

Phase Flow Diagram: A New Execution Trace  
Visualization Technique

Arya Shafiee

A Thesis

In

The Department

Of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Applied Science (Software Engineering) at

Concordia University

Montreal, Quebec, Canada

September 2013

© Arya Shafiee, 2013

# CONCORDIA UNIVERSITY

## School of Graduate Studies

This is to certify that the thesis prepared

By: Arya Shafiee

Entitled: Phase Flow Diagrams: A New Execution Trace Visualization  
Technique

and submitted in partial fulfillment of the requirements for the degree of

### Master of Applied Science (Software Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Brigitte Jaumard	Chair
Dr. Olga Ormandjieva	Examiner
Dr. Joey Paquet	Examiner
Dr. Abdelwahab Hamou-Lhadj	Supervisor
Dr. Constantinos Constantinides	Supervisor

Approved by:

Director

Chair of Department or Graduate Program

\_\_\_\_\_ 20\_\_

\_\_\_\_\_  
Dean of Faculty

# ABSTRACT

## Phase Flow Diagram: A New Execution Trace Visualization Technique

Arya Shafiee

Software maintenance tasks are known to be costly and challenging. The main challenge is that software maintenance must understand how the software system works before making any changes to it. This is due to lack of adequate documentation if it exists at all. Program analysis techniques aim to reduce the impact of this problem. In this thesis, we focus on the ones that permit the understanding of the behavioural aspects of software. These techniques operate on execution traces, generated from the system under study.

Traces are difficult to work with because of their size. One way to reduce their complexity is to automatically divide their content into meaningful clusters, each representing a particular execution phase. This is known as trace segmentation. Trace segmentation research is relatively new. The focus has been on building robust algorithms that achieve acceptable accuracy.

In this thesis, we introduce a new trace visualization technique called Phase Flow Diagram to represent the execution phases and the relationship between them in a visual manner. The diagram has a number of notations that can be used by software engineers to represent a trace as a flow of execution phases instead of mere events. We introduce a supporting tool for the diagram. The new diagram and the tool are validated through a user study that involves several users.

## Acknowledgements

I would like to express the deepest appreciation to my supervisor, Dr. Abdelwahab Hamou-Lhadj for providing me great support in all hard conditions, for the motivation, enthusiasm and engagement in every moment of this entire journey. I also want to thank him for providing me with incredible ideas and resources through the learning process of this thesis. Completion of this Master thesis was not possible without his precious and insightful comments. My thanks and appreciations also go to my co-supervisor Dr. Constantinos Constantinides for the useful comments and remarks which helped me in completion of this thesis. His valuable feedback and advice has been precious in shaping the results.

Furthermore, I have had the pleasure to work with Dr. Heidar Pirzadeh. I would like to thank him for introducing me to the topic, as well for the support on the way.

I wish to express my gratitude to my lab mates for their support through the duration of my studies.

I like to thank my parents, Soosan and Hossein, and my little sister, Arghavan, for their support during my life and for the strength they gave me to reach to this level.

I would like to thank my wife, Viola, for her patience and encouragement and for her endless support throughout my hard times of this thesis. I could not have achieved this without your amazing kindness and help. I love you forever.

## List of Figures

Figure 2.1. Example of trace of routine calls.....	7
Figure 2.2. SEAT screen snapshot.....	11
Figure 2.3. Hyades: Execution statistics view (from [Hya10]) .....	12
Figure 2.4. Hyades: Showing a trace as a sequence diagram .....	12
Figure 2.5. ISVis sequence diagram view with vertical mural view .....	13
Figure 2.6. ISVis horizontal mural view .....	14
Figure 2.7. Extravis Circular Bundle View .....	14
Figure 2.8. ALMOST combines both a mural view and a spiral view .....	15
Figure 2.9. Using metaphors to represent interactions among the system components. ..	16
Figure 3.1. Main phases of a program .....	18
Figure 3.2. Overview of TSR approach (from [PHS11c]).....	19
Figure 3.3 Sample trace (from [PHS11c]) .....	20
Figure 3.4 Similarity Gravity method result (from [PHS11c]).....	20
Figure 3.5 Trace with nesting levels (from [PHS11c]).....	21
Figure 3.6 Continuity Gravity method result (from [PHS11c]).....	21
Figure 3.7. Thread notation sample .....	27
Figure 3.8. Main flow start notation sample.....	28
Figure 3.9. Meta model of Phase Flow Diagram .....	28
Figure 3.10. Main flow end notation sample .....	29
Figure 3.11. Sub-Flow start notation sample .....	30
Figure 3.12. Sub-Flow end notation sample .....	31
Figure 3.13. Phase notation sample .....	33
Figure 3.14. Ordered Flow sample .....	34
Figure 3.15. Original flow of phases .....	35
Figure 3.16. Ordered Self-Flow notation sample.....	35
Figure 3.17. Fork notation sample .....	36
Figure 3.18. Join notation sample .....	38
Figure 3.19. Inter-Thread Order Flow sample 1 .....	39
Figure 3.20. Inter-Thread Order Flow sample 2 .....	40
Figure 3.21. Inter-Thread Order Flow sample 3 .....	40
Figure 3.22. Comment notation sample.....	42
Figure 3.23. Phase types sample.....	43
Figure 3.24. Flow of phase with super phases .....	45
Figure 3.25. Sub-phases of P1 .....	45
Figure 3.26. Flow of phases.....	46
Figure 3.27. Flow of phases with a manually unified phase.....	46
Figure 3.28 TSR Class Diagram .....	48

Figure 3.29 PFD Class Diagram .....	50
Figure 3.30. Phases on Complete Tree Editor .....	53
Figure 3.31. Phase flow diagram editor .....	54
Figure 3.32. Synchronized Editors; Left: Phase Editor, Right: Complete Tree Editor ....	54
Figure 3.33. Phase Editor and Phase View .....	55
Figure 3.34. Phase Property View .....	56
Figure 3.35. Phase representative elements view .....	57
Figure 3.36. Search View.....	57
Figure 4.1 Average Score / Groups/ Experience .....	70
Figure 4.2 Average of participants' scores to statements .....	71

## List of Tables

Table 3.1. Phase types.....	42
Table 3.2. Phase operations .....	47
Table 4.1. Participants Average Experience .....	59
Table 4.2. Expertise of participants .....	60
Table 4.3. Feature-Oriented Tasks and Number of Successful Users .....	61
Table 4.4. Successful users in each level of expertise .....	63
Table 4.5. Goal-Oriented Tasks and Average Results .....	64
Table 4.6. Goal-oriented tasks scores per participant expertise level.....	65
Table 4.7. Average of participants responses to statements .....	67

# Table of Contents

<b>List of Figures</b> .....	<b>v</b>
<b>List of Tables</b> .....	<b>vii</b>
<b>Chapter 1 INTRODUCTION</b> .....	<b>1</b>
1.1 Motivations .....	1
1.2 Research Contributions .....	4
1.3 Thesis Outline .....	5
<b>Chapter 2 BACKGROUND</b> .....	<b>6</b>
2.1 The Concept of Execution Traces .....	6
2.2 Software Maintenance and Program Comprehension .....	8
2.3 Trace Visualization Research .....	10
<b>Chapter 3 PHASE FLOW DIAGRAM</b> .....	<b>17</b>
3.1 The Concept of Execution Phases .....	17
3.2 Building the Phase Flow Diagram .....	23
3.2.1 Guiding Principles .....	23
3.2.2 Investigating Existing Diagrams .....	25
3.2.3 PFD Metamodel and Notation .....	26
3.2.4 Phase Operations .....	44
3.3 Tool Support .....	47
<b>Chapter 4 EVALUATION</b> .....	<b>58</b>
4.1 Participants .....	58
4.2 Trace File .....	60
4.3 Experiment Process .....	60
4.3.1 Training .....	61
4.3.2 Feature-Oriented tasks .....	61
4.3.3 Goal-Oriented Tasks .....	63
4.3.4 Questionnaire .....	66
4.4 Limitations .....	72
<b>Chapter 5 CONCLUSION AND FUTURE WORK</b> .....	<b>73</b>



5.1 Review of the Research .....	73
5.2 Contributions Highlights.....	74
5.3 Future Work .....	75
<b>References .....</b>	<b>77</b>

# Chapter 1 INTRODUCTION

## 1.1 Motivations

Software maintenance is perhaps one of the most challenging software engineering activities. An important step before performing a maintenance task is to understand the software system under study. To achieve this, software maintainers, normally, would access any available documentation, and/or consult with the original designers of the system. In most cases, however, system documentation is rarely up to date. One cannot rely on the original developers either. Most of them move to new projects and even new companies, taking with them important knowledge about the system.

To address this challenge, software engineers tend to resort to the analysis of the source code. Source code analysis techniques can be grouped into categories, static and dynamic analysis. These approaches vary depending on whether the analysis requires the execution of the system or not [Cor 89].

Static analysis consists of analyzing the code without executing it. The idea is to identify various system components and how they are interconnected. This could be done using reverse engineering tools. The advantage of static analysis is that it provides a complete coverage of the system. However, this can also be an inconvenience if only partial understanding of the system is needed. For example, understanding how a particular feature is implemented might not require the analysis of the entire system. Static analysis suffers from limitations when it comes to understanding the behavioural aspects of a

system. Dynamic binding, parallel executions, and other mechanisms make it almost impossible to rely solely on static analysis of the code.

Dynamic analysis, which is the main topic of this thesis, focuses on understanding the execution of the system. A common practice is to run an instrumented version of the system, generate execution traces, and examine them. Execution traces contain wealth of information that can reveal important facts about the system [Bal99]. Dynamic analysis is a preferred approach when it is necessary to link system output to input data [Bal99]. Debugging tasks can be made easier if this link is established. On the other hand, dynamic analysis suffers from the problem of incompleteness since during execution there is no guarantee that all the program paths are taken. A common solution is to combine both static and dynamic analyses techniques at various degrees depending on the task at hand.

Run-time information is typically represented in the form of execution traces. Example of traces include traces of function calls, statement-level traces, inter-process communication traces, etc. The challenge with working with traces is the large amount of data they contain. There is a need to develop techniques to simplify the analysis of trace content.

In recent years, there has been a noticeable increase in the number of trace analysis and abstraction techniques and tools (e.g. [CHZ+07, CMZ08, CZD+09, Dug07, HL02, HL06]). The common practice is to extract high-level concepts from low-level trace events. By doing this, software engineers can focus on the important elements conveyed in a trace before deciding to dive into the details.

In their recent work, Pirzadeh et al. proposed a new approach for trace abstraction based on the concept of execution phases [PH11a, PH11b, PHS11c]. The idea is to segment a trace into meaningful clusters that represent distinct execution (computational) phases of the traced scenario. An execution phase, for example, could be the execution of a particular algorithm. Recovering such information from a trace is important to software engineers. This way, they do not have to ever look at the trace as a low-level stream of events but rather as a flow of computational phases.

The authors, however, recognize that their approach can only be adopted in practice if it is embedded in a usable trace analysis tools. The problem is that most existing analysis tools are not designed to support execution phase identification and rendering.

In this thesis, we fill in this gap by proposing a new visualization diagram, called Phase Flow Diagram, which has the unique purpose of representing execution phases extracted from execution traces. Execution phases help to understand the content of execution traces, which in turn can facilitate maintenance tasks that require understanding of the feature under study as shown by Pirzadeh et al. [PH11b]. We also built a tool that supports this new diagram. A user-centric study is conducted to evaluate (a) the usefulness of execution phases and the phase flow diagram on quickly exploring the trace content to understand it, and (b) the usability of the tool and the diagram in maintenance activities.

## 1.2 Research Contributions

The work presented in this research contributes to trace abstraction research in the following ways:

- We implemented the components of the trace segmentation approach proposed by Pirzadeh et al. [PH11a] including the gravitational schemes, BIC-supported K-means clustering, the extraction of relevant information, the removal of utilities, and the detection of similar phases. The TSR approach will be discussed in detail in Chapter 2. The implementation of the approach will be explained in Chapter 3.
- We have developed a new visualization technique, called the Phase Flow Diagram (PFD), to represent the execution phases invoked in a trace. PFD is designed to take into consideration the various ways trace phases can be used to explore the content of a trace. The diagram and its implementation will be discussed in Chapter 3.
- We have conducted user-centric studies to evaluate the effectiveness of PFD in helping software engineers understand large traces.
- We have developed a new Eclipse plug-in to support the rendering of execution traces using PFD. The plug-in is embedded with SEAT (Software Exploration and Analysis Tool), which is a trace abstraction and analysis tool created by members of the Software Behaviour Analysis Research Lab at Concordia University in collaboration with university of Ottawa.

## 1.3 Thesis Outline

This thesis is organized as follows:

- **Chapter 2: Background**

In this chapter, we provide the necessary background to understand the research work presented in this thesis. We introduce the concepts of software maintenance and program comprehension. We will also survey related work including the work on trace abstraction and analysis for program comprehension.

- **Chapter 3: Phase Flow Diagram**

This is the core chapter of the thesis. We present the concepts of Phase Flow Diagram including the detailed notation. We will also discuss existing algorithms for extracting execution phases from large traces. The chapter also presents a tool that supports FDPs.

- **Chapter 4: Experiment**

For evaluating the usefulness of PFD and the supporting tool, we conducted a user study. In this chapter we will report on the user study's process and results.

- **Chapter 5: Conclusion and Future Work**

In this chapter we review our contributions and discuss future directions and opportunities.

## Chapter 2 BACKGROUND

### 2.1 The Concept of Execution Traces

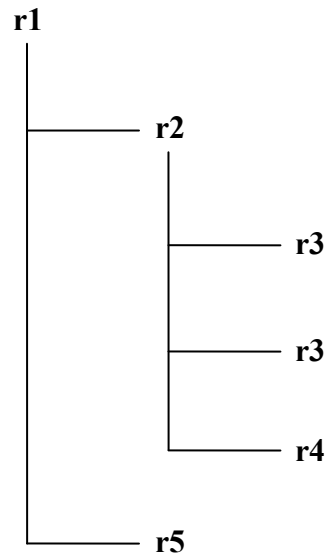
An execution trace is simply a stream of events generated by executing the system under study [Bal99]. There are various aspects of a system that can be traced. For example, one can trace the routine calls, statements, or both. In a distributed system, it is common to trace messages exchanged among processes. The trace type depends on the objective of the task for which tracing is needed.

In this thesis, we put an emphasis on traces of routine calls. We define a routine as any function, method, procedure, etc. Routines are perhaps the most important components of a software system be it object-oriented or not. They encapsulate the computations done by the system to solve the problem for which the system is built. They are therefore important for understanding what the system does in the absence of documentation or other sources of information.

It is also possible to trace statements in addition to routines. But this will result in extremely large traces that would be difficult to manage. Most researchers in the area of trace abstraction for program comprehension seem to agree that routine calls strike an adequate balance between richness of information (needed to understand the code) and complexity in terms of trace size [PH11b].

An event in a routine call trace has a number of attributes (some are optional) including the full name of the routine, the nesting level, timestamp, the thread in which the event

occurs, etc. A trace of routine calls can be represented as a tree structure as shown in Figure 2.1. In this example, there are five routines (r1, r2, r3, r4, r5) and five calls (r1, r2), (r2, r3), (r2, r3), (r2, r4), and (r1, r5).



**Figure 2.1. Example of trace of routine calls**

To generate a trace, the system needs to be instrumented. The idea is to insert probes in the code. A probe is simply a printout statement. This is known as code instrumentation. Code instrumentation is simple and automated but it can also add overhead to the system. It is intrusive, which means it may change the initial code if not done with care.

There are other means to instrument the system. A common one is to use aspect-oriented programming. Probes are added at the object level. This way, the code remains unchanged. Another alternative is to instrument the operating system (or the virtual machine). Finally, it is also possible to use the debugger to generate a trace. The idea is to



add break points in places of interest. This technique is known to slow down considerably the system and it is not recommended.

## 2.2 Software Maintenance and Program Comprehension

Software maintenance is defined as the process of modifying a software system after it is released [IEE98]. Maintenance tasks can be grouped into four categories:

- Adaptive maintenance: these activities are performed to adjust the system due to changing external environments.
- Corrective activities: these activities deal with fixing discovered problems and bugs.
- Perfective activities: this type of activities is concerned with changing the system to enhance or add new features to the system.
- Preventive activities: the goal here is to improve the quality of the system to prevent defects.

Literature has showed that the analysis of execution traces can help in many maintenance tasks. Jerding et al. [JR97], for example, showed the usefulness of analyzing execution traces in facilitating both corrective and adaptive maintenance tasks. Silva et al. [SPAM11] conducted experiments that show how the analysis of execution traces can help in perfective maintenance tasks. Cornelissen et al. [CZD11] also conducted a controlled experiment with a number of maintenance scenarios to evaluate how their trace analysis approach can improve the performance in terms of time spent on corrective maintenance.

As previously mentioned, software maintenance tasks can be very challenging due to lack of adequate documentation. Software engineers must understand how the system is implemented before making any changes. Basili showed that 50-60% of software engineering effort is spent on understanding the code [Bas97]. Similarly, Von Mayrhauser et al. [VV95] argued that for almost every type maintenance tasks software engineers need to understand the system (even if it is a partial understanding) before they can proceed.

Program comprehension is a research field in which researchers examines how software engineers understand a system (usually in the absence of documentation and other reliable sources of information). The goal is to build automated solutions that can accelerate the understanding process.

Many program comprehension models have been proposed in the literature to explain how software engineers understand programs [Pen87, VV95, Sto06]. The top-down model is a model in which programmers precede their understanding of the code in a top-down fashion. At the top level, they start by making assumptions about the code based on the knowledge they already have about the system under maintenance. They try to validate these assumptions at the down level by looking at the code to find a class, function or a line of code which could be the starting point of a specific functionality they are looking for. This model is often followed by software engineers who are somewhat familiar with the system domain

The bottom-up model is the opposite. Software engineers read the code (and accompanying comments) looking for hints that can help them identify what the code does. They group mentally these clues to form higher-level concepts.

In practice, it has been shown that most software engineers follow an integrated approach in which top-down and bottom-up strategies are both followed depending on the code, domain knowledge, etc.

In this thesis, we take into account these models. However, instead of using the source code, we use execution traces as the artefact that is analyzed by software engineers. Trace analysis techniques are used to vary the level of details of trace content. As such, they enable both bottom-up and top-down comprehension strategies. Software engineers can use abstractions to develop assumptions about what goes on in a trace and then dig into the details as needed (top-down). Another alternative would be that they examine low-level trace events to extract higher-level concepts (bottom-up). We anticipate that shifting between these strategies is possible depending on the complexity of the trace, the expertise of the users, etc.

### 2.3 Trace Visualization Research

In this section we summarize the main studies in which visualization is used to simplify the analysis of large traces.

Hamou-Lhadj et al. presented SEAT (Software Exploration and Analysis Tool), a tool that permits the analysis of routine call traces [HFL04, HLF05]. The tool contains many features including a new tree widget referred to as PictureTree. PictureTree has three states: An expand state (+), a collapse state (-), and an intermediate state (~). The intermediate state is used to show subtrees for which some elements are hidden (no needed for the analysis at hand). Users can therefore control the level of details viewed in each subtree. The tool also supports an exchange format, called CTF (Compact Trace

Format) [HL12] that facilitates the representation of large routine calls. Figure 2.2 shows the main screen of SEAT. Several views are used to assist software engineers in navigating the trace.

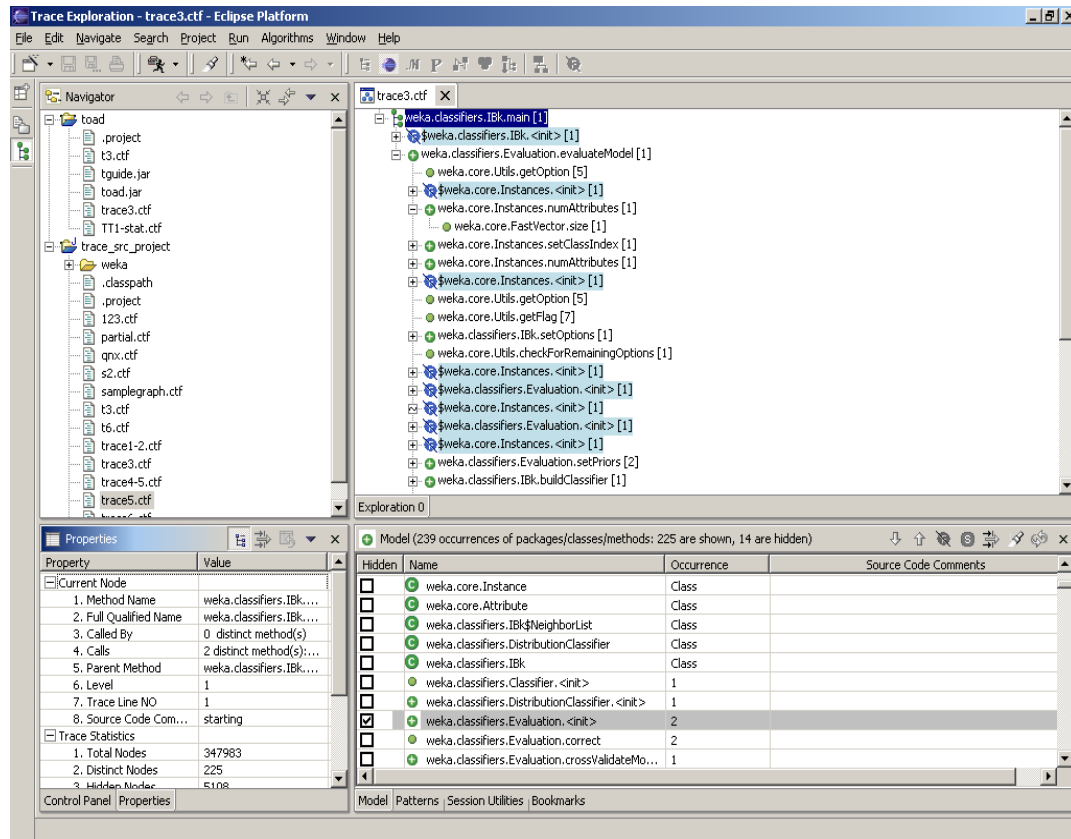


Figure 2.2. SEAT screen snapshot

Hydaes [Hya10] is an Eclipse plug-in, developed in the context of the Test & Performance Tools Platform Project (TPTP). Hydaes is used to analyze traces for the purpose of testing and performance analysis. It uses a table to represent different quantitative attributes of a trace. In this table, the top ten methods with the highest base time and their execution related information are represented (Figure 2.3). To visualize

large traces, Hydaes uses sequence diagrams. However, the tool does not support any abstraction mechanisms. In other words, it is up to the users to navigate the trace and use filtering capabilities to eliminate unneeded data (Figure 2.4).

Highest 10 base time							
Method	Class	Package	<Base Time ...	Average Base Time...	Cumulative Time...	Calls	
instance(int) weka.core.Instance	Instances	weka.core	14.732695	0.000127	23.893199	115868	
setInstances(weka.core.Instances) void	PreprocessPanel	weka.gui.explorer	13.037805	13.037805	13.037824	1	
find(java.lang.Class, java.lang.String) java.util.Vector	ClassDiscovery	weka.core	10.677044	0.197723	13.580628	54	
showDialog(java.awt.Component, java.lang.String) int	ConverterFileChooser	weka.gui	10.066024	5.033012	10.066812	2	
elementAt(int) java.lang.Object	FastVector	weka.core	9.721956	0.000073	9.721956	132964	
Main()	Main	weka.gui	8.579389	8.579389	8.579389	1	
classIndex() int	Instance	weka.core	6.773018	0.000156	7.821481	43406	
partition(int, int, int) int	Instances	weka.core	6.463654	0.001197	20.036775	5400	
value(int) double	Instance	weka.core	5.715326	0.000059	5.715326	96806	
LogWindow()	LogWindow	weka.gui	5.499440	5.499440	5.734171	1	

Figure 2.3. Hydaes: Execution statistics view (from [Hya10])

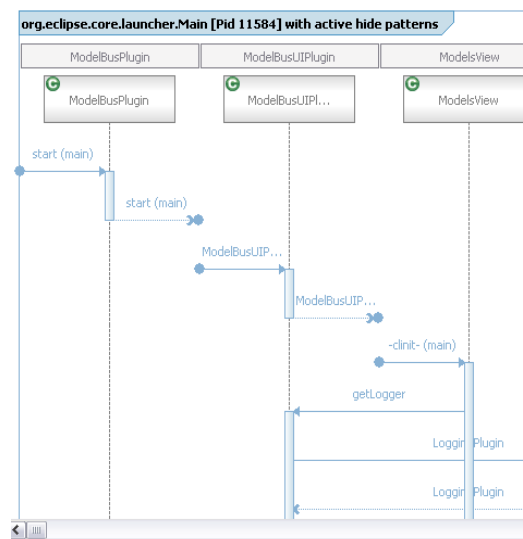


Figure 2.4. Hydaes: Showing a trace as a sequence diagram

Extravis is a new tool that offers an overview of the execution trace through a mural view (Figure 2.7) called circular bundle view. The idea is use colors and lines to fit large portions of a trace into the display.

The idea of mural views is not novel. It was used before by Jerding et al. in their tool called ISVis [JR97]. The authors combined a mural view with a sequence diagram (called temporal message-flow diagram by the authors) to allow analysts to navigate through portions of the trace. The mural view (two types: horizontal and vertical murals – see Figure 2.5 and Figure 2.6) is also useful to show patterns of sequences of events. In ISVis, these patterns are distinguished using color coding schemes. A user can zoom in and out on a pattern. The sequence diagram view is updated accordingly.

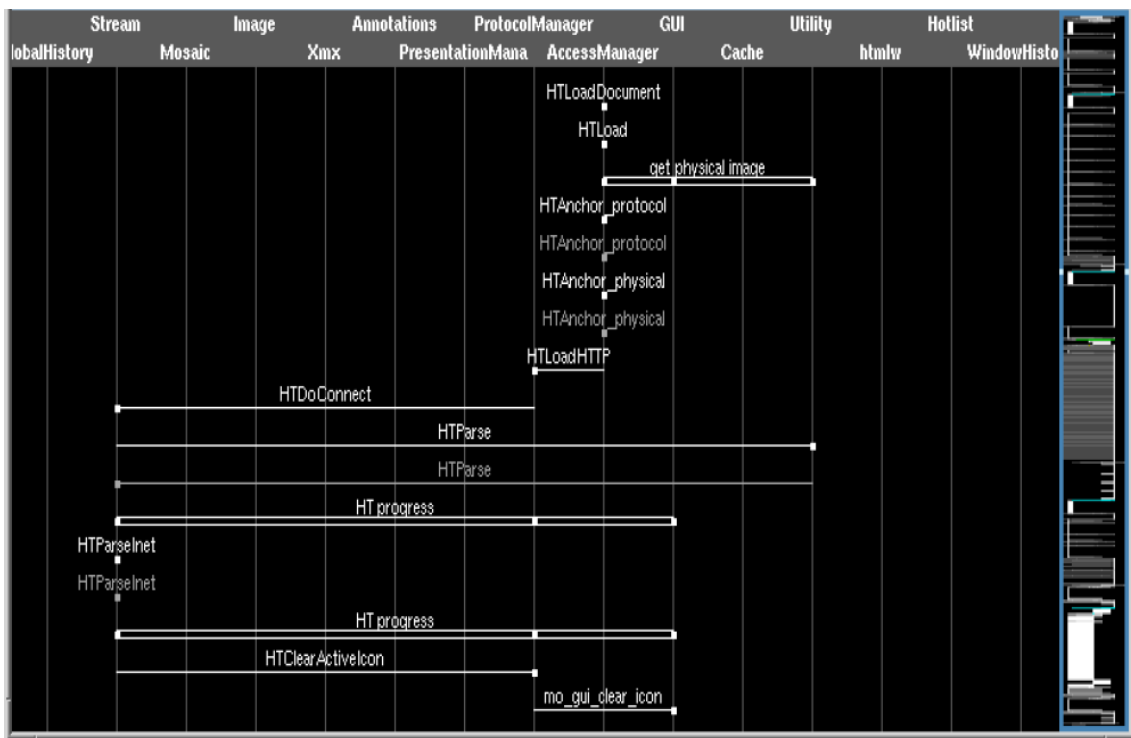


Figure 2.5. ISVis sequence diagram view with vertical mural view

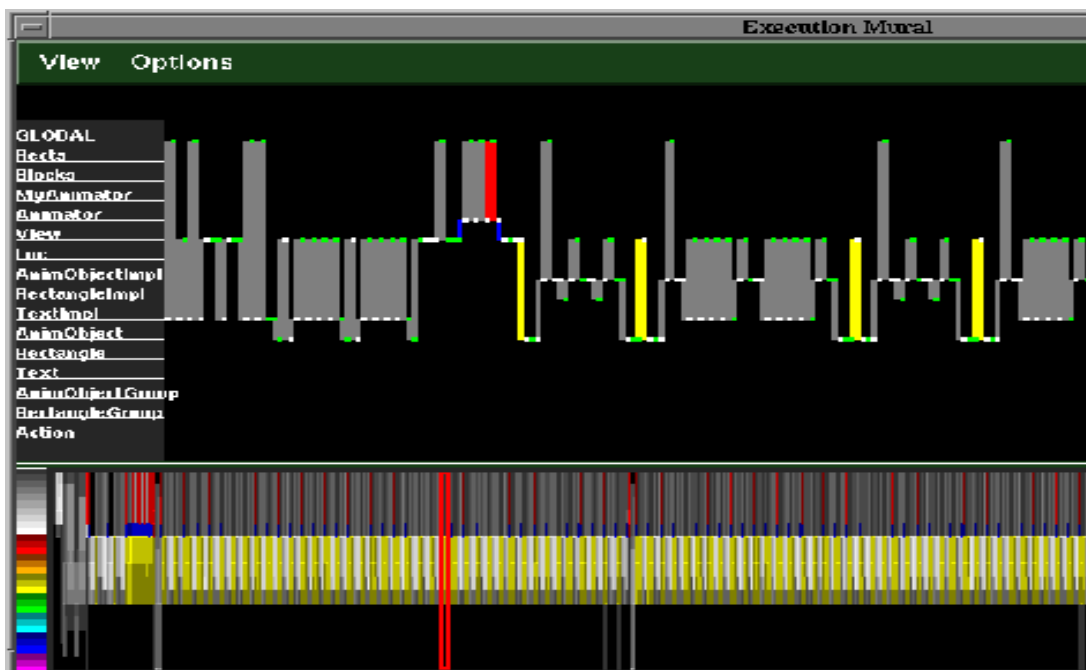


Figure 2.6. ISVis horizontal mural view

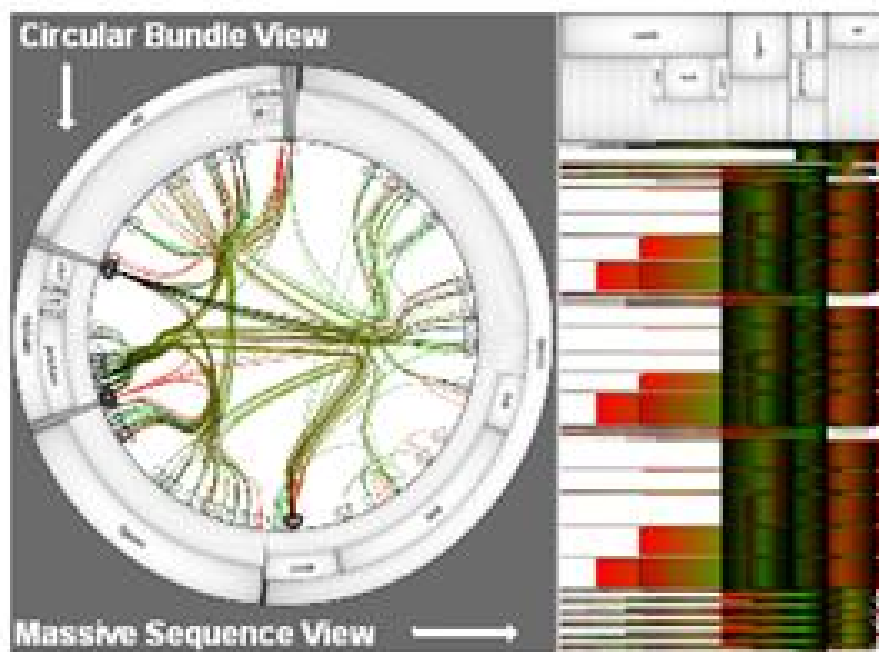


Figure 2.7. Extravis Circular Bundle View

ALMOST is another trace analysis tool. It was proposed by Renieris et al. [RR01, RR99]. It uses a mural representation just like ISVis. The representation is however mapped to a spiral view (Figure 2.8) and not a sequence diagram.

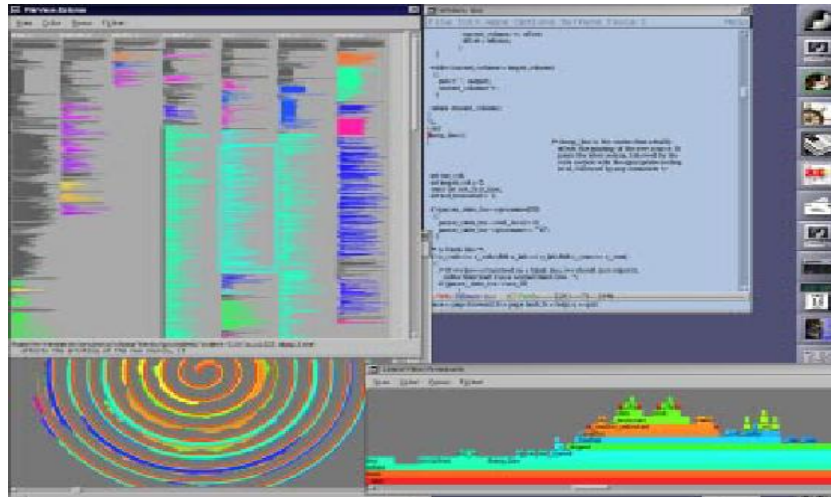


Figure 2.8. ALMOST combines both a mural view and a spiral view

Ovation by De Pauw et al. [DHKV 93, DJM+02] incorporates a zoomed-out execution mural-like view to show the highest level of the execution of a program. This view gives a general idea of the different phases in the program while reflecting the stack depth at each particular phase as the width of the pattern. Color coding is used in this view to indicate the classes that are widespread in each phase.

There are many other visualization techniques; many of them are quite innovative. Many researchers have used real-world visual metaphors such as buildings, cities, and planets to show traces. Virtual reality has also been exploited. Some examples include the work of Dugerdil et al. and Alam et al. [DA08, AD07] who used the city metaphor (originally proposed by Knight [Kni00]) in which the classes and files are shown as buildings in a 3D city landscape. Interaction among these components is shown. Zooming in on a



building makes it possible to see the methods and variables inside it or to check the corresponding part of source code. The user can also see relationships between the selected buildings (or methods inside buildings) as directed pipes between them (Figure 2.9).

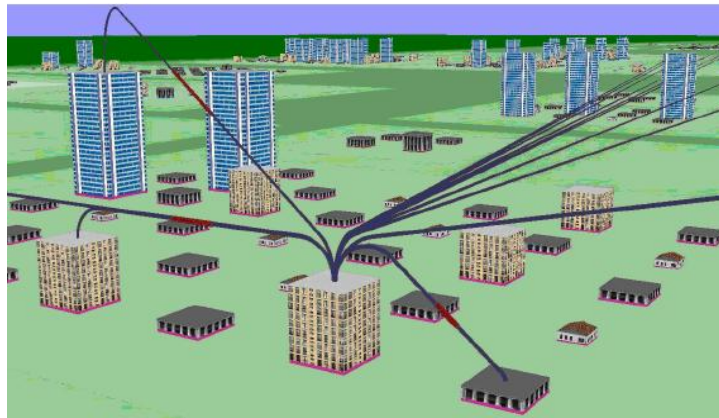


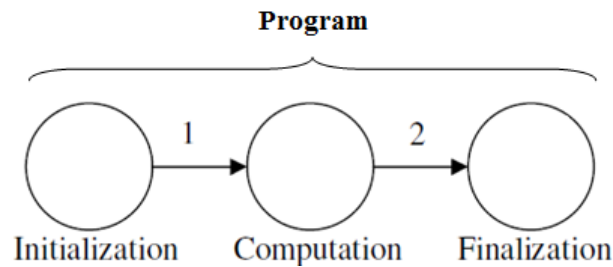
Figure 2.9. Using metaphors to represent interactions among the system components.

## Chapter 3 PHASE FLOW DIAGRAM

### 3.1 The Concept of Execution Phases

Pirzadeh et al. define an execution phase as “a segment of a program’s execution that performs a specific task” [PH11b]. Suppose that we have a program that allows drawing a car on the screen. In a simple scenario, a car can be composed of a rectangle as the body, and two circles that represent the tires. Executing such a scenario could generate a large trace. The question is where in the trace the drawing of each component occurs. It would be useful to software engineers to automatically identify these phases of the program from the trace. This is known as trace segmentation. The purpose is to divide a large trace into meaningful segments that represent specific computations. The level of granularity of each phase may vary.

At a high-level, any program execution can be seen as three main phases [PH11a]: The initialization phase, the computation phase, and the finalization phase (Figure 3.1). Each phase can also be divided into smaller segments, called sub-phases that are responsible to perform sub-tasks of the program. A tool that supports the handling of execution phases should allow enough flexibility to modify the level of granularity of a phase. Interesting results are obtained when the user is given the ability to zoom in and out a phase during the trace exploration process.



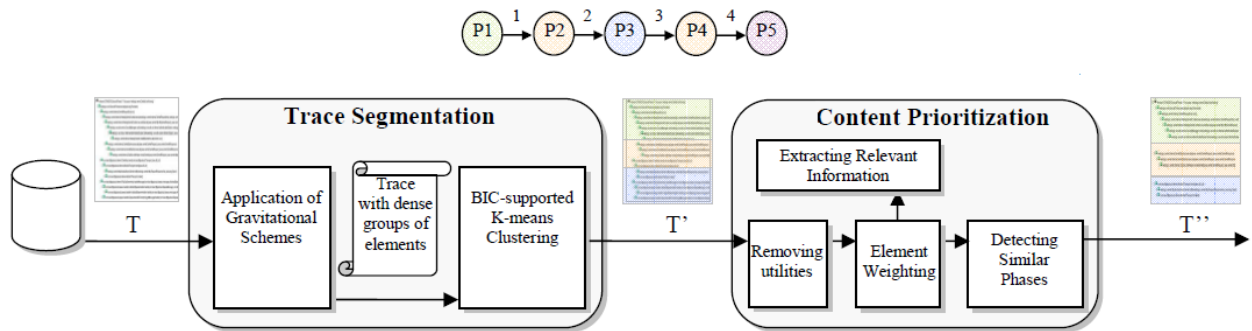
**Figure 3.1. Main phases of a program**

Automatic trace segmentation is not an easy task. The problem is that there is no construct at the programming level which could indicate the beginning and ending of each phase. This would have made tracing each phase easier.

There are some proposed techniques in the literature that attempt to automatically divide a trace into execution phases. Reiss et al. [Rei05, Rei07] propose to use profiling information such as, numbers of invocations from one class, thread execution time, as a heuristic for detecting the beginning and ending of each phase. The problem with their approach is that it uses extensively visualization to show changes in the program execution.

Watanabe [WII08] uses the number of objects that are created or destroyed as an indicator of phases. The assumption is that at the beginning of each phase many new objects are created to implement the tasks. Destruction of objects is an indication that the phase terminates. Their approach ignores control flow information and is not suitable for our research. A topic similar to phase detection, called concept extraction has been investigated by Asadi et al. [ADAG10]. Concepts are more fine-tuned elements than execution phases and are extracted based on heuristic search. Detecting concepts does not guarantee the detection of execution phases.

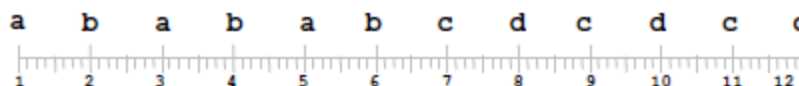
In this thesis, we adopt the phase detection algorithm developed by Pirzadeh et al. in [PH11a, PH11b, PHS11c], which, unlike existing algorithms, it was developed to apply to segmenting routine call traces. The authors proposed a phase detection framework that is based on the way the human perceive different scenes. The idea is to mimic the way the human perception system segments large scenes into objects and shapes. Their approach, called TSR (Trace Segmentation through Repositioning of trace elements) is depicted in Figure 3.2. It contains several components: Trace segmentation and content prioritization.



**Figure 3.2. Overview of TSR approach (from [PHS11c])**

The trace segmentation component that is composed of two sub-components (i.e., Application of Gravitational Schemes and BIC-supported K-means Clustering) is responsible for dividing the trace into execution phases. To achieve this, the authors proposed a number of techniques for measuring cohesiveness within groups of events. Their measures are inspired by Gestalt laws of similarity and continuity [Kof99, SF99, WKL+08, FRC10]. Gestalt laws are studied in cognitive psychology to explain how the human perception system groups lines and dots into shapes and objects.

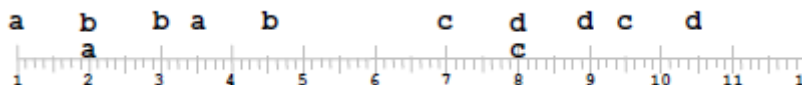
The TSR approach encompasses many steps. To help with the explanation, we use sample traces to depict how each step works. Figure 3.3 shows our first sample trace composed of different calls to methods a, b, c and d. As shown in this figure, the distance between each two consecutive calls is the same (i.e., one unit of distance on a ruler).



**Figure 3.3 Sample trace (from [PHS11c])**

The first step is to reposition the events of the trace such that cohesive methods are grouped together, which may indicate the presence of potential phases. To achieve this, Pirzadeh et al. proposed two repositioning methods called similarity and continuity gravity methods [PHS11c].

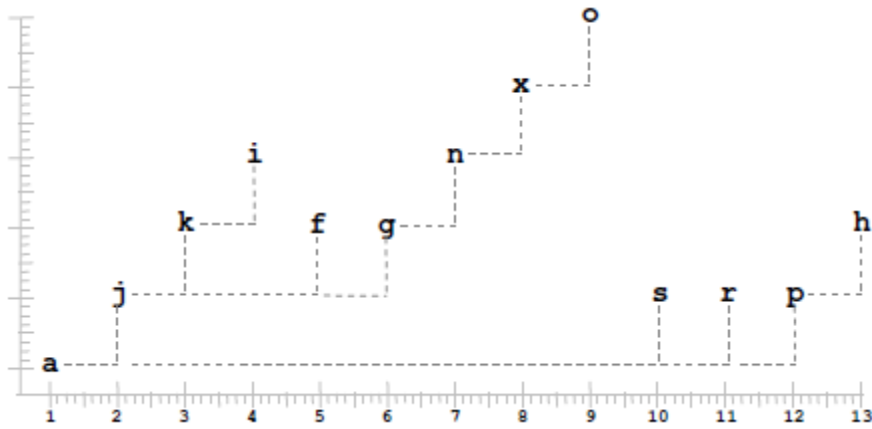
The similarity gravity method repositions the events of a trace by reducing the distance between similar events to form dense groups of events. Based on their definition, two events are similar if they are calls to a single method. Applying the similarity based on repositioning technique to the trace of Figure 3.3 results in two dense clusters as shown in Figure 3.4.



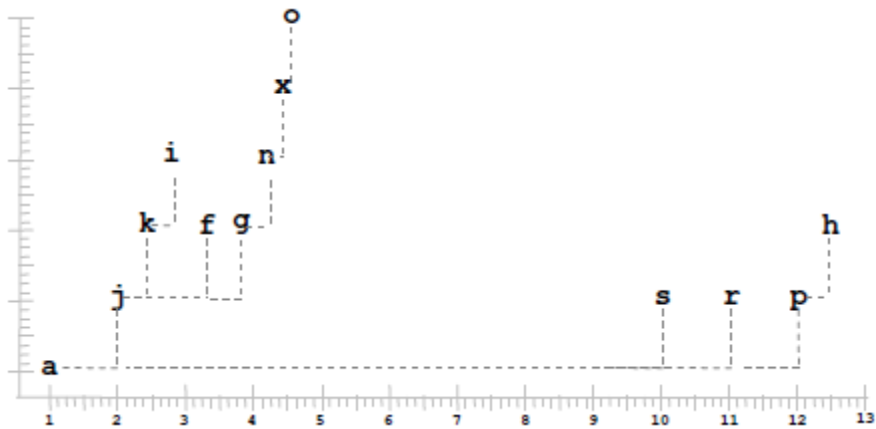
**Figure 3.4 Similarity Gravity method result (from [PHS11c])**

The continuity gravity method groups the events based on their nesting levels. For that, the gravity method reduces the distance between two nesting calls. In this method, the distance between two events is defined as the difference between their nesting levels. This method is inspired by the observation that as the nesting level of events in a trace

increases the probability of starting a new phase decreases. For the same reason, an important change in the nesting level of two successive events could indicate the beginning of a new phase. The authors have considered the same observation in their proposed continuity gravity method. Figure 3.5 shows a trace with nesting levels. The result of applying the continuity gravity method is shown in Figure 3.6. For more details on the repositioning techniques, the reader is invited to consult [PHS11c].



**Figure 3.5 Trace with nesting levels (from [PHS11c])**



**Figure 3.6 Continuity Gravity method result (from [PHS11c])**

Once the trace elements are repositioned, the authors used a K-means clustering with fine-tuning of parameters to automatically identify the beginning and the ending of each phase. Another interesting aspect of the TSR approach is that it embodies a threshold that can be changed to vary the level of granularity of the phases. This way, the users can browse the trace at various levels of abstraction. When applied to several systems, the TSR approach showed good results in extracting phases from large traces.

The Content Prioritization component of the TSR approach is used to find the most representative events of each phase. This is because many routines can be involved in implementing a phase despite the fact that only a few of them are important. The content prioritization component has four sub-components, Utility Removal, Element Weighting, Extract Relevant Information and Determining Similar Phases.

The Utility Removal sub-component removes the events such as call to library functions, etc. These events are considered as noise as shown by Pirzadeh et al. in [PHS11c].

The Element Weighting sub-component determines the weight of each event in each phase. The weight reflects the relevance of the event to the implementation of the phase. The authors argued that an event which repeated more frequently in a phase but less frequently in other phases is an indication of its relevance to that phase. When the process is done, it sorts the events of each phase based on their weights from high to low.

After the weighting process, the Extract Relevant Information sub-component picks up the first  $N$  events of each phase (which are already sorted by their weights) as representative events. A threshold is used to determine among the highly ranked events which ones are deemed to be the most representatives of the phase.

The final step of TSR is to determining similar phases. Phases are compared based on their representative events [PHS11c].

## 3.2 Building the Phase Flow Diagram

### 3.2.1 Guiding Principles

Based on analysis of the phase detection methods presented in Watanabe et al. [WII08] and Pirzadeh et al. [PHb], we identified a number of requirements that the phase flow diagram should support. We divide these requirements into the following categories: Ability to support various concepts, support of program comprehension tasks, and use of intuitive notations.

#### *A. Ability to support various concepts*

PFD should be expressive enough to represent the helpful information about execution phases. The diagram should be designed in such a way that the correct order of calls is preserved. Software engineers should clearly visualize the execution phases and the transition between them. Also, PFD should be powerful enough to handle the representation of multi-threaded and parallel executions. For example is that the PFD must be able to represent cases where a new thread is created during the execution (this could be shown as a fork off the main thread). There should be a way to allow software engineers to categorize execution phases based on the type of tasks that are performed in those phases. For example, a sequence of events that embodies calls to a database should be easily identified using a distinct notation.



### *B. Support of Program Comprehension Tasks*

There exist several program comprehension models that characterises how a programmer understands a program as shown in the background chapter. Similarly, we want to have a mechanism that supports the exploration of a PFD by varying the level of abstraction of the phases. The PFD should support notations that show phases and their sub-phases. The phase detection we adopted in this thesis uses a threshold that governs the level of granularity of the phases.

In addition to varying the level of details automatically (i.e., by controlling the threshold), we also support the ability for users to merge or divide phases manually. This is important since the automatic approach might miss some important phases that can be used as landmarks for the exploration of the trace. In other cases, the user might end up with phases that are too fine-grained and in which case he or she needs to merge them into higher phases.

Finally, the tool that implements PFD should allow effective mapping between the phases and the original trace. This way, the user is offered with an extra level of abstraction, which is, in this case, the trace itself if more details regarding specific aspects of the trace are needed.

### *C. Use of Intuitive Notation*

The introduction of a new diagram instead of reusing an existing one is always a risky choice. Users might not be willing to invest in learning a new notation. To mitigate this risk, it is important to use simple and intuitive notations. We will show in the next sections that the notation of PFD resembles to a great extent BPMN [BPMN] and UML

activity diagram [UML]. We carefully reused the same notations wherever we felt that there a close relationship between the model element in PFD and the one in BPMN or UML activity diagram.

### 3.2.2 Investigating Existing Diagrams

We started our investigation by identifying possible candidates that could be used for representing execution phases according the guiding principles discussed in the previous section. Our first choice was to use UML activity diagram as the most promising candidate in terms of fulfilling the requirements of PFD. We found some common concepts between activity diagram and PFD requirements such as transition, join and fork. However, the activity diagram does not support some of our requirements:

- Show the order of phases: The activity diagram control flow does not support the numbering of transitions or activities.
- Show a repeated phase once and refer to it multiple times: The activity diagram does not support the self reference or self loop.
- Differentiate between start of main flow and sub flows: Each activity diagram can have just one initial notation.
- Show when a sub-flow starts in comparison with the main flow. There is no notation in Activity diagram to meet this requirement.
- Show multi-threaded executions: Activity diagram does not have any notation that could be mapped to concept of threads.
- Show phase types and operation identifiers.

Given that the activity diagram was the most suitable candidate among UML diagrams, based on the above-mentioned results we decided to extend the activity diagram instead of creating a new diagram from scratch.

### 3.2.3 PFD Metamodel and Notation

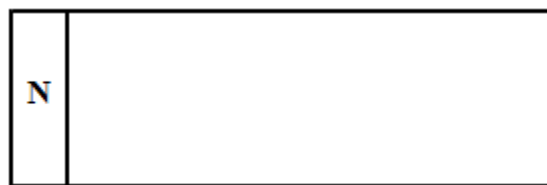
The metamodel of PFD is represented in Figure 3.9. We will discuss each model element in more detail in the next subsections following an informal template used by OMG to describe standardize modeling language such as UML (see [UML] for an example).

#### 3.2.3.1. *Thread*

Description:

The Thread class represents an execution thread of the traced scenario. A trace file can be composed of one or multiple threads. We represent the flow of phases in each thread individually. The interaction between threads is modeled using addition constructs.

*Notation:*

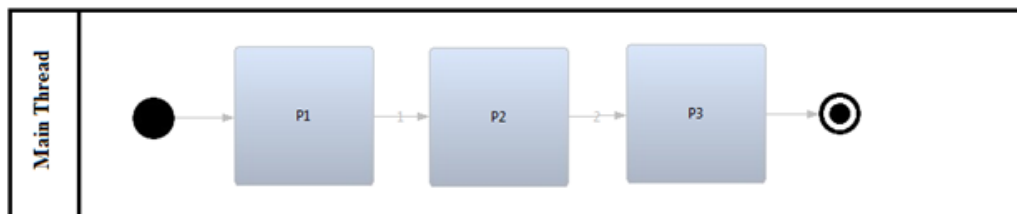


Constraint(s):

- Each Thread notation must include maximum one phase flow diagram.
- Each Thread notation must have a name.

Example:

The following example shows the flow of phases which are extracted from the main thread.



**Figure 3.7. Thread notation sample**

### 3.2.3.2 Main Flow Start

Description:

The Start notation indicates the beginning of the main phase flow diagram (the main phase flow diagram is a diagram which shows the flow of phases in the main thread).

Notation:



Constraints:

- There must be one main flow start notation in the main thread.
- It has no incoming flow.
- It must have one outgoing flow.

Example:

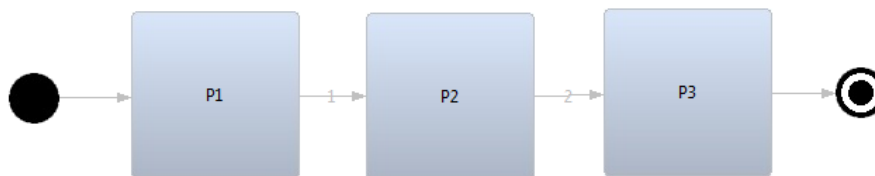


Figure 3.8. Main flow start notation sample

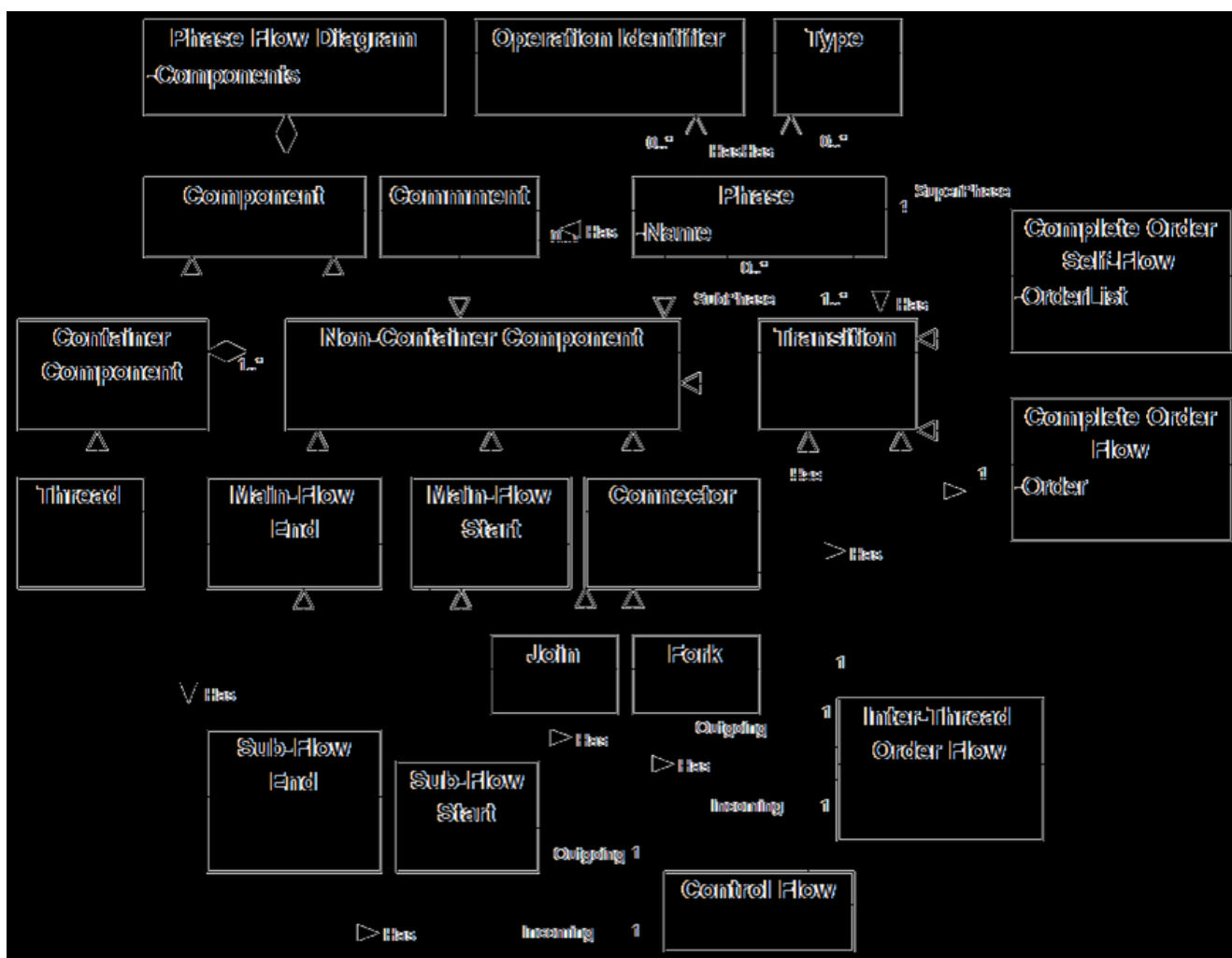


Figure 3.9. Meta model of Phase Flow Diagram

### 3.2.3.3. Main Flow End

Description:

The End flow notation indicates the end of the main phase flow diagram.

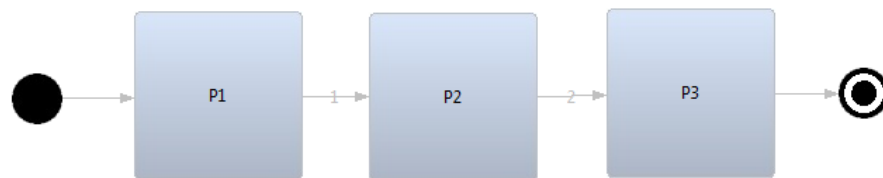
Notation:



Constraints:

- There must be one main flow end notation in the main thread.
- It has no outgoing flow.
- It must have one incoming flow.

Sample:



**Figure 3.10. Main flow end notation sample**

### 3.2.3.4. Sub-Flow Start

Description:

The Sub-Flow Start notation indicates the beginning of a phase flow diagram which shows the flow of phases in a sub-thread<sup>1</sup> of the main thread. A trace file can be

---

<sup>1</sup> A sub-thread is thread which starts from the main thread.

composed of one thread or more. This notation is used to represent the start of the phase flow diagram, extracted from a sub-thread of a main thread. It can be employed to distinguish the diagram which represents the flow of phases in the main thread and the diagram that depicts the flow of phases of a sub-thread especially when combined with the Thread notation.

Notation:

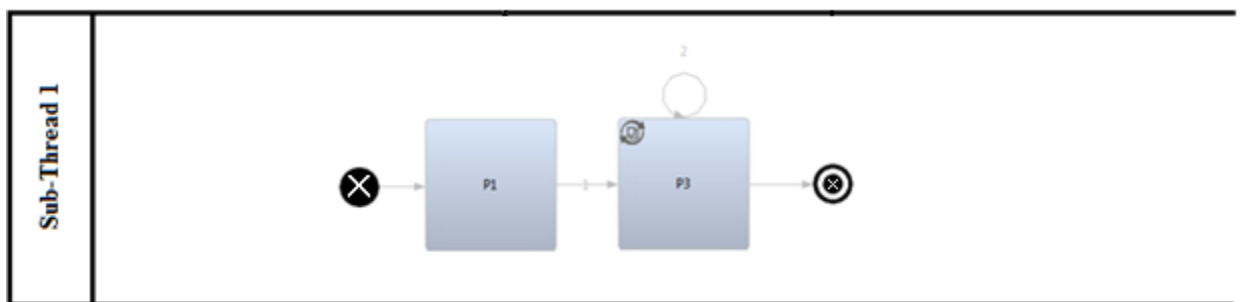
A Sub-Flow Start notation is black color filed circle which surrounds a white cross sign.



Constraints:

- There must be maximum one sub-flow start notation in a sub-thread.
- It has one incoming flow.
- It must have one outgoing flow.

Sample:



**Figure 3.11. Sub-Flow start notation sample**

### 3.2.3.5. Sub-Flow End

Description:

The Sub-Flow End notation indicates the end of a phase flow diagram which shows the flow of phases in a sub-thread of the main thread.

Notation:

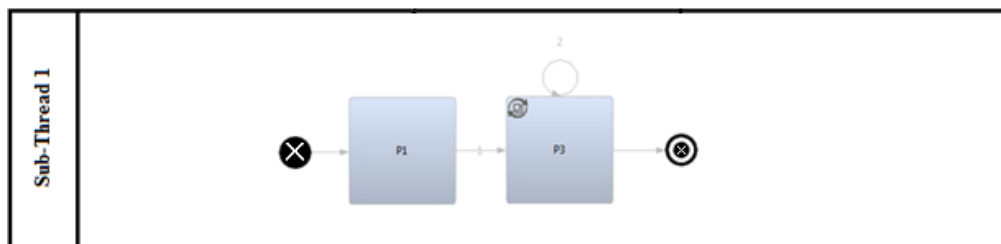
An End notation is a circle which must be drawn with a single thin line and surrounds another black color filed circle.



Constraints:

- There must be maximum one in a sub-thread.
- It has one incoming flow.
- It has one outgoing flow.

Sample:



**Figure 3.12. Sub-Flow end notation sample**



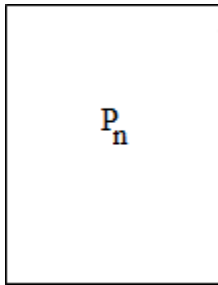
### 3.2.3.6. Phase

#### Description:

The Phase notation indicates a specific phase of the phase flow diagram.

#### Notation:

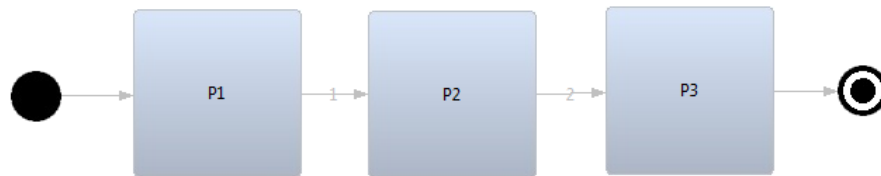
A Phase notation is a rectangle which must be drawn with a single thin line and includes the name of the phase.



#### Constraints:

- Must have at least one incoming flow if it is the first phase from flow start notation.
- Must have at least one outgoing flow if it is the last phase to the flow end notation.
- Must have maximum one outgoing Complete Order Flow if it is not the last phase.
- Must have maximum one incoming Complete Order Flow if it is not the first phase.

Sample:



**Figure 3.13. Phase notation sample**

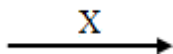
### 3.2.3.7. Ordered Flow

Description:

The Ordered Flow indicates the direction and order of the phase flow on the diagram. An Ordered Flow is an edge that represents the direction and the order of the flow by a number and starts a phase, after the previous one is finished in terms of time and if there is at least one event (method) in the previous phase that calls one or more events in the current phase. This flow is used between the phases of the same thread.

Notation:

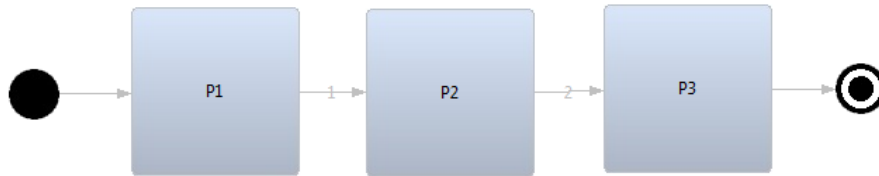
An Ordered Flow is a directional arrow which must be drawn with a single thin line and comes with a unique number which shows the physical order of transitions.



Constraints:

- It must have one source phase.
- It must have one destination phase.
- Must have a maximum of one order number.

Example:



**Figure 3.14. Ordered Flow sample**

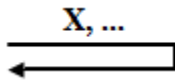
### 3.2.3.8. Ordered Self-Flow (OSF)

Description:

The Ordered Self-Flow indicates the direction and order of the phase flow on the diagram. This notation is used to represent the order of flow between phases that are deemed to be similar. An Ordered Self-Flow is an edge that represents the order of the flow by a number and starts the same phase, after the phase is finished.

Notation:

An OSF is a directional arrow which must be drawn with a single thin line and comes with one or more unique number(s) which shows the physical order of transitions.

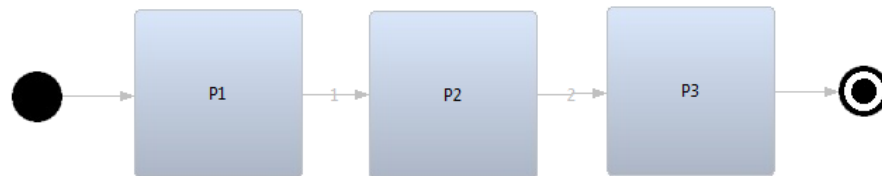


Constraint:

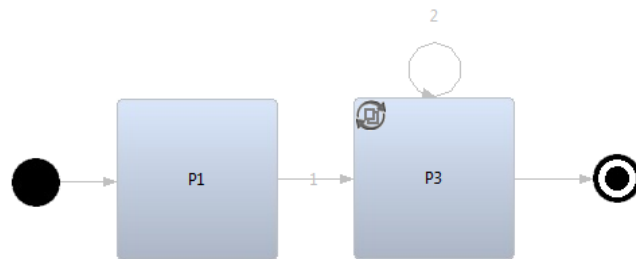
- It must have the same source and destination which is a phase.
- It must have at least minimum one order number.

Sample:

In the flowing example the first figure (Figure 3.15) represents the original flow of phases and the second figure (Figure 3.16) is another representation of the first figure in which P2 and P3 are considered similar.



**Figure 3.15. Original flow of phases**



**Figure 3.16. Ordered Self-Flow notation sample**

### 3.2.3.9. Fork

Description:

It is used to divide an execution flow into multiple flows that run in parallel in different threads.

Notation:

A Fork notation is a horizontal cut tip triangle which surrounds a plus sign and it must be drawn with a single thick line.

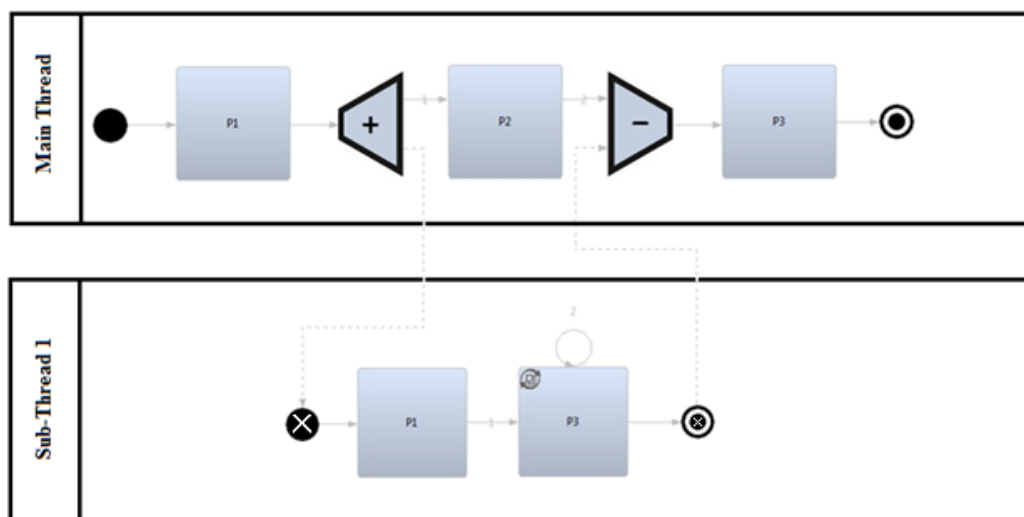


Constraints:

- It must have one incoming flow.
- It must have at least two outgoing flows.
- The parallel executions must be represented in different threads.

Example:

In the following example the outgoing flow from P1 is divided to two flows.



**Figure 3.17. Fork notation sample**

### 3.2.3.10. Join

#### Description:

This is the opposite of the Fork operation. The join point indicates where two or more flows of executions are joined together. This is needed to join execution flows coming from parallel executions.

#### Notation:

A Join notation is a horizontal cut tip triangle which surrounds a minus sign and it must be drawn with a single thick line.

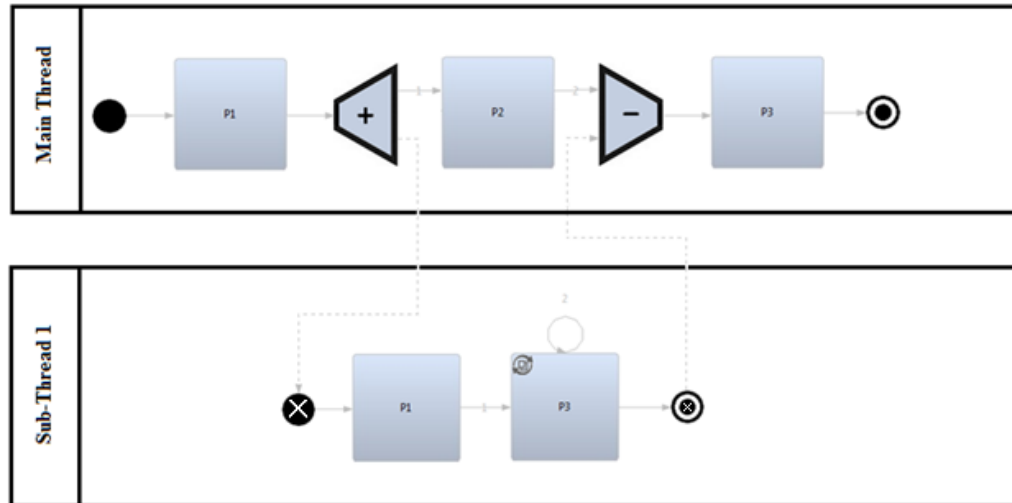


#### Constraints:

- It must have at least two incoming flows.
- It must have one outgoing flow.
- The phases that are joined must be in different threads.

#### Example:

In the following example the outgoing flow from P2 in the Main thread and the coming flow from Sub-Thread1 are joined together.



**Figure 3.18. Join notation sample**

### 3.2.3.11. Inter-Thread Order Flow (ITOF)

Description:

The Inter-Thread Order Flow is used to depict the flow between phases from different threads.

Notation:

Each ITOF must have one source and one destination. An ITOF is a directional arrow which must be drawn with a single dash thin line.



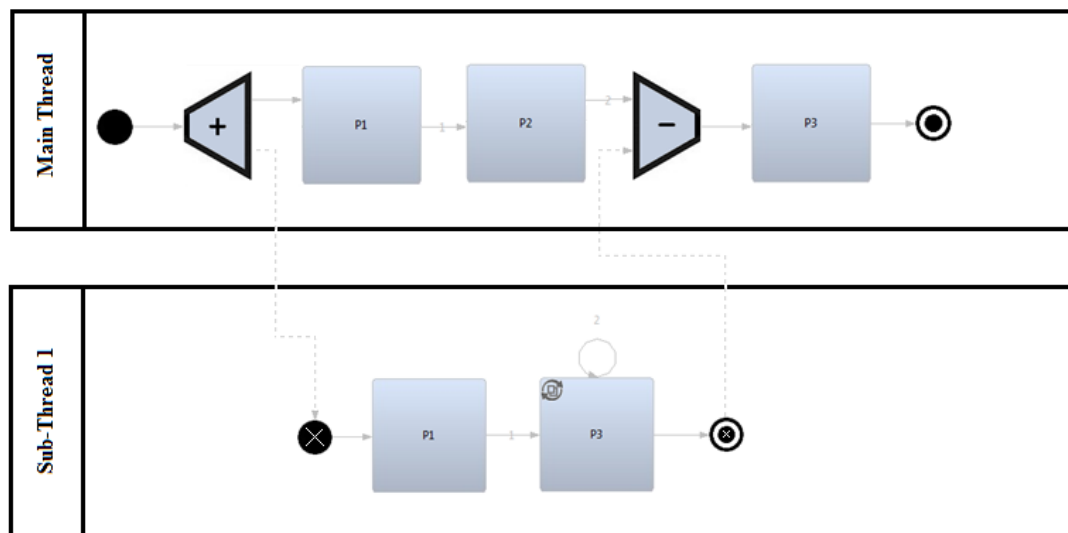
Constraints:

- The source must be a Fork notation if the original flow is originated from the main thread.
- The source must be a Sub-Flow end if the flow merging into the main thread.

- The destination must be a sub-flow start if the original flow is originated from the main thread.
- The destination must be a Join notation if the flow merging into the main thread.

Example 1:

In the following example, phase P1 in Sub-Thread1 has started before phase P1 in the Main thread ends and P3 in Sub-Thread1 ends after P2 and before P3 in the Main thread.



**Figure 3.19. Inter-Thread Order Flow sample 1**

Example 2:

In the following example the P1 phase in the Sub-Thread1 is started after P1 phase in the Main thread and P3 phase in Sub-Thread1 ends before P3 and after P2 phase in the Main thread.



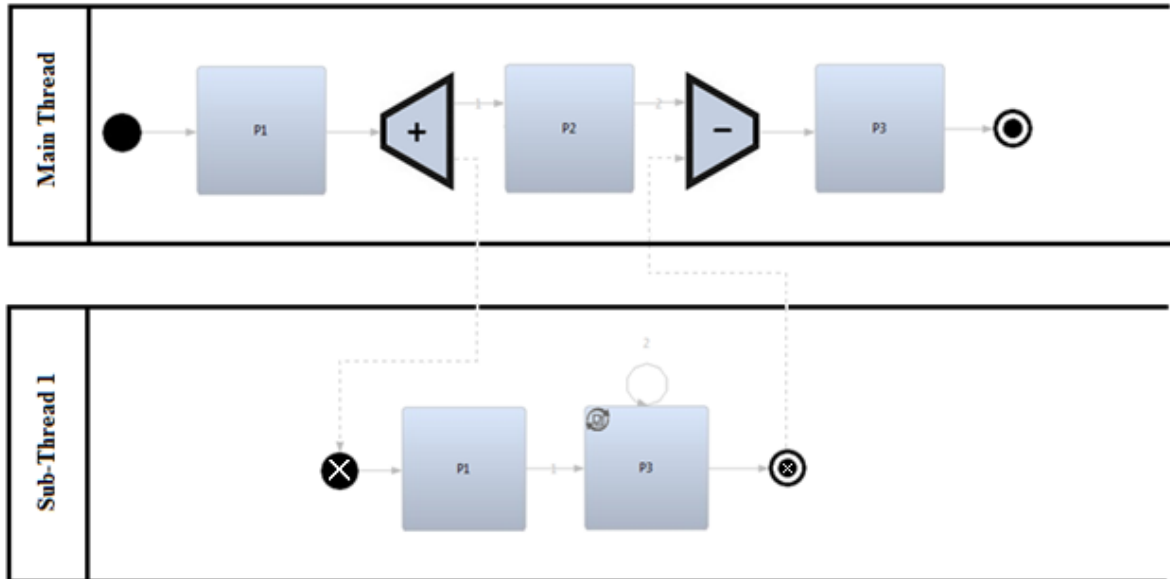


Figure 3.20. Inter-Thread Order Flow sample 2

Example 3:

In the following example, P1 in Sub-Thread1 has started before P2 in the Main thread and P3 in Sub-Thread1 ends after P3 in the Main thread.

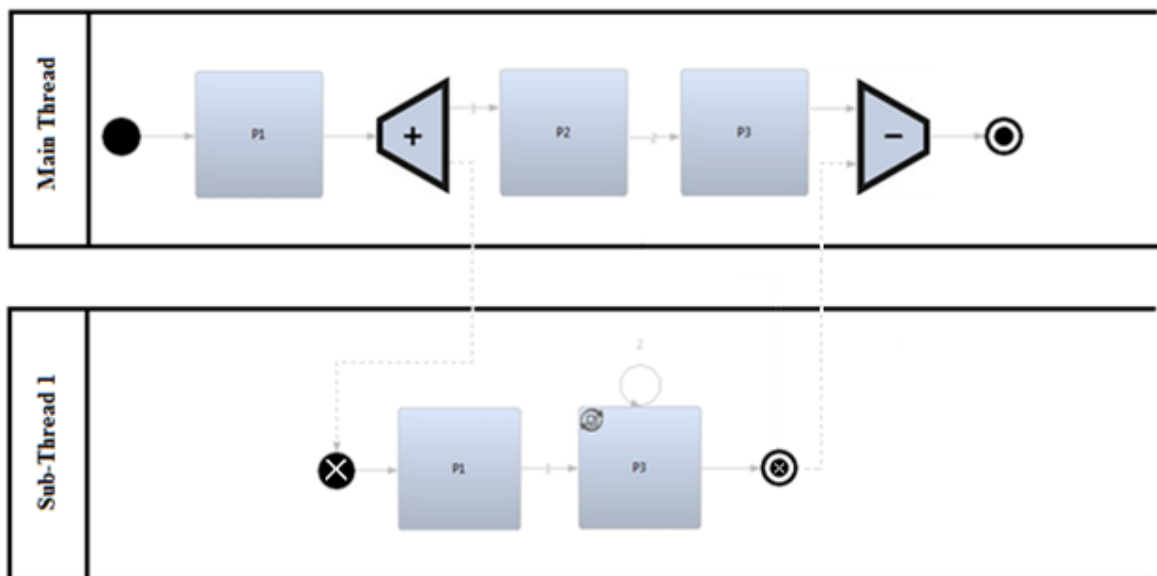


Figure 3.21. Inter-Thread Order Flow sample 3

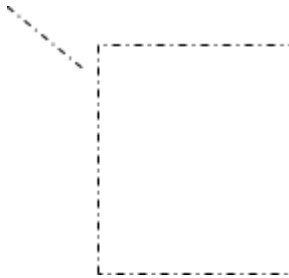
### 3.2.3.12. Comment

#### Description:

The Comment notation is used to write comments on a specific part of a phase flow diagram. Software analysts can use this notation to provide additional information about a phase or a set of phases. The comment notation can also be used to enter authorship information, version dates, etc.

#### Notation:

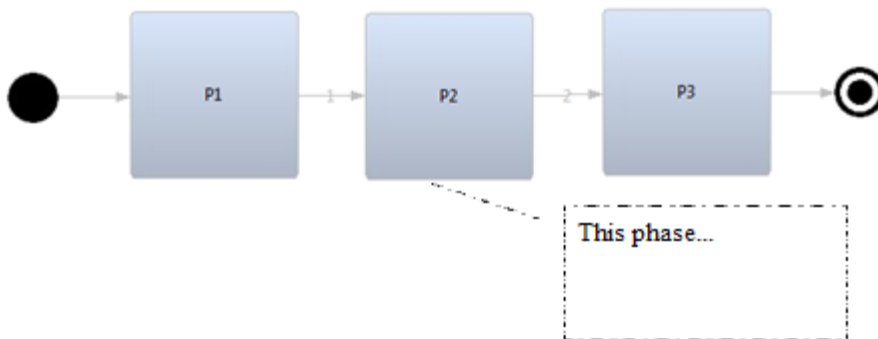
It is a rectangle which must be drawn with single thin dash lines and a single diagonal dash line which must be connected to a specific notation.



#### Constraints:

- It must be connected to a specific notation on the diagram.

Example:










**Figure 3.22. Comment notation sample**

### 3.2.3.13. Phase Types

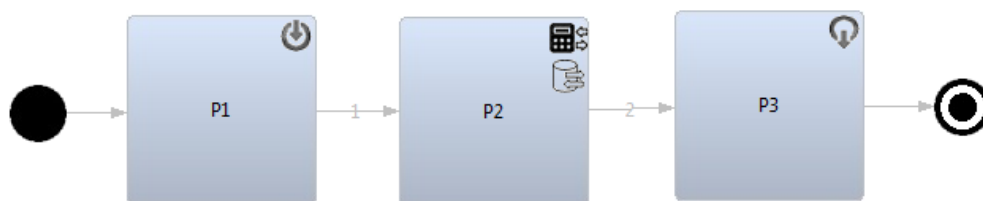
Through the analysis of several execution phases, extracted from different system traces, we have noticed that many of these phases are dedicated to specific computations such as accessing a database, performing networking operations, etc. We therefore decided to add a phase type to capture the essence of a phase. The types we suggest in this thesis are described in Table 3.1. The tool that supports this notation should allow software engineers to add other types as needed.

**Table 3.1. Phase types**

	It shows that the phase contains functions that manipulate information found in a database.
	It shows that the phase contains functions that modify the information displayed on a GUI.
	It shows that the phase contains functions that operate on external devices.

	It shows that the phase contains functions writing to or reading from hard drive.
	It shows that the phase contains functions using network operations.
	It is used to show an initialization phase.
	It is used to show a finalization phase.

From the notational point of view, each type is represented by an icon, which must be placed on the right side of the phase node. In Figure 3.23, phase P1 is typed as ‘initialization’ phase because it contains trace events that perform typical initialization operations such as creating the main user interface of a program. The P2 phase has two types External Device and Database types, meaning that this phase is responsible for retrieving some information from an external device and saving them into the database. Note that, as shown in this example, a phase can have multiple types. Phase P3 is typed as a ‘finalization’ phase, meaning that it implements the termination steps of a program.



**Figure 3.23. Phase types sample**

### 3.2.4 Phase Operations

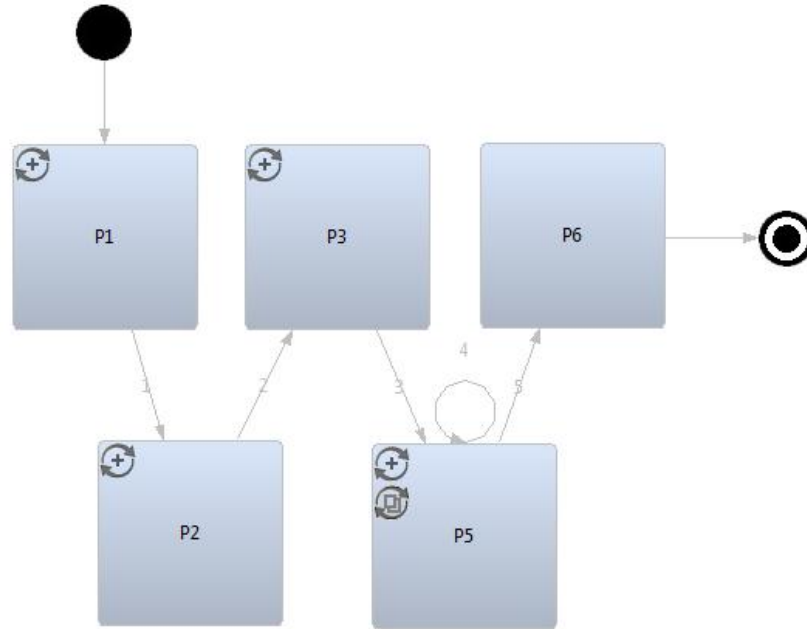
Software analysts might wish to have the flexibility to manually group or hide execution phases. For this, PFD supports three operations: Merging, Hiding and Unification.

#### **Merging:**

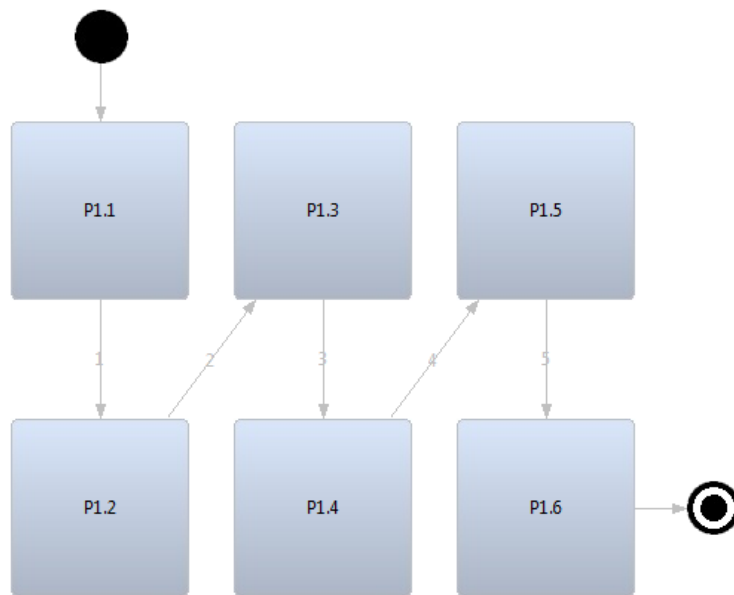
Merge is applied when two or more successive phases perform a single task or the software maintainer aims to understand a more general task without having to go through the sub-tasks of the general task. A merged phase is called “Super Phase” and contains at least two sub-phases. It should be noted that the phase detection technique implemented in this research allows to change the level of granularity of the phases automatically by changing the value of a threshold. However, this technique impacts the whole trace. The Merge operation is used in cases the software analysts wished to merge specific phases without having to change the level of granularity of the whole phase flow. In Figure 3.24, P1 is a phase that resulted from merging six sub phases, shown in Figure 3.25. The icon on the left corner of P1 denotes an automatically merged phase.

#### **Hiding:**

A phase can be hidden from the diagram to reduce the complexity of the model. A software maintainer can use this operation to remove from the display phases that are not needed for the analysis at task, hence reducing the amount of information displayed.



**Figure 3.24. Flow of phase with super phases**



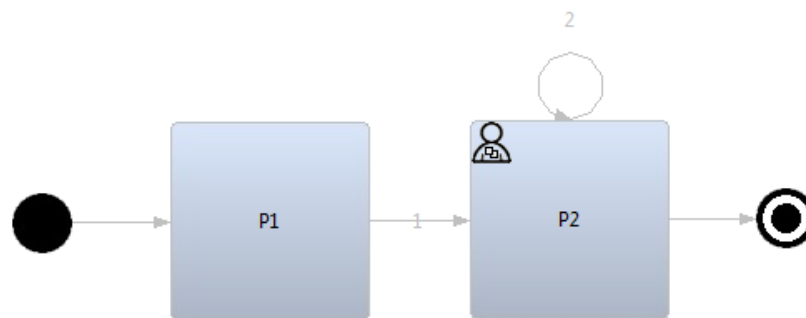
**Figure 3.25. Sub-phases of P1**

**Unification:**

Unification is applied when two or more phases are similar. As mentioned before, each phase is responsible for a specific task. When two phases perform the same task, it means that they are similar. The similarity between the phases can be identified automatically by looking at the number of similar routines in each phase [PH11a], or manually by the user. Figure 3.26 shows a flow of phases and Figure 3.27 shows the same flow in which P2 and P3 are manually unified by the user.







**Figure 3.26. Flow of phases**



**Figure 3.27. Flow of phases with a manually unified phase**

Table 3.2 shows the icons that we chose to distinguish between merging and unification.

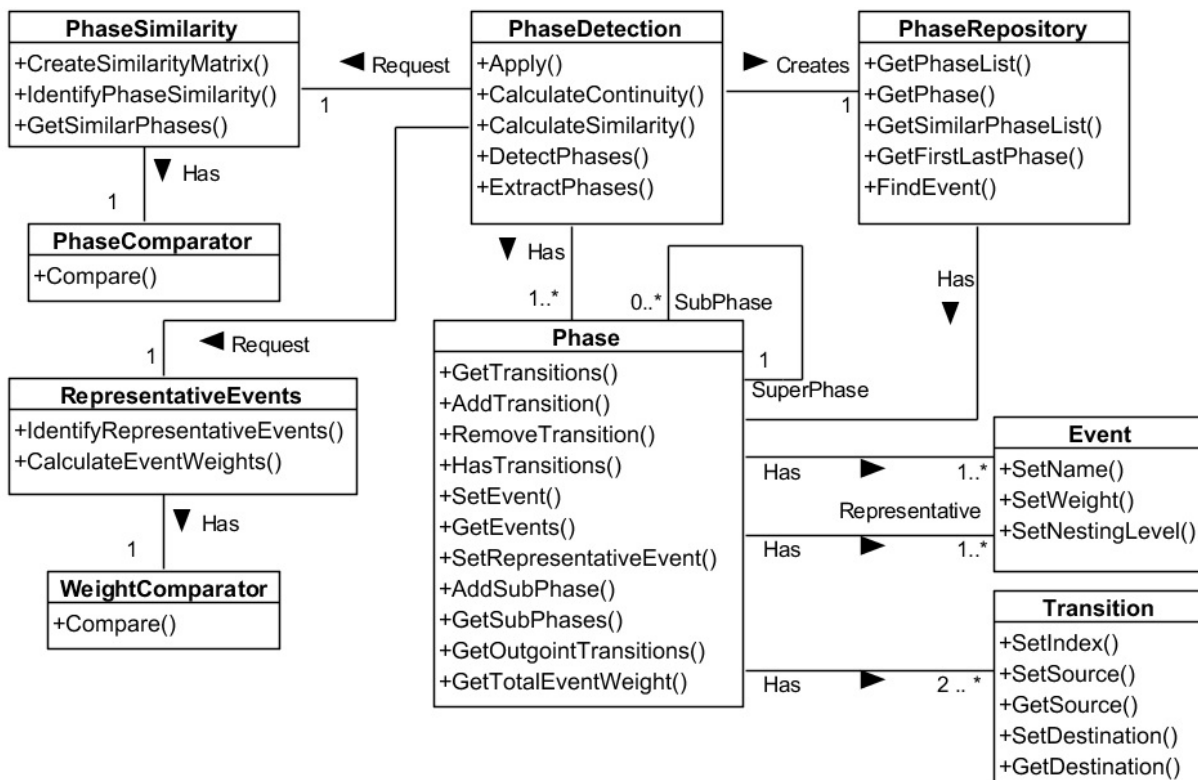
**Table 3.2. Phase operations**

	<p>It shows an automatically detected super phase (Merged phase).</p>
	<p>It shows that the phase is automatically unified.</p>
	<p>It shows a manually created super phase (manually merged phase).</p>
	<p>It shows that the phase is manually unified.</p>

### 3.3 Tool Support

We implemented the TSR's components (Trace Segmentation and Content Prioritization) including the gravitational schemes, BIC-supported K-means clustering, the extraction of relevant information, the removal of utilities, and the detection of similar phases. The following class diagram (Figure 3.28) shows all the involved classes in the implementation of the approach and some important functions of each class.





**Figure 3.28 TSR Class Diagram**

In the class diagram of Figure 3.28, the *PhaseDetection* class is responsible of finding execution phases and extracting them from the input trace file. In this class, the *Apply()* function is the starting point. It reads the trace file using Java StAX API. The *DetectePhases()* function finds the start and the end of each phase using the *CalcualteContinuity()* and *CalculateSimilarity()* functions. These two functions implement the similarity and continuity gravity methods discussed in Chapter 3. Finally, the *ExtractPhases()* function evokes the execution phases from the results returned by the *DetectPhases()* function. One of the sub-components of the Trace Segmentation component is BIC-supported K-means Clustering determines the quality of execution phases. To implement this sub-component we used an external Java library called Java-ML[JAM].

The *RepresentativeEvent* class, depicted in this class diagram, is used to find the most important events of each detected phase called representative events and belongs to Content Prioritization component. There are two important functions in this class; *IdentifyRepresentativeEvents()* and *CalcualteEventWeights()*. The *CalcualteEventWeights()* determines the weight of each event and then the *IdentifyRepresentativeEvents()* function, using these weights, determines the most representative events from the list of events of each phase.

The *PhaseSimilarity* class is used to calculate the similarity between the detected execution phases. To do so, the weight of representative events of each phase is used to specify the similarity between them. The *IdentifyPhaseSimilarity()* function in this class determines the similarity between two phases. The *CreateSimilarityMatrix()* function builds a two dimensional array called *SimilarityMatrix* which shows the similarity between phases based on the *IdentifyPhaseSimilarity()* returned values. The *GetSimilarPhases()* function returns all the similar phases to a specific phase.

The *Phase* class represents a single detected execution phase and each phase can be composed of Sub-Phases. The *Event* class represents an event of a phase and *Transition* class indicates incoming transition to or outgoing transition from a specific phase. Each class has some functions to set or get required information.

The *PhaseRepository* class is a store for the detected execution phases. This class has some functions to perform operations on the list of detected phase. For example, the *FindEvent()* function is used to search for a specific event through all the detected phases.

The *GetFirstLastPhase()* function returns the first and last phases of the execution flow and *GetPhase()* function returns a specific phase.

To Implement the Phase Flow Diagram (PFD) we extended some of the classes of a pre-implemented library called Zest to create PFD's notations. Zest contains a set of visualization components built for Eclipse. The subsequent class diagram (Figure 3.29) represents all the involved classes in implementation of the PFD including some of their important functions.

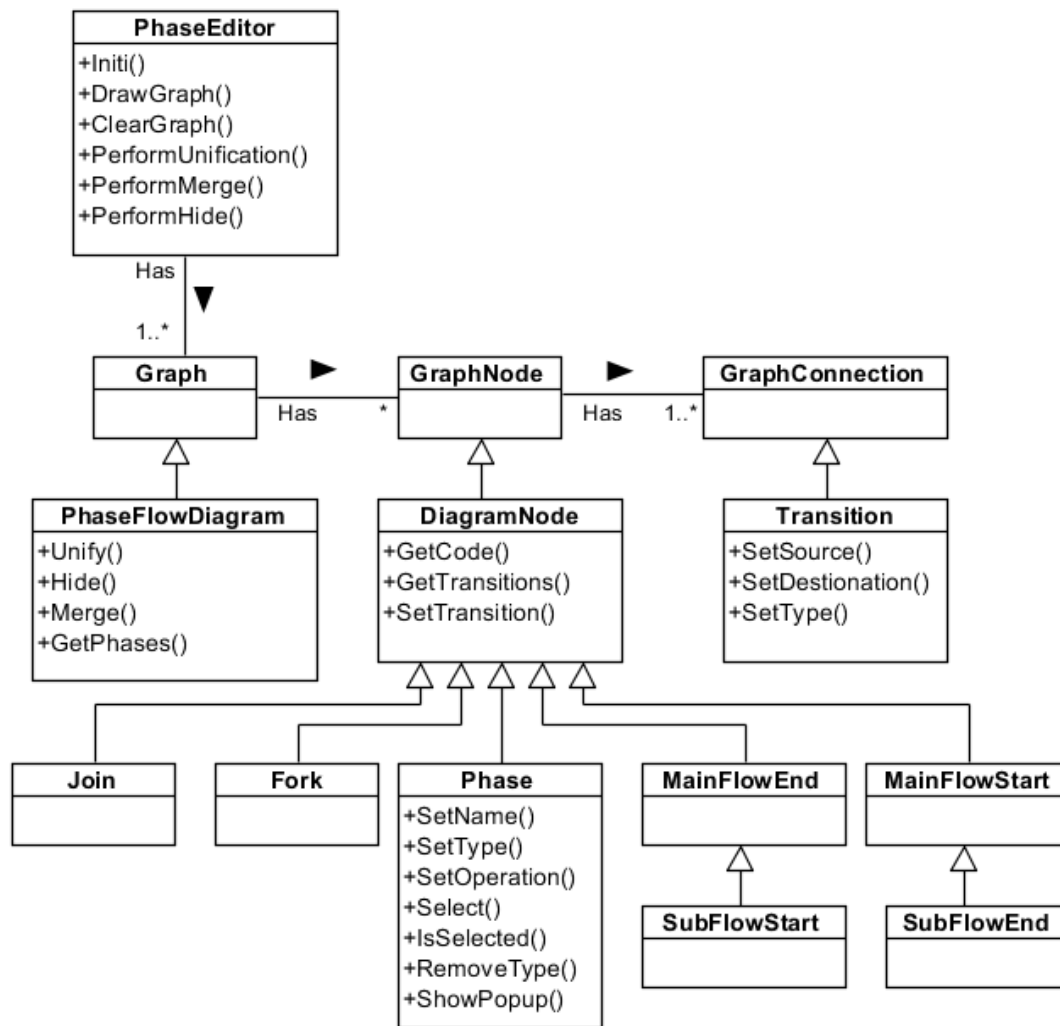


Figure 3.29 PFD Class Diagram

In the diagram of Figure 3.29, the *PhaseEditor* class receives an instance(s) of *PhaseRepository* class as its input and draws the diagram. The *Init()* function sets some global properties. The *DrawGraph()* function draws the Phase Flow Diagram(s) based on the input value(s) on the phase editor. There are also three other important functions, *PerformUnification()*, *PerformMerge()* and *PerformHide()*. Each of these functions calls the equivalent function in the *PhaseFlowDiagram* class to perform a specific operation on a specific PFD.

The *PhaseFlowDiagram* class represents a single PFD which is inherited from Zest's *Graph* class. The *Hide()* function in this class hides a specific phase from the diagram. To perform this operation the status of the desired phase is set to "Hidden" and then it is removed from the diagram. The other important function in this class is *Unify()* which is used in manually unification process. This function receives a list of phases that are deemed to be similar and asks the *PhaseRepository* class to change the similarity matrix and then applies the new changes on the diagram. Also, the *Merge()* function is used in manually merging process. As mentioned before, the merge operation has to be done on the successive phases.

The *DiagramNode* class is the base class for all the PFD's notations except transitions and is inherited from Zest's *GraphNode* class. Each *DiagramNode* has a unique code which is generated automatically and it is accessible by *GetCod()* function. The *GetTransitions()* function returns all the incoming and outgoing transitions of a diagram's notation. The *SetTranastion()* function attaches an incoming or outgoing transition to the notation.

The *Transition* class represents a single transition on the diagram and is inherited from Zest's *GraphConnection* class. It has several functions to set and get the transition's source and destination, set the transition's type and etc.

The *Phase* class illustrates a phase on the diagram. It consists of all the required operations needed to be performed on a phase. In this class, the *SetName()* function is used to assign a name or change the current name of the phase. To assign or remove one of the phase types discussed in previous chapter the *SetType()* and *RemoveType()* functions were implemented and the *SetOperation()* function is used to assign one of the phase operations also discussed in the previous chapter to the phase. When user right clicks on a phase a popup menu appears. Via that menu user is able to assign one or more than one type to the phase, go into the phase to see the sub-phases if the clicked phase is a super phase or go back to the parent phase flow diagram if the clicked phase is a sub-phase. To do so, the *ShowPopup()* function was implemented.

There are also some other classes in the diagram such as Join, Fork, MainFlowEnd and etc to handle all the required operations for a specific notation. The current version of the tool represents the flow of phases in a single thread, so the notations related to representing threads and the relationship between them such as join, fork and Inter-Thread Order Flow have not been implemented yet.

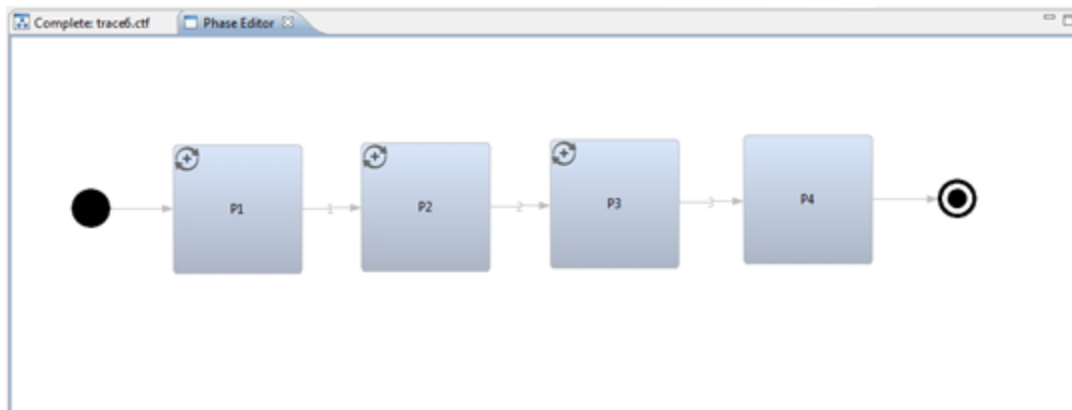
We implemented the PFD and TSR approach as an extension to SEAT [HLF05], a trace exploration and analysis tool developed by Fu et al. [HLF05, HFL04]. The discussed implementation results in new editors and views added to SEAT to support the phase flow diagram. Figure 3.30 shows the content of a trace view, which displays the trace in a

tree widget. The figure shows also the application of the phase detection algorithm on this viewed trace. In this figure, each color represents a single phase and each number at the beginning of the each event name depicts the sub-phase to which the event belongs.



**Figure 3.30. Phases on Complete Tree Editor**

We implemented a phase flow diagram editor to allow software engineers manipulate phases in a flexible manner. An example of this view is shown in Figure 3.31.



**Figure 3.31. Phase flow diagram editor**

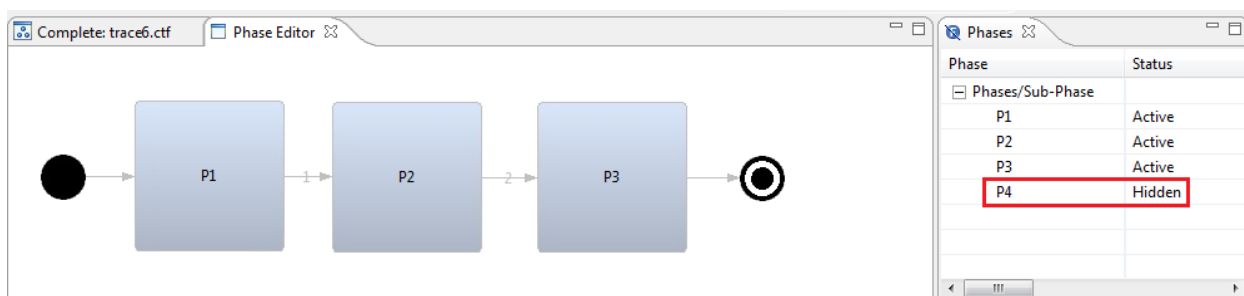
The trace and phase flow diagram editors are synchronized to allow software engineers to go back and forth. For example, selecting one phase in the phase flow diagram editor triggers the selection of the corresponding part of the trace in the trace editor. Figure 3.32 shows this synchronization. In this Figure P3 phase is selected in both editors.



**Figure 3.32. Synchronized Editors; Left: Phase Editor, Right: Complete Tree Editor**

In addition to editors, we also implemented new views in SEAT. The phase view shows the flow of phases in a table (see Figure 3.33). It has two columns, phases and their sub-phases are represented on a tree in the first column and in the second column, the last

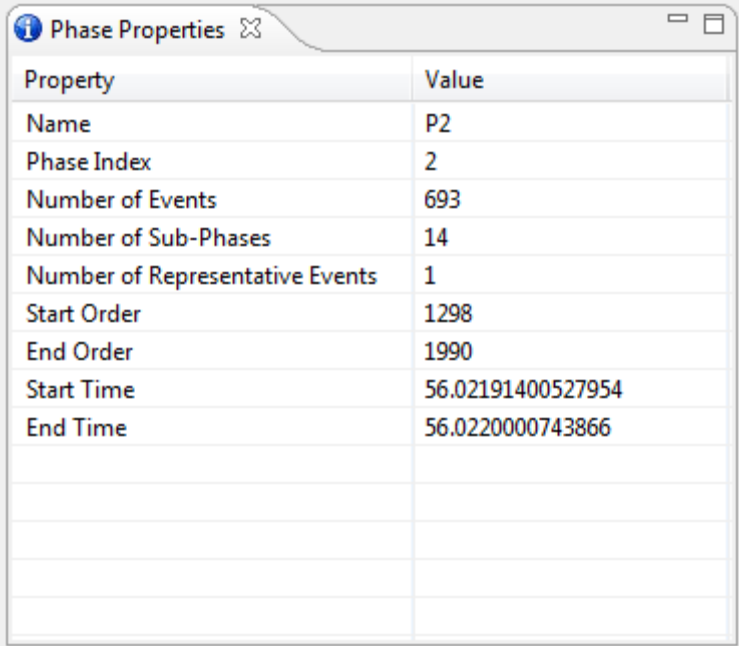
state of each phase or sub-phase is shown. The first column is never changed and always. In other words, making any changes to the diagram such as hiding a specific phase or merging two phases does not affect the represented phases in this column. The idea is to allow software engineers to see how many phases there are and what their state on the diagram is. Figure 3.33 shows the phase editor (left side) and the phase view (right side). The phase view shows that the P4 phase exists despite the fact that it is hidden in the phase editor.



**Figure 3.33. Phase Editor and Phase View**

Another view we have developed is the phase property view. This view displays information about the selected phase on the phase flow diagram including the name (the name of the phase is assigned by a user and can be edited), the phase index (order of the phase), the number of events (number of events that the selected phase contains), number of sub-phases, number of representative events, the order of first event of the selected phase on the trace, the order of last event of the selected phase on the trace, the start time (timestamp of the first event of the phase), and the end time (the timestamp of the last event of the phase). Figure 3.34 shows an example of the phase property view.



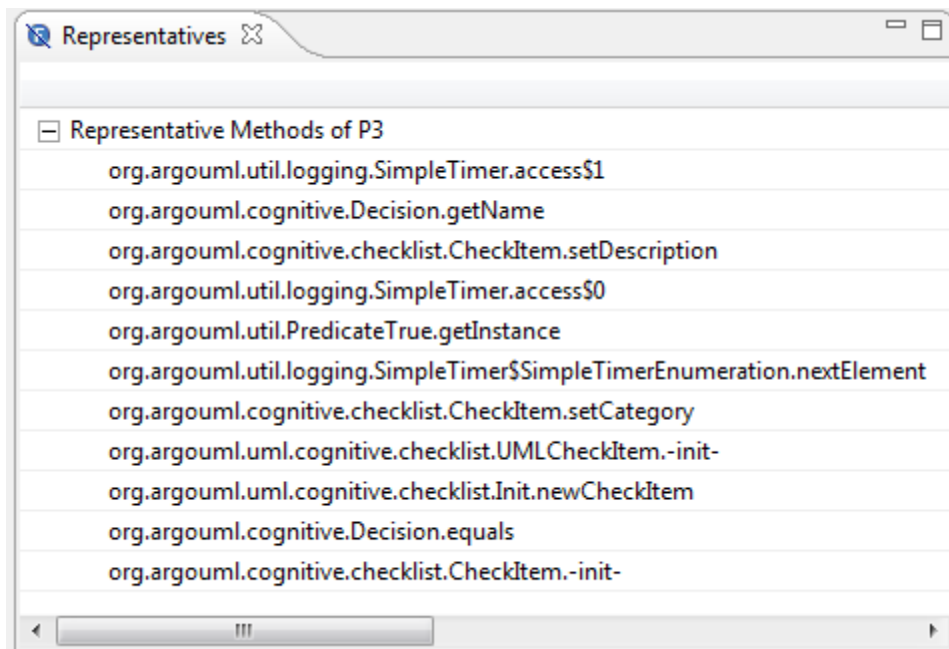


The image shows a window titled "Phase Properties" with a close button. It contains a table with two columns: "Property" and "Value". The table lists the following data:

Property	Value
Name	P2
Phase Index	2
Number of Events	693
Number of Sub-Phases	14
Number of Representative Events	1
Start Order	1298
End Order	1990
Start Time	56.02191400527954
End Time	56.0220000743866

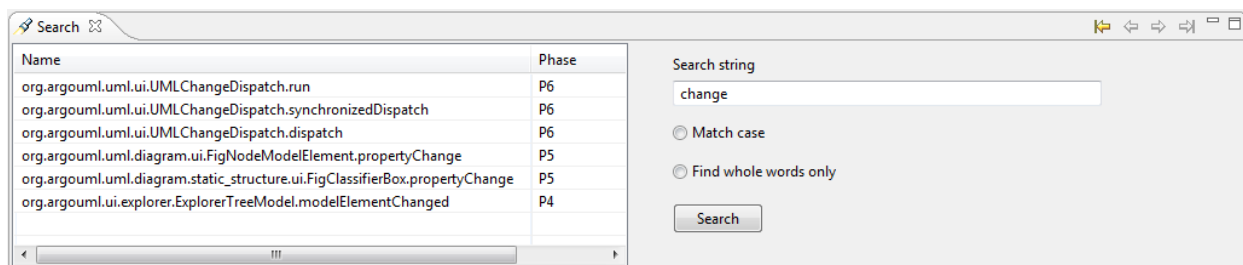
**Figure 3.34. Phase Property View**

Another view we have developed is the phase representative elements view, which shows the most important event(s) of each phase. These are detected using an algorithm developed by Pirzadeh et al. [PHS11]. The algorithm ranks the phase events based on their frequency within the phase in question and across phases. The idea is that the events that appear more in one phase and less in other phases are the most representative events. The authors conducted several experiments to support their approach. Extracting events that are most relevant to the implementation of a phase can help software engineers focus only on these events instead of going through the entire list of phase events. Hence, the objective is to improve the productivity of software engineers when attempting to understand how a particular event is implemented. Figure 3.35 shows an example of the phase representative elements view. In this figure, the methods that appear in the view are the ones that are deemed most relevant of the traced phase.



**Figure 3.35. Phase representative elements view**

Finally, we have also added many views that allow retrieving information from phases once extracted. The search view is perhaps the most effective one. The view is used to search through the events of all the detected phases. The result of search is shown on a table which has two columns. The first column contains the name of the event and the second one contains the name of the phases where the event occurs. Figure 3.36 represents the Search View.



**Figure 3.36. Search View**

## Chapter 4 EVALUATION

We conducted a user-based experiment to evaluate the effectiveness of the phase flow diagrams to help software engineers understand the content of a large trace. We also evaluated the usability of the new improvement made to SEAT to support the understanding of phase flow diagrams.

Before conducting the experiments, the first step of our research is to obtain the ethics approval from the Office of Research, which we did. We asked all participants to sign a consent form before starting the experiment.

### 4.1 Participants

To perform the study we asked ten individuals to participate to the study. We asked them to provide their experience level rank by answering four questions using a five-level scale: 1-Very Poor, 2-Poor, 3-Good, 4-Very Good, 5-Excellent.

The questions are as follows:

- Q1. Experience in software development,
- Q2. Knowledge of Object-Oriented programming,
- Q3. Experience with Eclipse
- Q4. Experience with Java

Since we are trying to divide the users based on their level of expertise, the experience in software development is important because during the study process we ask some questions about the efficiency of the tool and its implemented feature in performing

software maintenance tasks. So, if a user has more experience, he/she might have more expertise, better understanding of software maintenance process and the problem we encounter during the maintenance of an application which results into providing more careful and more reliable answers. Knowledge of object-oriented programming is a key measure for us because during the study we provide the ArgoUML[ARG] source code, which is an Object-oriented implemented application, to perform some tasks. So, knowledge of Object-oriented programming help users to perform the tasks more precise and provide more accurate answers to the questions we ask regarding to the performed tasks. As regards, the ArgoUML is a Java application, the knowledge of Java is important to accomplish the tasks in which investigation of the code is required and to give more exact answers to the questions we ask about those tasks. Because the implemented tool is an eclipse plug-in so the familiarity with eclipse can help users in using the tool.

Table 4.1 depicts the average level of participant's expertise in different areas.

**Table 4.1. Participants Average Experience**

Question	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10
Q1	3	3	5	4	4	4	4	5	5	5
Q2	3	3	3	4	4	4	4	4	5	5
Q3	3	3	2	4	4	4	4	4	4	4
Q4	3	3	2	4	4	4	4	4	4	5
Average	3	3	3	4	4	4	4	4.25	4.5	4.75

We grouped the participants based on the average of their answers to three groups: Intermediate, Experienced and expert.

**Table 4.2. Expertise of participants**

<b>Group</b>	<b>Participants</b>
Intermediate	P1, P2, P3
Experienced	P4, P5, P6, P7
Expert	P8, P9, P10

Participants P1, P2 and P3 fit into Intermediate group because their average answer to the questions is less than 4. We call the participants P4 to P7 experienced based on their average answer to the questions which is 4. Participants P8 to P10 belong to the expert group because their average answer to the questions is more than 4.

## 4.2 Trace File

During the study process, we provided the user with a trace file, collected from running ArgoUML (which is the target system), while a class diagram is created. ArgoUML is an open source UML modeling tool [ARG]. ArgoUML trace file is 7.70MB. The trace contains 37,739 events.

## 4.3 Experiment Process

To complete the study, the users were given the Experiment document, the trace files, and the ArogUML source code. The experiment document is composed of four parts: training, feature-oriented tasks, goal-oriented tasks, and debriefing questions.

### 4.3.1 Training

The goal of this section is to help the participants become familiar with the concepts and tools used throughout the experiment, namely, SEAT-ng, ArgoUML. We also introduce the scenarios for which the trace files were generated. The training document explains how to work with SEAT and initiates the user to the phase flow diagram notations. We also explained the target system ArgoUML. In average, it took around twenty minutes per participant to complete this task.

### 4.3.2 Feature-Oriented tasks

We asked the user to perform thirteen tasks that aim to evaluate the features of the new addition to SEAT for handling phase flow diagrams. Examples of these tasks include opening a new trace, applying the phase detection algorithm, selecting and merging phases, etc. The complete task list is shown in Table 4.3. The table also shows the number of users who successfully accomplished the task. The estimated time to complete this step was around ten minutes per participants.

Table 4.3. Feature-Oriented Tasks and Number of Successful Users

No.	Task	No. of successful users out of 10
1	Open thread-6.ftf trace file.	10
2	Apply the Phase Detection Algorithm.	10
3	Select one Phase and find its representative methods.	10
4	Select one Phase and find Start time and End time of that Phase.	10
5	Select one Phase and change its name.	10
6	Select an Automatic Detected Super Phase and go to its sub-phases.	10

7	Return back to the main Phase Flow Diagram.	10
8	Select two phases and unify them.	7
9	Select one phase and find all the events belong to that phase.	10
10	Select one phase and make it hidden.	10
11	Find the number of phases or sub-phases that contain “isMissing” event.	10
12	Select one phase and assign a type to that phase.	10
13	Select one phase and find the size of that phase (number of events).	8

As we can see in Table 4.3, the questions cover a large set of features that are implemented in the new SEAT. The idea is to test the ability of users to perform these simple and yet essential features. Uncovering usability issues at this stage is important since without this basic utilization of the tool, it would be hard for users to achieve specific goals with the tool (e.g., understanding the context of a trace).

Out of the thirteen tasks, Task 8 and Task 13 were the ones that were challenging. The reason of failing Task 13 was due to the ambiguity of the question as mentioned by one Intermediate (P1) and one experienced (P4) user who did not complete the tasks. For Task 8, three users could not perform the task including one Intermediate (P1) and two experienced (P6 and P7). They tried to find the ‘Unify’ menu by clicking on the selected phases instead of clicking on the phase editor. The following table (Table 4.4) shows the percentage of successful users in each task divided by users’ skill level groups. As the table depicts, we obtained an acceptable result in testing the usability of the tool’s features (phase flow diagram and its implemented features) although we had number of failures in two tasks.

**Table 4.4. Successful users in each level of expertise**

No.	Task	Intermediate %	Experienced %	Expert %
1	Open thread-6.ftf trace file.	100	100	100
2	Apply the Phase Detection Algorithm.	100	100	100
3	Select one Phase and find its representative methods.	100	100	100
4	Select one Phase and find Start time and End time of that Phase.	100	100	100
5	Select one Phase and change its name.	100	100	100
6	Select an Automatic Detected Super Phase and go to its sub-phases.	100	100	100
7	Return back to the main Phase Flow Diagram.	100	100	100
8	Select two phases and unify them.	66.7	50	100
9	Select one phase and find all the events belong to that phase.	100	100	100
10	Select one phase and make it hidden.	100	100	100
11	Find the number of phases or sub-phases that contain “isMissing” event.	100	100	100
12	Select one phase and assign a type to that phase.	100	100	100
13	Select one phase and find the size of that phase (number of events).	66.7	75	100

#### 4.3.3 Goal-Oriented Tasks


The goal of these tasks is to investigate the usability of the tool and the phase flow diagram to help software engineers understand the traced scenario through the analysis of its corresponding trace. To achieve this, we gave the users a trace file generated for a



specific ArgoUML scenario. The scenario consisted of running ArgoUML, creating a class diagram with one class, changing the name of the class, and finally closing ArgoUML.


Table 4.5 shows the list of tasks and the percentage of correct answers. Table 4.6 shows the average results divided by participants' expertise level. If the user did not provide an answer, we considered this as a wrong answer.

**Table 4.5. Goal-Oriented Tasks and Average Results**

NO.	Tasks	Total Correct answers %
1	What are the main execution phases of the traced scenario? Complete the task by looking at the ArgoUML source code and the trace file only.	80
2	Now, complete the previous task by using SEAT-ng.	90
3	Which phase or phases contain(s) the events creating the main user interface of ArgoUML?	100
4	Which phase or phases contain(s) the events since you click on the notation tool bar to select a class till you drop the class on the class diagram?	90
5	Which phase or phases contain(s) the events participating in assigning a name to the dropped class?	80
6	Which phase or phases contain the events since you click on the "Close" button (  ) on the top-right of the window till the application is completely terminated?	80
7	Find the five most important classes or interfaces contributing to the drawing of a class in the discussed scenario by looking at ArgoUML source code and using the given trace file.	80

8	Complete the previous task by using Seat-ng.	70
9	Find the method(s) which return(s) the name of each figure (e.g. Class, State and etc.) in ArgoUML by using Seat-ng.	80

**Table 4.6. Goal-oriented tasks scores per participant expertise level**

NO.	Tasks	Correct answers %		
		Intermediate	Experienced	Expert
1	What are the main execution phases of the traced scenario? Complete the task by looking at the ArgoUML source code and the trace file only.	100	75	66.6
2	Now, complete the previous task by using Seat-ng.	100	75	100
3	Which phase or phases contain(s) the events creating the main user interface of ArgoUML?	100	100	100
4	Which phase or phases contain(s) the events since you click on the notation tool bar to select a class till you drop the class on the class diagram?	100	75	100
5	Which phase or phases contain(s) the events participating in assigning a name to the dropped class?	100	75	66.7
6	Which phase or phases contain the events since you click on the “Close” button (  ) on the top-right of the window till the application is completely terminated?	100	75	66.7
7	Find the five important classes or interfaces contributing to the drawing of a class in the discussed scenario by looking at ArgoUML source code and using the given trace file.	100	75	66.7
8	Complete the previous task by using Seat-ng.	66.7	75	66.7
9	Find the method(s) which return(s) the name of each figure (e.g. Class, State and etc.) in ArgoUML by using Seat-ng.	100	75	66.7

Note that, for Tasks 1 and 7, on the contrary of other tasks, the desired answer was failure to provide any output which demonstrates the difficulty of finding execution phases (in

Task 1) and finding important classes or interfaces without using Seat-ng. Two participants, P7 (Experienced) and P10 (Experienced) did not provide any answer to both tasks (the participants were asked to write something to show the difficulty of doing this task) that is why the average of correct answers for these tasks (Task 1 and Task 7) is 80%.

For Task 2 one of the experienced users (P5) answered “OK” instead of providing the exact number of phases. We considered this answer as the wrong answer so the average of correct answers for this task is 90%.

For Task 4, the average of correct answers is 90%. Participant P5 provided a wrong answer. For Task 5, two participants including one experienced and one expert (P5 and P8) were not successful in performing the tasks.

For Task 6, the same participants who provided wrong answers for Task 5 (P5, P8) gave wrong answer to this task too and brought down the average of correct answer to 80% for this task.

The task 8 was designed to show how much the tool can help in maintenance activities and the average of correct answer for this task is 70%. Participant P1 (Intermediate) mentioned number of function’s name instead of class name and just two out of five answers of two other participants consist of one expert (P8) and one experienced (P5) were correct so all the answers were considered wrong.

#### 4.3.4 Questionnaire

We conducted a debriefing session with the participants to obtain their overall feedback on their experience using the tool. The questions are shown in Table 4.7. We asked the

participants to rank nine statements on a scale from 1 to 5 (1-Totally disagree, 2-Disagree, 3-Average, 4-Agree, 5-Totally agree).

**Table 4.7. Average of participants responses to statements**

<b>NO.</b>	<b>Statement</b>	<b>Intermediate</b>	<b>Experienced</b>	<b>Expert</b>
1	Overall, the automatic extraction of phases and the phase flow diagram are good features to have in a trace analysis tool.	4.6	4.5	5
2	The phases and phase flow diagram allows me to quickly explore the trace content.	4.6	5	4.6
3	The phase flow diagram notations are easy to learn and understand.	4.6	4.2	4.3
4	Negated: (I can find phases in a trace just by browsing the trace content (no need for the automatic extraction of phases and the corresponding new diagram)).	5	5	5
5	I believe the concept of execution phases can help in software maintenance task.	4.3	4.7	4.6
6	While performing the tasks, information from Seat-ng (Phase Flow Diagram, etc.) was not distracting.	4.6	4.5	4.6
7	The execution phases and the phase flow diagram help to find places in the code where a specific computation is located.	4.3	4.2	4.6
8	The execution phases and the phase flow diagram help to identify the main methods that implement a specific scenario.	5	4.5	4.6
9	I prefer using Seat-ng instead of traditional approaches for understanding how a scenario is implemented.	5	5	4.3

The objective of the first statement is to know what users generally think about having automatic execution of phases and a phase flow diagram as a mean to analyze execution traces. The average of scales given by intermediate users, experienced users and expert users respectively are 4.6, 4.6 and 5. As shown in Table 4.7, the scores of intermediate and experienced are similar. They are also very close to expert users' score and the average of scales to this question given by all ten users is more than 4.5 (agree). So, we can conclude that the automatic extraction of execution phases and phase flow diagram are good features to have and it is helpful for all kind of users.

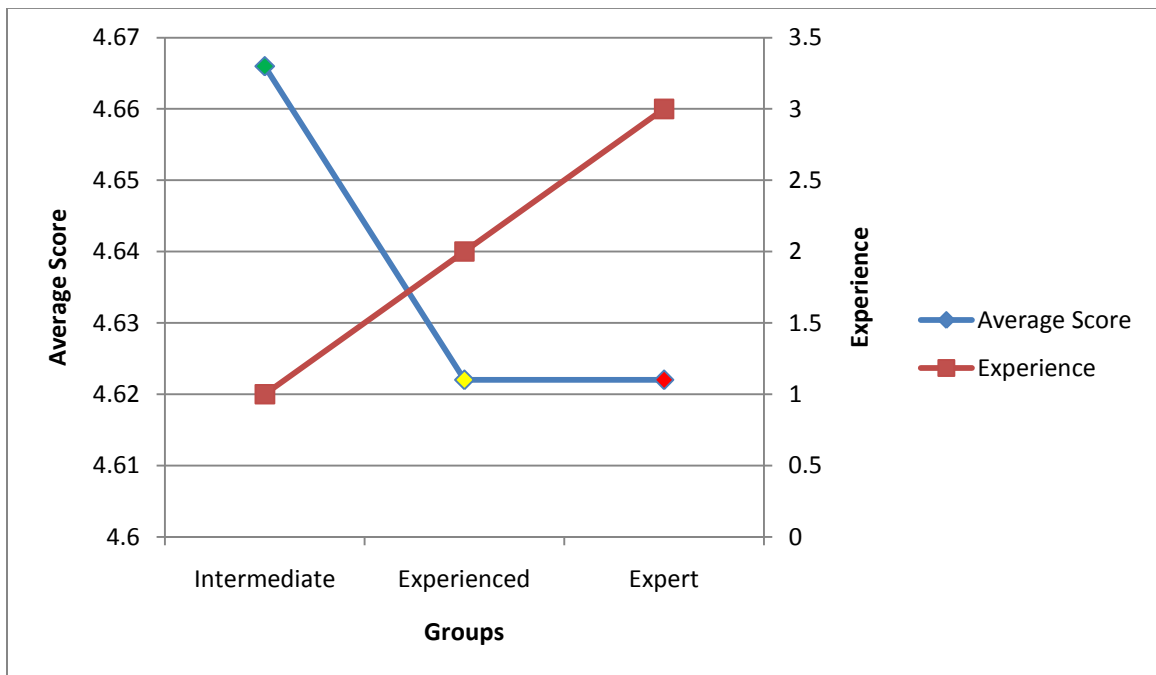
The second statement is used to evaluate if the execution phases and the phase flow diagram could speed up the process of navigating the trace content. The average of score provided by intermediate and expert users is 4.6. Experienced users believe that execution phases and the phase flow diagram help them to dig into the trace content very fast since the average of their scores to this statement is 5. Based on the closeness of the average of scores provided by each level of users to this question and since the average of scores to this statement given by all groups of users is more than 4 (agree), we can conclude that execution phases and phase flow diagram can help users with different level of expertise to quickly explore the trace content.

The third statement purpose is to evaluate whether the phase flow diagram and its notations are easy to understand by user. The average score of this statement is 4.6, 4.5 and 5 respectively presented by intermediate, experience and expert users which confirms that the diagram and notations are easy to understand by users with different level of expertise.

The aim of statements 4 to 8 is to gather users' opinion on using the tool and the phase flow diagram in maintenance activities. Based on the average of scores given by all ten participants to statement 4 which is 5 out of 5, all the users completely agree that it is not possible to find execution phases just by opening a trace file and looking at the trace events and a phase detection tool like ours tool is required . The objective of statement 7 is to evaluate if the phase flow diagram and execution phases can help users find a specific place in the code which belongs to a specific computation. Among all the users, the expert ones are the most confident (4.6 out of 5) that PFD and execution phases are effective. The overall conclusion based on the average of scores given to this question (4.3 out of 5) by all groups of users is that the execution phases and PFD can help them find a particular place in the code. In statement 8, we ask users if the tool, execution phases and phase flow diagram help them to find important methods involved in implementation of a specific scenario. Intermediate users are totally agreed that tool and PFD can help them find important methods by giving the average score of 5 to this statement. The average score give by experienced users is 4.5 and expert users have given the average score of 4.6 to this statement. Since they are all agreed on this statement, we can say that execution phases and the phase flow diagram can help users with different level of expertise to find important methods. The purpose of statement 5 is to ask users if the tool and implemented features can help is software maintenance. Among the three groups of users, the intermediates agree (average scale of 4.3 out of 5) that the tool can be used to facilitate the software maintenance tasks although the level of confidence of experienced and expert users is higher than this group. Based on the average scores

provided by all groups (4.5 out of 5) to statement 6, they are unanimous that the information provided by the tool was not distractive.

In the following diagram ( Figure 4.1) the left vertical axis represents the average of scores given to questionnaire statements and the right axis demonstrates the level of expertise. The horizontal axis shows the groups of participants. The first point (green) in the blue line shows the average of scores to all the statements provided by intermediate users which is 4.66. The second point (yellow) depicts the average of scores given to all the statements provided by experienced users which is 4.62 and the last one (red) shows the average of scores given to all the statements provided by expert users which is 4.62.

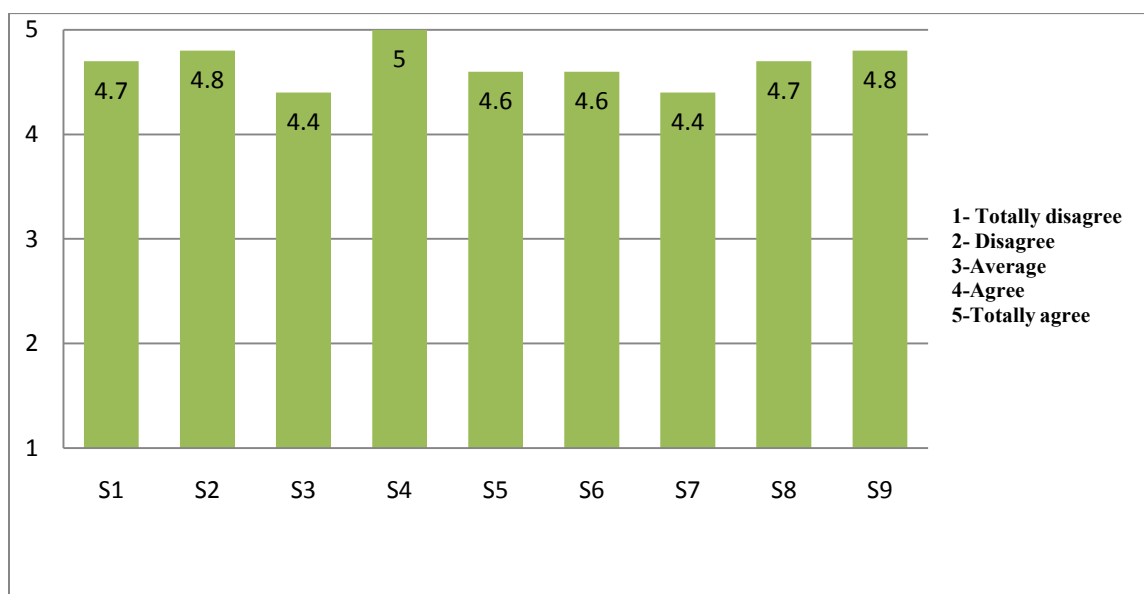


**Figure 4.1 Average Score / Groups/ Experience**

Figure 4.1 shows, the overall results obtained by all participants. The figure shows that the tool (Seat-ng) and its features are more helpful and effective for intermediate users than the other users. However, since the average of scores give by all participants to each statement are all over 4 (Figure 4.2) then we can conclude that the tool can help users with varying levels of expertise to perform software maintenance tasks that necessitate understanding the trace content .

Figure 4.2 represents the average of participants' responses to the statements mentioned in Table 4.7. The vertical axis represents scores of responses to questionnaire statements. Where, the horizontal axis shows the statements (S1 is equivalent to Statement 1).

As shown below, the average of scores to our statements is more than 4. For statement 4 the desired answer was 1 (negated to 5) and the result is quite satisfactory. This question is a check question to make sure that the participants are providing consistent answers.



**Figure 4.2 Average of participants' scores to statements**



In general, the participants were able to quickly identify (and recognize) program phases using the new diagram. All participants acknowledged that this task would have been more difficult if one needs to go through all trace events and diagram helps to perform maintenance tasks. They also find the tool usable despite some glitches due mainly to the instability of the tool.

#### 4.4 Limitations

In our case study, we evaluated the approach using one trace file only generated from ArgoUML. To generalize our results, we must continue to experiment with other traces. We also need to target specific maintenance tasks such as debugging and feature enhancement to understand the effectiveness of trace segmentation and the phase flow diagram.

This study focus on one approach to solving the problem of extracting execution phases from traces, which is the TSR approach [PH11a]. TSR is still an experimental approach. It does not guarantee 100% accuracy. Therefore, the resulting phase flow diagram may be flawed as well. We can overcome this limitation by investigating how trace segmentation (TSR) approach can be improved.

## Chapter 5 CONCLUSION AND FUTURE WORK

### 5.1 Review of the Research

Making changes during the maintenance phase is inevitable. To do so, we need to understand the software. Comprehending the software under maintenance is a challenge. In this thesis, we propose techniques that can help to understand the system behaviour. More particularly, we implemented a phase detection approach called TSR proposed by Pirzadeh et al. [PH11b]. The approach was developed to segment routine call traces by the repositioning of trace elements. We also present a new diagram, called the phase flow diagram, which is used to show the flow of execution phases. Along with the diagram's notations we defined some operation such as merge and unification that can be performed on flow of phases to ease navigation of the phase diagrams.

As already mentioned, to find the best way to represent the flow of phases, we started investigating the potential existing diagrams based on the guideline principles discussed in Section 2.1 to represent the required concepts such as sequence diagram, state diagram and activity diagram. We found that the activity diagram is the most suitable candidate because of the common concepts and definitions between the activity and phase flow diagrams such as the concepts of transition, join, fork and phase. The phase can, for example, be mapped to an activity. However, the activity diagram does not support all the requirements discussed in Section 3.2.2. So, we decided to extend the activity diagram.

We implemented the phase flow diagram and its features as an eclipse plug-in. This plug-in contains several views and editors to represent useful information regarding execution phases and facilitate the understanding process of the trace content.

Finally, we performed a user study to evaluate the phase flow diagram and its features. The study showed that the new plug-in can be used by software maintainers with varying levels of expertise to perform software maintenance tasks that necessitate understanding the trace content.

To build on this work, we need to experiment with many execution traces. We also need to select specific maintenance tasks and examine how the concept of execution phases, supported by the phase flow diagram, can help software maintainers achieve the task.

We also need to integrate other trace segmentation techniques since the TSR approach [PH11a] does not guarantee 100% accuracy as it is still at the experimental level.

The current version of the tool represents the flow of phases only in a single thread. Multi-threading has not been implemented yet. So the notations related to representing threads and the relationship between them such as join, fork and Inter-Thread Order Flow have not been implemented.

## 5.2 Contributions Highlights

As a part of this research, we implemented the components of the trace segmentation approach proposed by Pirzadeh et al. [PH11a] including the gravitational schemes, BIC-supported K-means clustering, the extraction of relevant information, the removal of utilities, and the detection of similar phases.

We also designed a new visualization technique, called Phase Flow Diagram, which is used to represent the flow of execution phases and their related information.

We implemented the phase flow diagram and the TSR approach as an extension to SEAT. SEAT is a powerful trace abstraction and analysis tool proposed by Fu et al. [HLF05].

We evaluated the automatic extraction of execution phases, phase flow diagram and its implemented features by conducting a user study.

### 5.3 Future Work

As a future work one possible improvement is to investigate a better way to represent the order of phases among multiple threads. The current approach results in diagrams that are cumbersome when the number of threads increases.

One could also explore additional phase types that cover a wider range of computational phases. Automatic identification of computational type of phases could be considered in the same context. In the current version of our tool, the user can only manually annotate a phase with its type.

Another avenue to explore is to perform additional empirical studies to evaluate the tool and its features by conducting controlled experiments. For example, we can have two groups of users. The first group will be given our tool and the phase flow diagram while the second user will be given traditional tools for exploring execution traces. Then, we compare the performance of both groups to see if the phase flow diagram can indeed help users be more productive users when solving specific tasks. The default hypothesis would be that there is not any difference between using and not using our tool (null hypothesis)

and we have to prove that the null hypothesis is wrong (alternative hypothesis). To prove the alternative hypothesis the tasks and questions must be well designed to help us to extract out the required information, and experimental errors must be taken into consideration to get a more precise result.

Another improvement could be the support for representing the flow of phase captured from multi-threaded trace files. The current version of the tool represents the flow of phases in a single thread and multi-threading has not been implemented yet. Finally, work could be directed towards improving the performance and the scalability of implementation of the TSR approach.

## References

- [AD07] S. Alam, and P. Dugerdil, “EvoSpaces Visualization Tool: Exploring Software Architecture in 3D”, *In Proc. of the 14th Working Conference on Reverse Engineering*, pp. 269-270, 2007.
- [ADAG10] F. Asadi, M. Di Penta, G. Antoniol, & Y-G Guéhéneuc, “A Heuristic-Based Approach to Identify Concepts in Execution Traces,” *In Proc. of the 14th European Conference on Software Maintenance and Reengineering*, pp. 31-40, 2010.
- [ARG] ArgoUML, URL: [argouml.tigris.org](http://argouml.tigris.org)
- [Bal99] T. Ball, “The concept of dynamic analysis,” *ACM SIGSOFT Software Engineering Notes*, 24(6), pp. 216-234, 1999.
- [Bas97] V. R. Basili, ”Evolving and packaging reading technologies,” *Journal of Systems and Software*, 38(1), pp. 3-12, 1997.
- [BPMN] BPMN 2.0 Specification: <http://www.omg.org/spec/BPMN/2.0/>
- [CDMZ07] B. Cornelissen, A. van Deursen, L. Moonen, & A. Zaidman, “Visualizing Test suites to Aid in Software Understanding,” *In Proc. of the 11th European Conference on Software Maintenance and Reengineering (CSMR’07)*, pp. 213-222, 2007.
- [CHZ+07] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. van Wijk, & A. van Deursen, “Understanding Execution Traces Using Massive Sequence and

- Circular Bundle Views,” *In Proc. of the 15th IEEE International Conference on Program Comprehension (ICPC '07)*, pp. 49-58, 2011.
- [Cor89] T. A. Corbi, “Program understanding: Challenge for the 1990s,” *IBM Systems Journal*, 28(2), pp. 294-306, 1989.
- [CZH+08] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, “ Execution trace analysis through massive sequence and circular bundle views,” *Journal of Systems and Software*, 81(12), pp. 2252 - 2268, 2008.
- [CZD+09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, & R. Koschke, “A Systematic Survey of Program Comprehension through Dynamic Analysis,” *IEEE Transactions on Software Engineering*, 35(5), pp. 684-702, 2009.
- [DA08] P. Dugerdil, and S. Alam, “Execution Trace Visualization in a 3D Space”, *In Proc. of the Fifth international Conference on information Technology: New Generations*, pp. 38-43, 2008.
- [DHKV93] W. De Pauw, R. Helm, D. Kimelman, & J. Vlissides, “Visualizing the behavior of object-oriented systems,” *In Proc. of the 8th annual conference on Object-oriented programming systems, languages, and applications*, pp. 326-337, 1993.
- [DJM+02] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J.M. Vlissides, & J. Yang, “Visualizing the Execution of Java Programs,” *Revised Lectures on Software Visualization, international Seminar, LNCS, vol. 2269*. pp. 151-162, 2001.

- [Dug07] P. Dugerdil, "Using trace sampling techniques to identify dynamic clusters of classes," *In Proc. of the IBM Conference of the Center for Advanced Studies on Collaborative research*, pp. 306-316, 2007.
- [FRC10] S. Frintrop, E. Rome, & H. I. Christensen, "Computational visual attention systems and their cognitive foundations," *ACM Transactions on Applied Perception*, 7(1), pp. 1-39, 2010.
- [HLF05] A. Hamou-Lhadj, and T. Lethbridge, and L. Fu, "SEAT: A Usable Trace Analysis Tool", *In Proc. of the International Conference on Program Comprehension*, pp. 157-160, 2005.
- [HFL04] A. Hamou-Lhadj, L. Fu, T. Lethbridge, "Challenges and Requirements for an Effective Trace Exploration Tool", *In Proc. of the International Conference on Program Comprehension (ICPC)*, pp 70-78, 2004.
- [HL02] A. Hamou-Lhadj, T. C. Lethbridge, "Compression techniques to simplify the analysis of large execution traces", *In Proc. of the 10th International Workshop on Program Comprehension*, pp. 159-168, 2002.
- [HL06] A. Hamou-Lhadj, T.C. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *14th IEEE International Conference on Program Comprehension (ICPC)*, pp. 181-190, 2006.
- [Hya10] Hyades Tool - URL: <http://www.eclipse.org/hyades>
- [IEE98] IEEE Std. 1219-1998, "Standard for Software Maintenance, IEEE Computer Society Press", Los Alamitos, CA, 1998.



- [JAM] Java-ML - URL: <http://java-ml.sourceforge.net/>
- [JR97] D. Jerding, S. Rugaber, “Using visualization for architectural localization and extraction”, *In Proc. of the Fourth Working Conference on Reverse Engineering*, pp. 56-65, 1997.
- [Kni00] C. Knight, “System and Software Visualisation”, *Handbook of Software Eng. and Knowledge Eng. World Scientific*, 2000.
- [Kof99] K. Koffka, “Principles of Gestalt Psychology”, *Harcourt, New York, Routledge reprint, ISBN: 0415209625*, p. 732, 1999.
- [PAH10] H. Pirzadeh, A. Agarwal, A. Hamou-Lhadj, “An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension”, *International Conference on Software Engineering Research, Management and Applications*, pp. 207-214, 2010.
- [Pen87] N. Pennington, “Comprehension strategies in programming”, *Empirical Studies of Programmer: Second Workshop*, pp. 100-113, 1987.
- [PH11a] H. Pirzadeh, A. Hamou-Lhadj, “A software behaviour analysis framework based on the human perception systems”, *Proceeding of the 33rd International Conference on Software Engineering New Ideas and Emerging Results Track*, pp. 948–951, 2011.
- [PH11b] H. Pirzadeh, A. Hamou-Lhadj, “A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension”, *16th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 221-230, 2011.

- [PHS11c] H. Pirzadeh, A. Hamou-Lhadj, M. Shah, “Exploiting text mining techniques in the analysis of execution traces”, *27th IEEE International Conference on Software Maintenance*, pp. 223 – 232, 2011.
- [Pri10] R. J. Price, “Automatic stop word identification and compensation”, *US Patent*, pp. 720-792, 2010.
- [PSH+11] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, A. Shafiee, “Stratified Sampling of Execution Traces: Execution Phases Serving as Strata”, *Science of Computer Programming*, 2012.
- [PSHM11] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, A. Mehrabian, “The Concept of Stratified Sampling of Execution Traces”, *19th International Conference on Program Comprehension*, pp. 225-226, 2011.
- [Rei05] S. P. Reiss, “Dynamic detection and visualization of software phases”, *Proceedings of the third international workshop on Dynamic analysis*, pp. 1-6, 2005
- [Rei07] S. P. Reiss, “Visual representations of executing programs”, *Journal of Visual Languages & Computing*, pp. 126-148, 2007
- [RR99] M. Renieris, S. P. Reiss, “Almost: Exploring Program Traces”, *Workshop on New Paradigms in Information Visualization and Manipulation*, pp. 70 – 77, 1999
- [SF99] K. Smith-Gratto, M. Fisher, “Gestalt theory: A foundation for instructional screen design”, *Journal of Instructional Technology Systems*, pp. 361–371, 1999

- [Shn96] B. Shneiderman, “The eyes have it: a task by data type taxonomy for information visualizations”, *Proceedings 1996 IEEE Symposium on Visual Languages*, pp. 336-343, 1996
- [SKM01] T. Systä, K. Koskimies, H. Muller, “Shimba: an environment for reverse engineering Java software systems”, *Software: Practice and Experience*, pp. 371-394, 2001
- [SO94] P. Schyns, A. Oliva, “From blobs to boundary edges: Evidence for time and spatial scale dependent scene recognition”, *Psychological Science*, pp. 195 – 200, 1994
- [SPAM11] L. L. Silva, K. R. Paixao, S. de Amo, M. de A. Maia, “On the Use of Execution Trace Alignment for Driving Perfective Changes”, *15th European Conference on Software Maintenance and Reengineering*, pp. 221-230, 2011
- [Sto06] M.-A. Storey, “Theories, tools and research methods in program comprehension: past, present and future”, *Software Quality Journal*, pp. 187-208, 2006
- [UML] UML 2.4.1 Specification: <http://www.omg.org/spec/UML/>
- [VV95] A. Von Mayrhauser, A. M. Vans, “Program comprehension during software maintenance and evolution”, *IEEE computer society*, pp. 44-55, 1995
- [WKL+08] J. M. Wolfe, K. R. Kluender, D. M. Levi, L. M. Bartoshuk, R. S. Herz, R. L. Klatzky, S. J. Lederman, “Gestalt Grouping Principles, Sensation and Perception (2nd ed.)”, *Sinauer Associates*, pp. 78-80, 2008.