing the

TABLE 1.
PHASE CHANGE DETECTION

Chunk no.	Working set name	Methods introduced in chunk	Snapshots	Phase Shift Detected
1	Original working set (Snapshot 1)	A, B, C	{A, B, C}	×
2	Snapshot 2	D	{B, C, D, A}	×
3	Snapshot 3	H	{C, H, A, B, D}	×
4	Snapshot 4	No new methods	{C, A, B, H, D}	×
5	Snapshot 5	M	{M, B, C, H, A, D}	×
6	Snapshot 6	N, L	{N, L, B, C, M, H, A, D}	×
7	Snapshot 7	No new methods	{N, B, H, C, L, A, M, D}	×
8	Current working set (Snapshot 8)	P	{N, P, L, B, H, C, M, A, D}	✓

content of Chunk 2. We continue the process of updating the working set and comparing the new one to the original set as new chunks are processed. Table 1 shows the snapshots of the working sets that correspond to each chunk. Using a 20% threshold, a new phase will be detected after Chunk 8 is processed, where none of *A*, *B* or *C* appear.

B. Phase Shift Location

Once a new phase has been detected in the trace, the next step is to find the exact location of the phase transition. This is the objective of the phase shift location step. We achieve this by locating the chunk from which many methods have started to fade by observing the positions of these methods in different snapshots of the working set. We add the distinct methods invoked in all working sets up to the one in which a phase has been detected to what we call an *observation set*. The observation set resulting from adding the methods of previous example is: {A, B, C, D, H, M, N, L, P}.

Next, we need to find the chunk where many of these methods start to fade. If we consider the fading of a method *m* as it is going from its best rank (somewhere in one phase) to its worst rank (somewhere in another phase), then the starting point where the ranking of the method *m* starts to decline is in the middle. We call this point a mid-rank point and we compute it as follows:

$$midrank\ (m) = \frac{lowestrank\ (m) + highestran\,k(m)}{2}$$

Table 2 shows the methods in the observation set and their mid-rank points. For example, the method *A* has the highest ranking in Chunk 1 and the lowest ranking in Chunk 8. Its mid-rank point is therefore 4.5 (i.e., (8+1)/2). It maintains a ranking close to 4.5 when Chunk 2, Chunk 5, or Chunk 6 are being processed (see Figure 3). For each method in the observation set, we list the chunks in which the method reaches its mid-rank point (as shown in Table 2).

We call the chunks in which a method *m* reaches its mid-rank point as the *voting chunk set* of this method. This set indicated possible places where the method might start fading. For example, the voting chunk set of *A* is {Chunk2, Chunk5, Chunk6}. In other words, "*A*" could have started to disappear in either of these chunks (see Figure 3). Similarly the voting chunk set of the method B is {Chunk3, Chunk4, Chunk5, Chunk6, Chunk7} since the mid-rank of *B* is 2.5 (see Figure 4). The voting chunk set of *C* is {Chunk5, Chunk6, Chunk7} as shown in Figure 5.

To find the phase transition, we simply need to compute the voting chunk sets for all the methods of the observation set and locate the chunk that receives the highest vote. This is the chunk in which most methods of a phase have started to disappear and therefore the chunk most likely to be the

TABLE 2.
METHODS VOTING FOR CHUNKS BASED ON MID-RANK VALUES

Method call	Mid-rank	Chunks where the rank of the method is close to the mid-rank value
A	4.5	Chunk2, Chunk5, Chunk6
B	2.5	Chunk3, Chunk4, Chunk5, Chunk6, Chunk7
C	3	Chunk5, Chunk6, Chunk7
D	6	Chunk3, Chunk4
H	4	Chunk7
M	5	Chunk6, Chunk8
N	1	Chunk6, Chunk7, Chunk8
L	3	Chunk8
P	1	Chunk8

phase transition location.

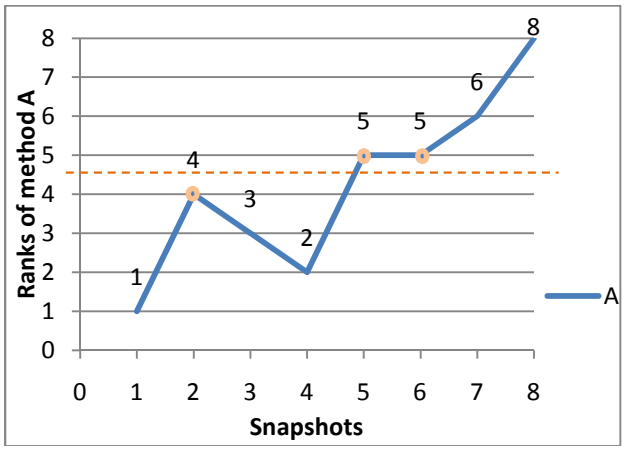

Fig. 3 The rank of method A in each snapshot
Ranks of A = (chunk1: 1, chunk2: 4, chunk3: 3, chunk4: 2, chunk5: 5, chunk6: 5, chunk7: 6, chunk8: 8) mid-rank(A) = 4.5.

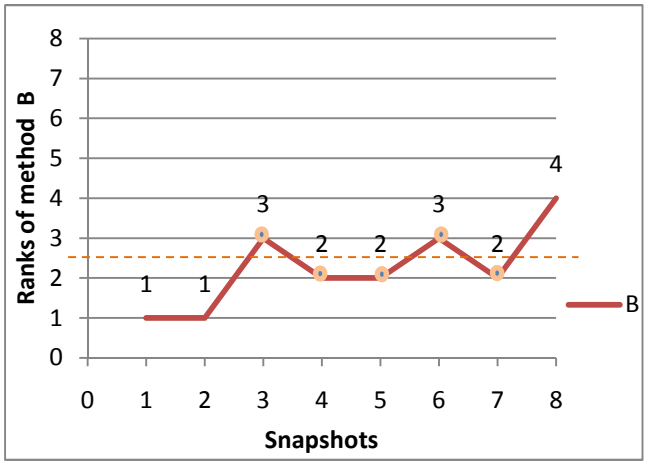

Fig. 4 The rank of method B in each snapshot
Ranks of B = (chunk1:1, chunk2: 1, chunk3: 3, chunk4: 2, chunk5: 2, chunk6: 3, chunk7: 2, chunk8: 4) mid-rank(B)= 2.5

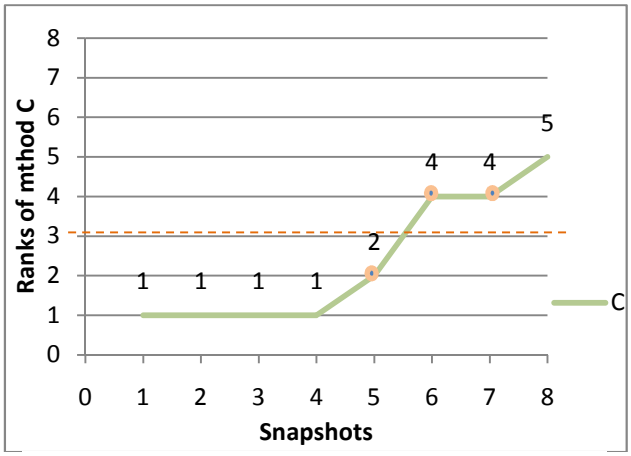

Fig. 5 The rank of method C in each snapshot
Ranks of C = (chunk1:1, chunk2: 1, chunk3: 1, chunk4: 1, chunk5: 2, chunk6: 4, chunk7: 4, chunk8: 5) mid-rank(C)= 3

Following the previous example, the result of voting is shown in Table 3. The chunk which obtained the most votes is Chunk 6, meaning that the phase transition has started from this chunk. If we look at the trace of Figure 2, we can see that from this chunk, many methods like *A*, *B*, and *C* start appearing less frequently and the new methods like *H*, *M*, and *N* start to emerge, therefore invoking a new phase.

TABLE 3
PHASE SHIFT LOCATION BASED ON MAJORITY VOTING

Chunk no.	Votes	Phase Shift
Chunk1	0	✗
Chunk2	1	✗
Chunk3	2	✗
Chunk4	2	✗
Chunk5	3	✗
Chunk6	**5**	✓
Chunk7	4	✗
Chunk8	4	✗

III. CASE STUDY

A. *Target System*

In order to evaluate the effectiveness of our algorithm, which we implemented in Java, we have applied it to a trace generated from JHotDraw 5.2 system [8]. JHotDraw is a GUI framework implemented in Java for technical and structured graphics. It consists of 9 packages, 148 classes and 1963 methods. JHotDraw 5.2 has 17,819 lines of code.

B. *Scenario Description*

To generate a simple execution trace, we used an execution scenario that involves a major activity for which we wanted to detect the phases. We applied our approach to identify the phases when exercising the scenario "draw a rectangle". The resulting trace contained 2259 calls. Since JHotDraw registers all mouse movements, and mouse movements are required while drawing a rectangle, the trace that resulted from our scenario was bound to contain a lot of noise. We have therefore filtered these mouse movements to obtain a trace that is cleaner.

We have applied our approach to the execution trace with the following parameter setting: The chunk size is 10 and the threshold value is 20%.

C. *Results*

Figure 6 shows the phase shift locations resulted from the application of our approach on the execution trace as it was generated by exercising the "draw a rectangle" scenario. It should be noted that the first phase can be detected as a result of our pre-processing step (i.e. removing utilities [9]) which gives us a sparse section in the beginning of the execution trace (the end of this phase is demarcated

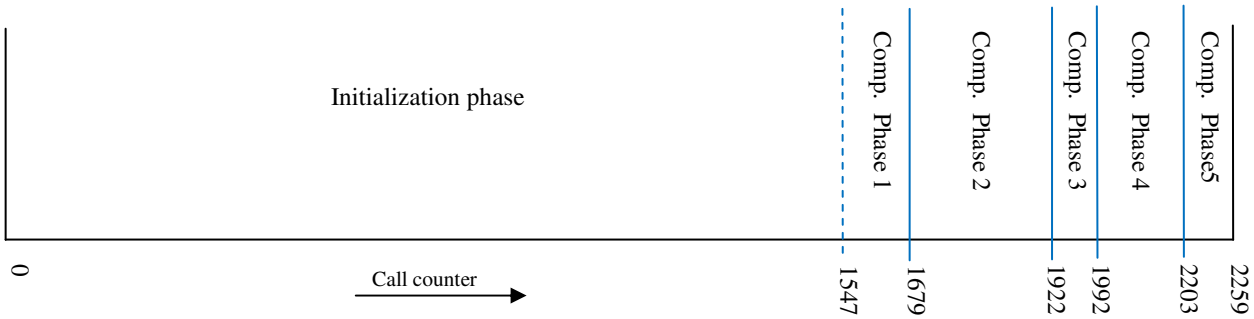

Fig. 6. The phase shift locations of JHotDraw 5.2 (scenario: draw rectangle)

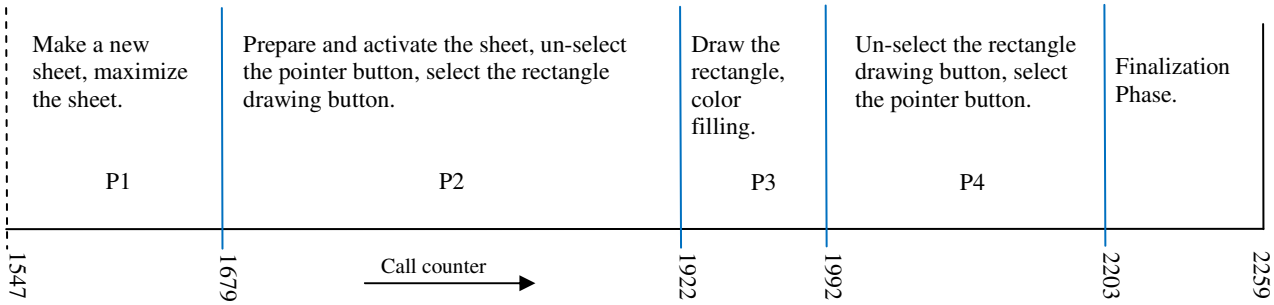

Fig. 7. Focused view of the computation phases

by a dashed line). This leads us to the assumption that the first phase is the initialization phase of the program. To verify this, we checked this phase against the original execution trace manually. The result of this verification confirms the correctness of our assumption. The solid lines on the right-hand side of the dashed line show the phase shift locations as a result of applying our algorithm on the filtered trace. Our preliminary assumption is that these phases are the computational phases of the program. As a means of verification, we focused on these phases to have a more detailed view.

Figure 7 shows a focused view of the phases that follow the initialization phase. This closer investigation against the documentation of JHotDraw reveals the most important phases that contribute to the major activities of the program while drawing a rectangle. For instance, after mapping Phase P3 to the execution trace, we were able to find that `figures.RectangleFigure.drawBackground` and `figures.RectangleFigure.drawFrame` are the most important methods specific to drawing the rectangle. Figure 7 also shows three other major phases and their important activities resulting from mapping those phases to the execution trace. Furthermore, we found out that the last phase is a Finalization phase in which the sheet and then the application are closed. This confirms the effectiveness of our approach in detecting the essential phases involved in the implementation of drawing a rectangle in JHotDraw.

As mentioned earlier, while we locate the phases inside a trace, the chunk where the phase is located is the chunk that earns the majority of votes which can also expressed as follows:

$$votes_{max} = max(votes_o..votes_d)$$

where, $votes_0$ and $votes_d$ correspond to the number of votes for the original chunk and the chunk where the algorithm detects that a phase shift has taken place. To be able to evaluate the obtained phase shift locations are adequate, we introduce in this case study the metric of confidence. The confidence of a chunk being a phase transition location is the strength of its votes relative to the other candidate chunks. A larger difference between the highest vote and the average vote shows a stronger vote for the phase transition chunk: (i.e, a bigger $Lapse_r$)

$$Avg_r = \frac{\sum_{i=0}^{d} votes_i}{2}$$

$$Avg_r = \frac{\sum_{i=o}^{d} votes_i}{d - o + 1}$$

$$Lapse_r = votes_{max} - Avg_r$$

where, $Lapse_r$ is the difference between the highest vote ($votes_{max}$) (which is the number of votes of the phase transition chunk) and the average vote of a chunk (Avg_r) as resulted by the voting.

Now to be able to compare the strength of the votes of all phase transition chunks with one another the value of $Lapse_r$ needs to be normalized. For this we need to obtain the biggest possible difference between the highest vote and the average vote of a chunk which is the case where all votes are for our selected phase transition chunk and other chunks have zero votes. This can be calculated as follows:

$$Avg_s = \frac{votes_{max}}{d - o + 1}$$

$$Lapse_s = votes_{max} - Avg_s$$

$$Confidence = \frac{Lapse_r * 100}{Lapse_s}$$

where, Avg_s is the average of the votes in the special case where the chunk with the highest number of votes is considered to have 100% of the votes. $Lapse_s$ is the difference between the highest vote and the average vote of a chunk (Avg_s) in the stated special case. The $Confidence$ is the percentage of $Lapse_r$ over $Lapse_s$.

For example in our case study, the results of voting for finding the chunk of phase transition between phase P1 and phase P2 is shown in Table 4. In this example, the total number of votes is 55 and the highest vote is 15 (Chunk86). If these 55 votes were distributed equally between the 11 chunks, each chunk could have an average of 4.54 votes. Thus, Chunk86 with 15 votes has 10.64 votes more than an average vote that a chunk could have. If we consider the case where the votes of Chunk86 (15 votes) was exactly 100% of the votes, the average votes of each chunk would be 1.34 votes (i.e. 15 / 11) and consequently the biggest possible difference between the highest vote and the average vote would be 13.64. Therefore, the confidence of the Chunk86 to be the phase transition location is (10.64 * 100) /(13.64) which is 76.6%.

Table 5 shows the chunk number corresponding to each phase location and their confidence level according to the equations given above. Based on the results obtained, we can notice the high confidence of the first two phase locations and the other two confidence levels with comparatively less confidence level.

TABLE 4
RESULTS OF VOTING FOR PHASE SHIFT LOCATION BETWEEN P2 AND P3

Chunk no.	Votes
Chunk 77	1
Chunk 78	0
Chunk 79	0
Chunk 80	3
Chunk 81	5
Chunk 82	5
Chunk 83	6
Chunk 84	3
Chunk 85	4
Chunk 86	15
Chunk 87	8

IV. CONCLUSION AND FUTURE WORK

To help maintainers understand a program through analysis of its execution trace which could be inherently large, we present a novel technique for simplifying the analysis of execution traces by devising an algorithm that can divide a trace content into various fragments that correspond to the execution phases of a program that implement specific tasks such as initializing variables, performing a particular computation, etc.

The idea behind our phase detection approach is to detect when and where during the execution of a program, the methods that implement a particular phase start to disappear as new methods begin to emerge, indicating the beginning of another phase. Therefore, our approach finds phases in two distinct and consequent steps: detecting the phase change by alerting when the most frequent methods become less frequent as the program enters a new phase (Step 1) and indicating when the phase change has started to take effect by showing an estimated location of phase transition (Step 2).

Our proposed approach is an online phase detection

TABLE 5
THE RESULTS OF OUR PHASE FINDING APPROACH FOR THE CASE STUDY

Phases	Phase-Transition Locations	No. of Chunks	MaxVotes	Avg_r	$Lapse_r$	Avg_s	$Lapse_r$	Confidence
P1- P2	Chunk 74	11	15	5.36	9.64	1.36	13.64	70.6%
P2- P3	Chunk 86	11	15	4.54	10.64	1.36	13.64	76.6%
P3- P4	Chunk 92	7	12	7.7	4.3	1.71	10.29	41.7%
P4- P5	Chunk 100	11	11	6.27	10	1	4.73	47.3%

technique that detects the phases and the locations of phase transitions while the trace is being generated. This is contrasted with an offline approach, where the entire trace is first collected before applying the algorithm. Online processing of traces is usually more desirable than an offline approach since the users can see the results early and may need to make decisions based on this early feedback without having to wait until the entire trace is generated.

There are a number of open questions about the two parameters of our phase finder algorithm. For example, what chunk size should be used? What is a good threshold to find phases in a program? Determining appropriate parameter that balances the computation overhead with the accuracy of the approach is a topic that we intend to tackle in the future.

REFERENCES

[1] I. Carmichael, V. Tzerpos, and R. C. Holt, "Design Maintenance: Unexpected Architectural Interactions (Experience Report)", *In Proceedings of the International Conference on Software Maintenance,* pp. 134-137, 1995.

[2] O. Greevy, and S. Ducasse, "Correlating features and code using a compact two-sided trace analysis approach", *In Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering,* pp. 314–323, 2005.

[3] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering behavioral design models from execution traces", *In Proceedings of the 9th European Conference on Software Maintenance and Reengineering,* pp. 112 – 121, 2005.

[4] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency", *In Proceedings of the 8th European Conference on Software Maintenance and Reengineering,* pp. 329–338, 2004.

[5] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patternsin Object Oriented Visualization", *In Proceedings of the 4th Conference on Object Oriented Technologies and Systems,* pp.219-234, 1998.

[6] D. Jerding, J. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", *In Proceedings of the 19th International Conference on Software Engineering,* pp. 360-370, 1997.

[7] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, A. J.J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views", *The Journal of Systems & Software,* 81(12), pp 2252-2268, 2008.

[8] JHOTDRAW website: http://www.jhotdraw.org/.

[9] A. Hamou-Lhadj, "Techniques to simplify the analysis of execution traces for program comprehension", *Ph.D. Dissertation*, School of Information Technology and Engineering, University of Ottawa, 2005.

[10] M. Hind, V. T. Rajan, and P. F. Sweeney, "Phase Shift Detection: A Problem Classification" IBM Research Report # 22887, 2003.

[11] P. Nagpurkar, M. Hind, K. Chandra, P. F. Sweeney, and V. T. Rajan, "Online Phase Detection Algorithms", *In Proceedings of the International Symposium on Code Generation and Optimization,* pp. 111 - 123, 2006.

[12] D. Gu, C. A. Verbrugge, "A Survey of Phase Analysis: Techniques, Evaluation and Applications", *Sable Technical Report* No. 2006-1, 2006.

[13] S. P. Reiss, "Dynamic detection and visualization of software phases", *In Proceedings of the 3rd ICSE Workshop on Dynamic analysis (WODA),* pp. 1–6, 2005.

[14] S. P. Reiss, "Visual representations of executing programs", *Journal of Visual Languages and Computing* 18, 2, 2007.