Automatic generation of execution traces and visualizing sequence diagrams

Srinivasan, K.

Award date:
2013

Automatic Generation of Execution Traces and Visualizing Sequence Diagrams

2IM91- Master Thesis ES Report

Author: ***Kaushik Srinivasan (0786828)***

Host Company: ***Research& Development Department, Canon Oce***

Supervisor Oce: ***Joost Janse , dr. Lou Somers***

Supervisor Tu/e: ***dr. Alexander Serebrenik***

Acknowledgement

I would like to express my deepest gratitude to Eindhoven University of Technology for providing me this wonderful opportunity to do a complete graduation project on Automatic generation of execution traces and visualizing sequence diagrams as a part of the Master's course, Embedded Systems

I started off my internship at a company, Oce Canon Group, from the third week of February 2013 and it has been a remarkable journey ever since. I would like to thank my supervisor Mr. Joost Janse for offering me this chance and tremendous support in this effort.

None of this would have been possible without the support and much needed encouragement of my graduate professor, Dr. Alexander Serebrenik. He has continuously enthralled me with different ideas and thought-provoking questions and corrected me whenever I was wrong.

I would also like to thank Professor Dr. L.J.A.M. Lou Somers for his invaluable guidance and motivation.

Abstract

Program comprehension is an important part in legacy software, since only through understanding software it can be enhanced or maintained. Program comprehension can be achieved by applying the concept of reverse engineering of software systems. To aid a better understanding of a program, important parts of the system behavior are represented using abstracted UML (Unified Modeling Language) sequence diagrams of use cases. This project presents an approach and tool to automatically instrument a Python application using source transformation technique and retrieve dynamic information of the system behavior during runtime. The use of dynamic information to aid in program comprehension poses a scalability issue. The information retrieved is very large which makes it that much difficult to successfully understand the interesting parts of the information. So the tool also implements some reduction rules to reduce the large amounts of information thereby improving the readability and comprehension of the sequence diagrams when visualized. Apart from the rules, a suitable visualization tool (ExTraVis) is chosen which can support visualization of large amounts of information together with various inbuilt features that can help find the interesting parts of the information. Finally, the ExTraVis is combined with another visualization tool MSc-generator to visualize a sequence diagram for the selected part of the interesting information. An experimental study is presented that evaluates the practicality of the analysis based on a use case of the application. The tool also validates precision of the output by comparing the automatically generated sequence diagram with manually sketched sequence diagram by the user.

Table of Contents

List of figures

List of Tables

Chapter 1 Introduction

A significant portion of the software life cycle includes software maintenance. The role of the software maintenance is to fix faults in the system or improve performance by adding new features, etc. The important aspect of software maintenance is to understand the behavior of the program under study. Program comprehension is achieved through extensive analysis of the source code of the system. But this approach is largely influenced by the complexity and the size of the system (large) leading to highly time consuming process. To be precise almost 50 to 60 % of the software life cycle costs is dedicated for program comprehension [1, 2]. Reverse-engineering can be used as an approach to support program comprehension of complex systems by automatically retrieving significant information from the source code of the system [3]. In general, a reverse engineering is defined as a process of analyzing a system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction [4]. Reverse engineering by itself involves only analysis, not changing the system. The purpose of reverse engineering is to understand a software system in order to facilitate enhancement, correction and maintaining documentation of the system itself. This is a much faster and better approach to aid program comprehension rather than inspecting the source code manually. Reverse engineering can be achieved through both static and dynamic analysis of the system [5].

UML language is a standard notation to represent the structure and behavior of a system by using the information retrieved during reversing engineering process [6]. In particular, UML sequence diagram can be used in visualizing or document a system's dynamic behavior. The diagram mainly visualizes sequence of message interactions between various components of the system. Due to its competence of visualizing various sequences of events reflecting the exact behavior of the system, sequence diagram are considered as one of the important diagram in UML language [7].

The goal of the project is to facilitate program comprehension by reverse engineering a Python application and visualize abstracted sequence diagrams representing the behavior of various use cases of the application. A dynamic analysis is used as an approach to analyze program executions to capture the relationship between the elements of the source code and extract these relationships as information during runtime [8]. This extracted information can later on be used in generating UML sequence diagrams. The extraction of information is achieved by inserting codes in program which computes and provides the required data during execution. One of the strong benefits dynamic analyses provide to support program understanding is that they provide the exact behavior of the software system.

The main contribution in this project is to implement a tool that automatically generates a sequence diagram from an application provided by the user. The idea is to retrieve the necessary dynamic information during runtime and store it as a trace file. This trace file is then analyzed in order to automatically classify patterns of the object interaction and subsequently generate the sequence diagram. However, dynamic approaches are often characterized by huge amounts of data, which

gives rise to scalability issues [9]. To solve the problem, some strategies were formulated and integrated during the implementation of the tool.

1.1 Design questions

The following are typical design questions of this project

- What are the approaches implemented in this project to effectively visualize large amounts of information retrieved during dynamic analysis of the source code?
- If more than one approach is used to effectively visualize large amounts of information, how can it be best combined to achieve the desired goal?
- How does sequence diagram representing a system behavior facilitate program comprehension?

The above questions are analyzed and motivated during the course of the project. The answers to the above design questions are provided in the Section 6.2

1.2 Report Organization

The rest of the chapter is organized as follows,

Chapter 1 presents an introduction to program comprehension and the need for reverse engineering of the source code. The problem and the research domain of the work are also discussed.

Chapter 2 surveys background knowledge and the related work gathered from various scientific literatures.

Chapter 3 discusses the requirements to reverse engineer a sequence diagram. Based on the requirements, architecture is proposed to reverse engineer a sequence diagram for an application. This chapter explains in detail each and every step from extracting dynamic information to modeling a sequence diagram.

Chapter 4 explains in detail about the implementation of each of the steps discussed in the architecture. This chapter also discusses in detail about the visualization chosen for the project.

Chapter 5 discusses in detail about the evaluation of the implemented tool using test cases. The yielded results are validated.

Chapter 6 has the conclusion of the work and the answers for the design questions.

Chapter 7 discusses about the various future work possibilities for this project.

Chapter 2: Related work

A survey of the reverse engineering of sequence diagrams was presented in [4]. In the same article, Briand et al. propose an approach to the reverse engineering of UML sequence diagrams for distributed Java applications. The paper implements the approach of adding entry and exit logging traces in the required parts of the application and extracting the dynamic information during runtime. A same approach was considered during implementation of the project.

Different techniques for the reverse engineering of sequence diagrams have been published [10, 11]. These papers also focus on reducing the amount of information from the trace file by applying certain rules for example removing repetitive occurring due to loops, etc. The paper [12] provides solutions to solve the problem of scalability. The research showed formulation of rules to identify repetitive patterns and redundant information in the trace file so as to reduce the trace file. Few rules were formulated of our own during the course of the project based on the understanding of these papers. The project presents an approach similar to the one in the paper [13] where a tool is developed to automatically instrument dynamic web applications and reverse engineer a sequence diagram from the generated trace file. A TXL tool was considered for source code transformation and the result from the generated trace file is visualized using UML toolset, Rational Software architect. A similar architecture is designed in our work. Many approaches to the extraction of dynamic models from object oriented systems based on dynamic analysis have been proposed in [4, 11, 14].

Papers [15, 16, 17, 18, 19] mainly focus on identifying and visualizing interactions using the generated execution traces. The main problem that was analyzed in these papers was how to effectively analyze or view large number of execution traces. The author considers user interactive visualizations as a solution to the above problem. Papers [15, 16] developed a visualization tool to effectively visualize interaction. These interactions that are visualized are similar to a sequence diagram but not exact. The tool developed in paper [16], "ExTraVis" is the main tool that is being used as a visualization tool in this project. Some effective features such as filtering and zooming are also investigate in these paper which help support our analysis while designing such features in the project.

Chapter 3 Design and Architecture

3.1 Requirements of a Sequence diagram

UML Sequence diagrams (Interaction diagram) are used to represent or model the flow of messages, events and actions between the objects. Sequence diagrams are used primarily to document and validate the architecture of the system by describing the sequence of actions that need to be performed to complete a scenario. It is also useful because they provide a dynamic view of the system behavior [7]. A sequence diagram, more often than not, represents the interaction between the object of the classes in a system unlike in this project, where it represents the interaction between the classes. This kind of interaction reduces the number of lifelines participating in the sequence message interactions. Section 3.3 discusses about the problem of scalability both in terms of number of lifelines and number of messages interacted. The method of message interaction between the classes of the system yields a solution to the scalability problem.

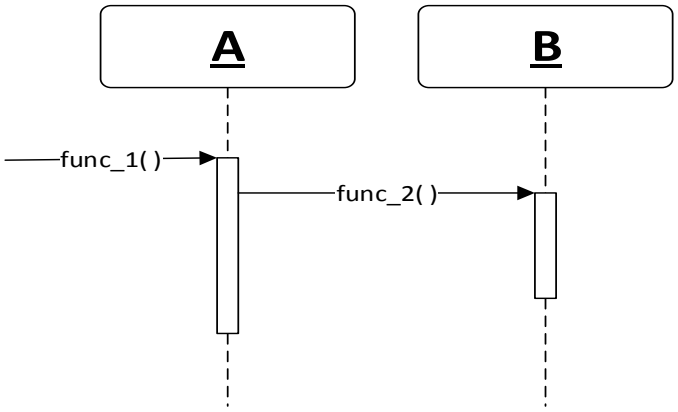

Figure 3- 1 Scenario of a sequence diagram

Figure 3-1 shows an example of a basic sequence diagram which depicts the interactions between two classes; 'A' and 'B. The vertical line represents the lifetime of a given object. The horizontal arrows represent message interaction between the classes. In software programming platform a normal communication occurs whenever a method or a function is invoked. So a message interacted between objects is always a method. The diagram depicts a small scenario where a function "$Func\_1()$" of class "A" is invoked externally and during the execution of function "$Func\_1()$" a function "$Func\_2()$" of class "B" was invoked. So the function "$Func\_2()$" is the message being interacted between class "A" and class"B". The life time of the execution of a function is depicted as a rectangular box on the object lifeline.

Table 3- 1 Requirements of a sequence diagram

To identify caller details(caller class)
To identify callee details(callee class)
To identify message details (Method signature)
To identify method logging type(Entry/Exit, timestamp)

The basic observation from the above scenario is that certain requirements are paramount to construct a sequence diagram. These requirements are listed in the Table 3-1 and explained in detail below.

- ***To identify caller details***: The caller details represent information occurring at the moment when a message is being sent. In this case, message sent occurs whenever a method is invoked. The information consists of only class of the caller. In the chapter 4, file name of the caller is also added to the information which will be explained later.
- ***To identify callee details***: The callee details represent information occurring at the moment when a message is received. In this case, message received occurs whenever a method execution starts. The information consists of only class of the callee. In the chapter 4, file name of the callee is also added to the information which will be explained later.

- ***To identify Message details:*** The message details represent information related to the details of the message being interacted. In this case, the details of the message interacted refers to the method signature. The information consists of the method (function) name and parameters passed.

- ***To identify Method Logging Type:*** The method logging details represent information to track the life time of a method being executed. The information consists of either an Entry or an Exit of a method and also timestamp of its occurrence.

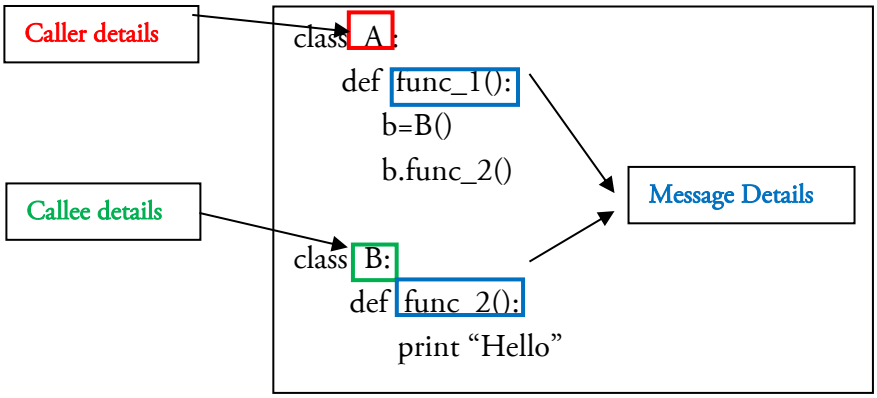

Figure 3- 2 Code scenario of the sequence diagram shown in figure 3.1

The figure 3-2 shows a sample Python code scenario mapping all the requirement details mentioned above. Figure 3-1 represents the behavior of the above code. It is clearly understood that all the details required can be obtained from the code. From capturing the requirements (in figure 3-1) to mapping them on to the code (in figure 3-2) represents a reverse engineering process. In the next chapter a detailed architecture is proposed which elaborates the process of extracting the above information dynamically and generating the required sequence diagram out of it[1].

3.2 Proposed Architecture

In this chapter, architecture is proposed that explains the approach to extract the required information from the application. Later the architecture also explains how with retrieved information a sequence diagram is modeled.

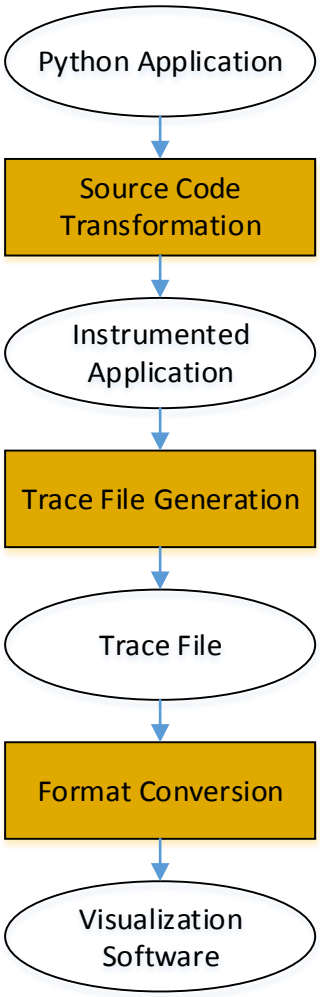

Figure 3- 3 Application to UML sequence diagram tool architecture

[1] Note: Since the goal of the project is to reverse engineer a Python application all the code scenarios explained in this report is in Python language

The flow chart shown in Figure 3-3 depicts the proposed architecture of the reserve engineering tool being implemented. The architecture comprises three steps to reverse engineer a sequence diagram from the given application. The first step is a source code transformation that parses a given application and instruments code automatically. The instrumented code includes program statements necessary to derive the behavior information which later on is retrieved during runtime. The next step is a trace file generation. Here the existing instrumented code is executed and the trace file is generated. The final step of the architecture is format conversion where the generated trace file is converted to a file of particular file format. The type of file format depends on the visualization tool used to generate the sequence diagram.

3.2.1 Source code transformation

Source code transformation is a process of transforming the original source code into modified code either by inserting an additional code fragments or by rewriting a code fragment in the source code. The modified code reflects the basic functionality of the original source code together with modifications implied by the transformation rules. These transformation rules are imposed conditions based on which the original code would be modified. For example the condition could be either to find the exact code fragments that needs to be modified or finding a specific location in the source code to insert a code fragment. In this project, the transformation of the code is done by adding specific fragments of the code in the necessary parts of the source code. This process is called *instrumentation* [20]. The main purpose of instrumenting an application is to extract informative details during the runtime of an application. These details are later analyzed and can be used to generate sequence diagram. Apart from the trace file generation, the instrumented application produces an output same as the output of the non-instrumented application

The code that is added during the *instrumentation* implements logging entry and exit statements. These entry logging is added wherever the method is entered and exit logging is added wherever the method is exited. The information that is extracted is method signature (method name and parameters passed), the classes of the targeted objects and method logging type (entry or exit).

3.2.2 Trace file Generation

In this step, the instrumented source code is executed on a suitable platform resulting in generation of a trace file. Since the application considered for the analysis is written in Python, a Python compiler is used to execute the given instrumented application. After execution a trace file (text file) is generated recording dynamic information. This process comes hand in hand with the source code transformation. This particular step does not need any considerable analysis. Even the format of the data that needs to be written in the trace file is done in the previous step of the architecture.

3.2.3 Format conversion

In this step of the architecture the already generated trace file is converted to a format compatible with specific visualization tool. The trace file consists of information as per the requirements of the sequence diagram. The input format file which is being converted varies with different visualization

tool. The converted file can be imported into the specific visualization tool and a sequence diagram can be simulated.

3.3 Feasibility of the proposed architecture

In this chapter, an analysis is done using a practical example to understand the feasibility of the proposed architecture. The main reason for the analysis is to see if the proposed architecture is feasible under the problem of scalability. Dynamic analysis approach is the main cause for producing large amounts data's leading to scalability issues. The data generated during runtime depends upon the following factors:

- Application Complexity: Complexity of an application can influence the generation of large data. The behavior of a complex system might sometime lead to execution of the instrumented code several times generating large data's. The number of files in the application can also impact the amount of data generated. Since increase in number of files could directly increase the number of instrumented code fragments.

- Instrumentation Complexity: This complexity arises whenever the data that needs to be extracted increases. If the dynamic data requirement increases then the amount of *instrumentation* of the application would also increase. This increase in amount of *instrumentation* leads to the scalability issues.

The scope of the project focuses more on the application complexity. A DiagnositicFramework application provided by the user was considered as a practically example for the feasibility study. A prototype of the currently proposed architecture was implemented and evaluated using the given application. DiagnosticFramework is an industrial Python application that visualizes various diagrams like pie chart, line chart etc representing the of error details of various sensors in the system. The data regarding the sensor values are recorded and stored in a csv file which is extracted during the visualization process. The application is developed from 200 files consisting of 189 classes and distributed over 9 packages. The application consists of many files there by guaranteeing the possibility of the size explosion problem with respective to number of classes involved and also the number of messages interacted during runtime. The first two steps of the architecture, source code transformation and trace file generation are implemented and evaluated.

The details of the implementation are not discussed in this chapter since there is a possibility that the current architecture might not be suitable for further implementation. If the outcome of the analysis shows that the current architecture is not useable then a revised architecture would be proposed. During the implementation of the first two steps a large trace file in the order of 100000 was indeed produced. The feasibility analysis mainly focuses on visualization part of the architecture. The goal is to validate the possibility of generating a sequence diagram for the produced huge trace file and also validate its usability.

For this purpose a visualization tool 'Msc - generator' is used. Msc-generator is free software licensed under the GNU General Public License (GPL) [21]. This version of msc-generator is heavily extended and completely rewritten version of the 0.8 version of Michael C McTernan's Msc-generator [22]. The reason for choosing this tool is because it is free open source software. The tool provides simple texts that are used to create, edit and understand the produced sequence diagram easily. Also the generated diagram from the textual description can be exported in various image formats file like PNG, PDF, SVG, EMF and EPS.

Figure 3- 4 Sample of dynamic information being transformed to a specific format and its corresponding sequence diagram in MSc-generator

Figure 3-4 (left) shows a sample of extracted dynamic information that has been transformed to an Msc-generator input file format. The textual description of this file format consists of necessary information to generate the sequence diagram. The text should always start with all the unique class names that are generated in the dynamic trace. After classification of the unique class names, the actual execution statements are written. Always in the execution statements, the class name specified on the left of the arrow (represented using a minus symbol '-'followed by a greater than symbol '>') is caller class and the one on the right is the callee class. For instance the last line shows that 'A' is caller class and 'B' is callee class and the message interacted between the class is "func2_()". The right side of the figure 3-4 is a simulated sequence diagram for the textual description. In the same manner as the above Figure 3-4 the large trace file generated during the feasibility analysis was converted to this specific format of the Msc-generator. A Python script was written to convert the huge trace file to a format compatible with the Msc-generator as shown in the left side of the figure 3-4.

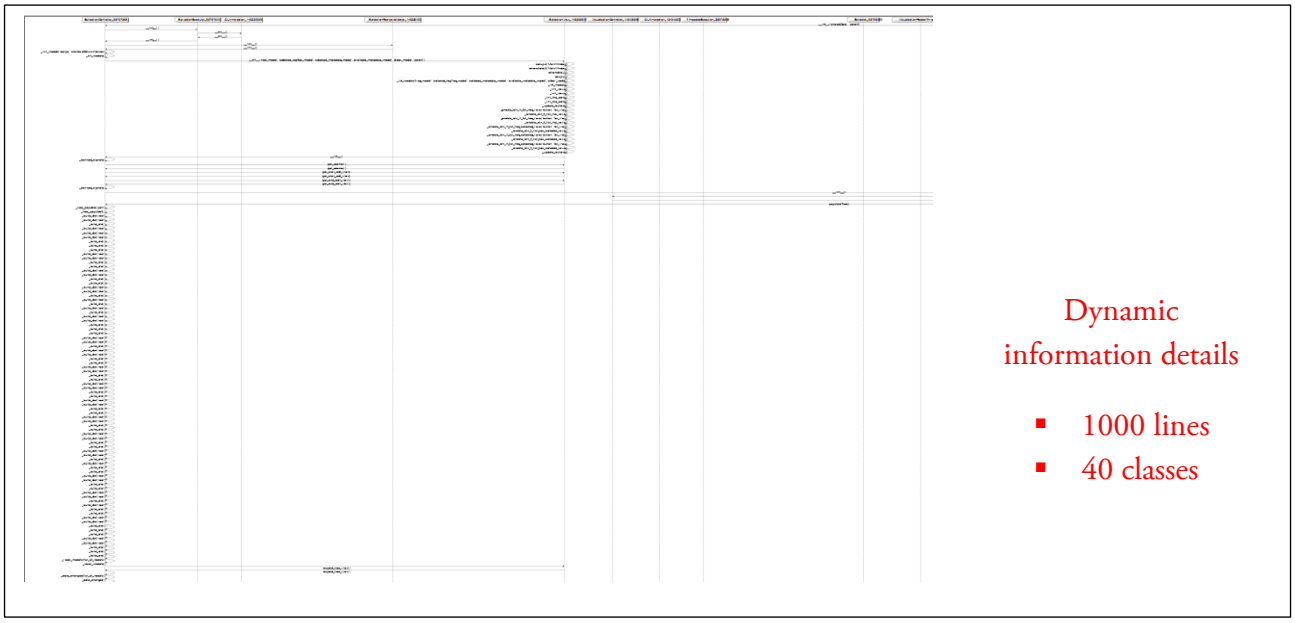

Figure 3- 5 Sample of dynamic information of the DiagnosticFramwork application being visualized as sequence diagram in MSc-generator

The Figure 3-5 shows a generated sequence diagram of a sample of the trace file. During the feasibility analysis a trace file consisting of 60,000 lines of execution with almost 120 unique classes were generated. The above Figure 3-5 was generated using just 1000 lines of the dynamic trace and yielded a non-usable sequence diagram. The feasibility analysis shows that the current proposed architecture is not suitable for visualizing large dynamic data. The current architecture needs to be revised such that it will resolve the problem of scalability and visualizes a sequence diagram which could be usable for the user. In left most corner of the Figure 3-5, a long column of self-calls can be observed. These self-calls are method calls that are occurring within the same class. These calls constitute a significant amount of execution traces in the trace file. Removal of self-calls are actually considered as one of solution for the problem of scalability and is discussed in detail in Section 3.4.1.2.

3.4 Revised Architecture

The figure 3-6 shows the updated architecture of the above proposed architecture. The previous architecture consisted of 3 main steps namely source code transformation, trace file generation and format conversion. To solve the problem of scalability the revised architecture consists of five main steps with the inclusion of Trace file analyzer and static analysis. The trace file analyzer step presents a set of techniques which aims at filtering the trace by removing unnecessary data with respect to program comprehension. The inclusion of static analysis to the architecture is to support the concept of abstraction. Instead of reducing execution traces in a trace file, hiding information through varying levels of abstraction is more efficient. For example traces can be extracted at the class level or at the architectural level, hiding the messages exchanged among classes of the same subsystem.

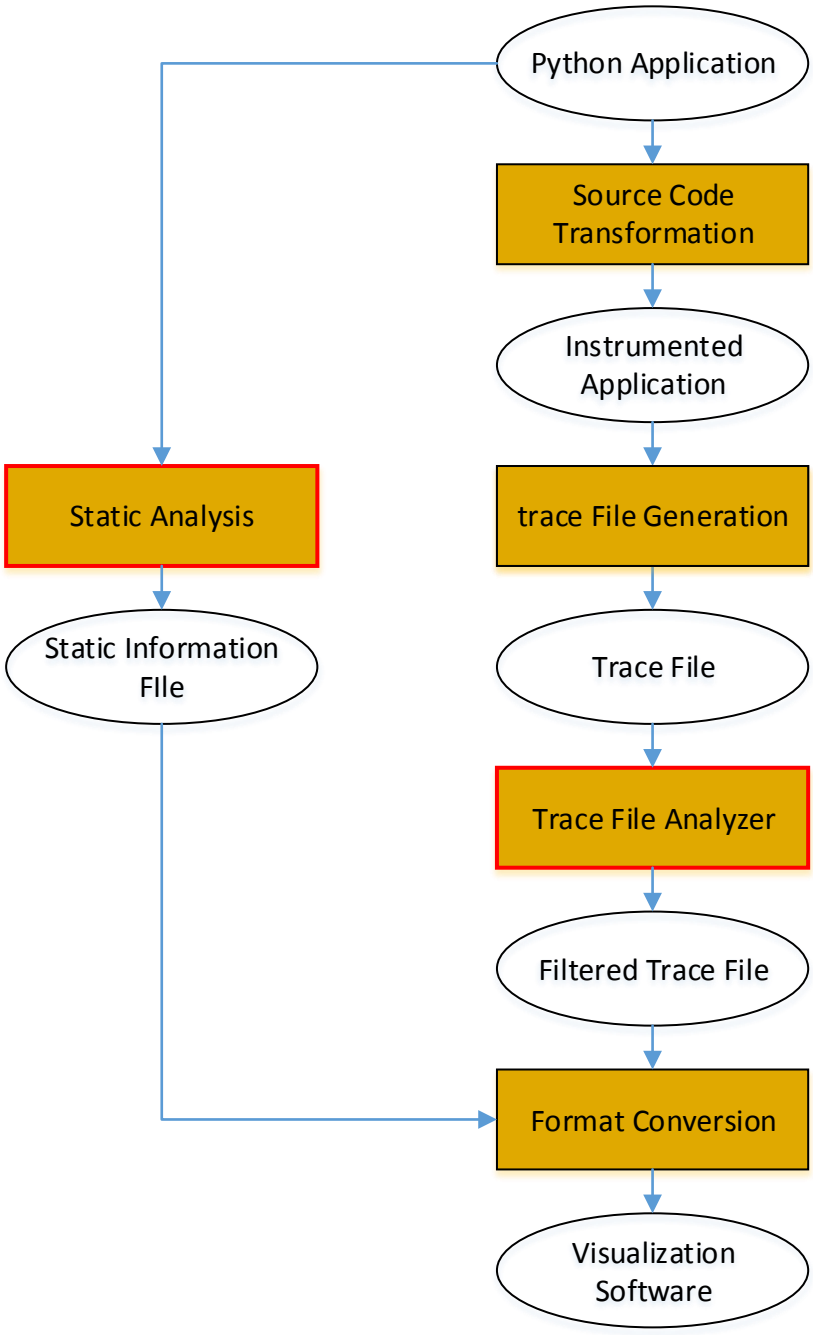

Figure 3- 6 Revised Application to UML sequence diagram tool architecture

3.4.1 Trace File analyzer

This is the third step in the revised architecture. As explained earlier, the generated execution traces can be very large and hard to understand. For a user identifying key important trace execution in the huge file would be very difficult. This is due to the fact that important interactions are mixed with low-level implementation details. To overcome the size explosion problem few reduction techniques are being analyzed in this step of the architecture. These reduction techniques are applied to the large trace file so that it aids the user to convenient explore the interested traces. Apart from the reduction rules analysis, the trace file analyzer step also includes an abstraction step.

The following are the various reduction rules that are analyzed to reduce the trace file:

- Redundant loop deduction
- Filtering "self" calls
- Filtering constructor calls.
- Extraction of unique method calls

Each of the reduction rules listed above are analyzed and designed in the coming subsections of this chapter.

3.4.1.1 Redundant Loop Reduction

The redundant loop deduction is one of the rules that detects and removes repetitions of sequence of method calls due to loops. A small example is considered to understand the basic working of the rule. This example illustrates the reason for the repetitive statements and also how the repetitive statements are filtered when subjected to the rule.

```
Class    A :
  def   func_1():
        b = B()
        iteration = 0
        while (iteration < 4):
            if (iteration%2 == 0):
                b.even()
          else:
                b.odd()
          iteration = iteration + 1

Class    B:
  def    even():
         print "Even number"
  def    odd():
         print "Odd number"
```

Figure 3- 7 Python code scenario of a repetitive message sequence due to loops

Figures 3-7 and 3-8 show a Python code scenario and the corresponding sequence diagram scenario respectively depicting an occurrence of repetitive sequence. The code scenario shows two classes "A" and "B" consisting of three functions; "func_1 ()" in class "A" and "even ()", "odd ()" in class "B". The "while loop" in function "func_1 ()" executes the function "even ()" whenever the "if" condition is true and executes the function "odd()"whenever the "if" condition is false. In general, the presence of "if-else" results in execution of different code statements in each iteration of loop subsequently producing non-repetitive trace executions during runtime.

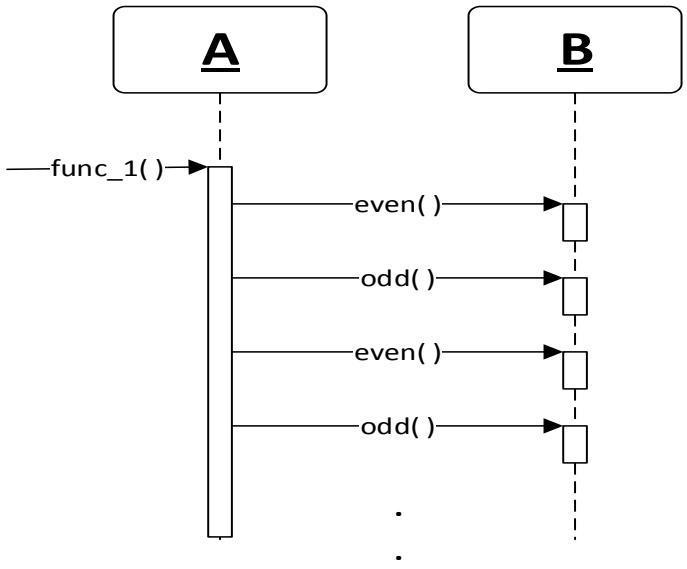

Figure 3- 8 Sequence diagram scenario of a repetitive message sequence due to loops

The repetitive execution of both the function alternatively is depicted as repetitive message interactions between the class A and class B in the sequence diagram shown in Figure 3-8.When the redundant loop deduction rule is applied to this case, all the executions except for the first executions of the function "even ()" and "odd ()" are removed from the dynamic information.

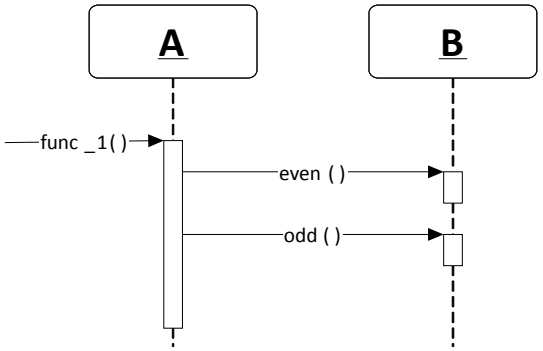

Figure 3- 9 Reduced Sequence diagram scenario when subjected to Redundant Loop deduction rule

The Figure 3-9 shows the transformed sequence diagram of the figure 3-8 after removal of all the repetitive statements by applying the rule. The implementation and working of the rule will be explained in detail in the Chapter 4.

3.4.1.2 Filtering "Self" calls

Self-calls are basically method calls that occur within a class. This is the same problem that was discussed in the Section 3.3 where a long column of self-calls being visualized in a sequence diagram. The reason for implementing the rule is basically to give more importance to sequence of interaction between different classes rather than a single class. This rule is user specific since some user's needs

information about self-calls. Same as the first rule an example is consider illustrating occurrence of self-calls and also the outcome of the rule when self-calls are filtered.

```
Class    A :
  def   func_1():
        a = A()
        a.func3()
        b = B()
        b.func2()

  def   func_3():
        print "world"

Class    B:
  def    func_2():
        print "hello"
```

Figure 3- 10 Python code scenario depicting the occurrence of self-call

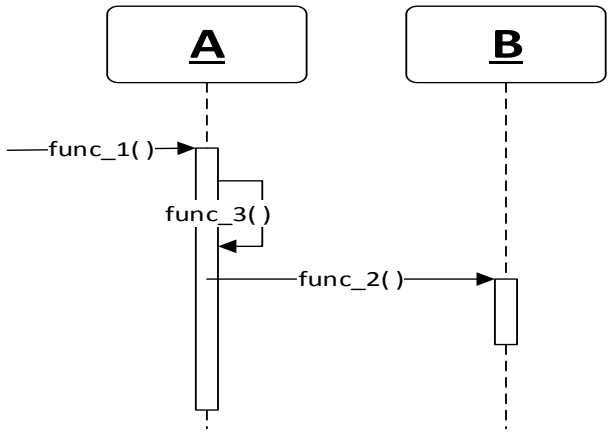

Figure 3- 11 Sequence diagram scenario depicting the occurrence of self-call

The figure 3-10 and figure 3-11 shows a Python code scenario and a sequence diagram scenario depicting an occurrence of self-calls. The code scenario shows two classes "A" and "B" consisting of three function; "func_1 ()" and "func_3" in class "A, "func2" in class "B". The function "func_3" is called by its own object of class "A". The resulting self-call execution is shows in the figure 3-11. When a self-call filter is used it detects the occurrence of the self-call from the trace file and removes these trace statements. In this case, the execution of the function "func_3" will be removed from the trace.

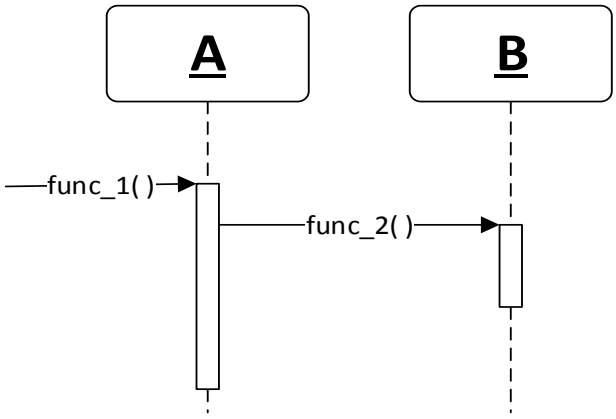

Figure 3- 12 Reduced Sequence diagram scenario when subjected to filtering "Self" calls rule

The figure 3-12 shows the transformed sequence diagram of the figure 3-11 after removal of all the self-call statements by applying the rule. The implementation and working of the rule will be explained in detail in the Chapter 4.

3.4.1.3 Filtering constructor calls

The constructor calls are invoked when a class object is created. The removal of these statements gives significance for the behavior of a specific scenario of a system. This reduction rule is also user specific since some users might require constructor calls to understand memory leaks in the system. Same as the other rules an example is consider illustrating the occurrence of constructor calls and also the outcome of the rule when constructor calls are filtered. The constructor calls are represented as "_init_()" functions in Python.

```
Class    A :
  def   func_1():
        b = B()
        b.func2()
class    B:
  def   __init__():
        pass
  def    func_2():
        print "hello"
```

Figure 3- 13 Python code scenario depicting the occurrence of constructor calls

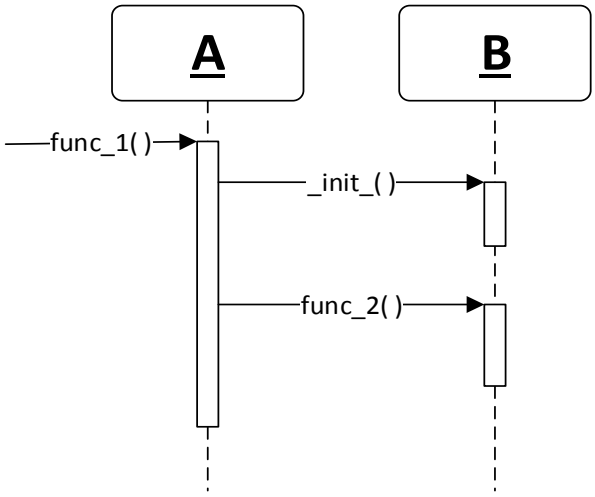

Figure 3- 14 Sequence diagram scenario depicting the occurrence of constructor calls

The figure 3-13 and figure 3-14 shows a Python code scenario and a sequence diagram scenario depicting an occurrence constructor calls. The code scenario shows two classes "A" and "B" consisting of three function; "func_1 ()" in class "A," _init_()" and "func2" in class "B". In the code when an object "b" of class B is created the constructor function "_init_" is invoked. This concept is applied for any object creation of any class. The resulting constructor call execution is shown in the figure 3-14. When a constructor call filter is used it detects the occurrence of the constructor call from the trace file and removes these trace statements. In this case, the execution of the function "_init_" will be removed from the trace file.

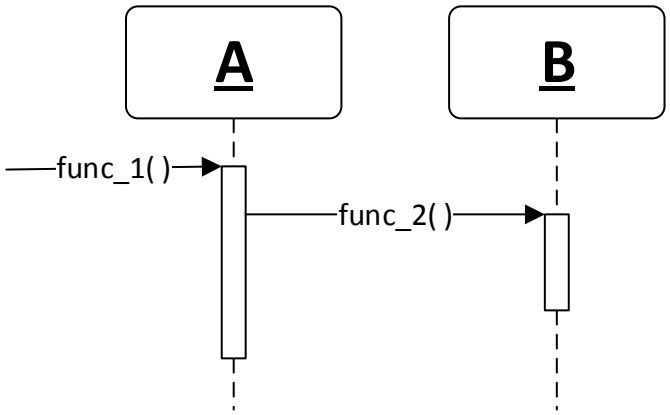

Figure 3- 15 Reduced Sequence diagram scenario when subjected to filtering constructor calls rule

The figure 3-15 shows the transformed sequence diagram of the figure 3-14 after removal of all the constructor call statements by applying the rule. The implementation and working of the rule will be explained in detail in the chapter 4

3.4.1.4 Extraction of unique method calls

This rule removes all function repetition occurring between two classes. These function repetitions statements are not influenced by loops instead they occur due to recursion or when a user performs same action multiple times. This reduction rules applied retain only unique messages exchanged between two classes.

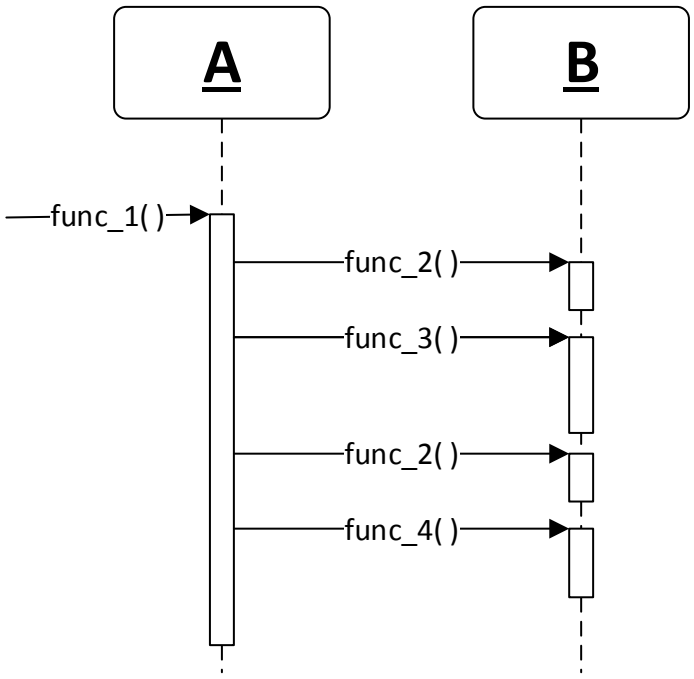

Figure 3- 16 A sequence diagram scenario depicting the occurrence of repetitive statements

Figure 3-16 shows a sequence diagram scenario depicting an occurrence of repetitive message. The call to the function *"func_2()"* is repeated. When *extraction of unique method call* rule is used all the repetitive function are filtered and only the unique function are retained. In this case the second occurrence of the function *"func_2()"* will be removed. Figure 3-17 is the reduced sequence diagram after removing all the repetitions.

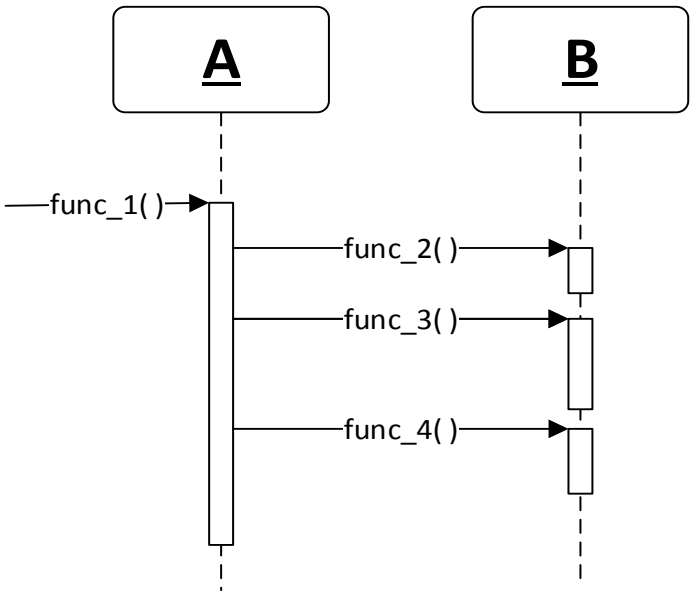

Figure 3- 17 Reduced Sequence diagram scenario when subjected to extraction of unique method calls rule

3.4.1.5 Information hiding

The concept of information hiding is to provide user with interesting parts of execution traces and hide the insignificant parts. This concept can be achieved my varying levels of abstraction. The depth of the levels depends upon the structure of the application. In our case the interaction between the classes of the system represents the lowest level and the interaction between directories of the system consisting of sets of file of an application is of the highest level. Even with the above reduction rules, applying the concept of information hiding is more efficient in solving the problem of scalability.

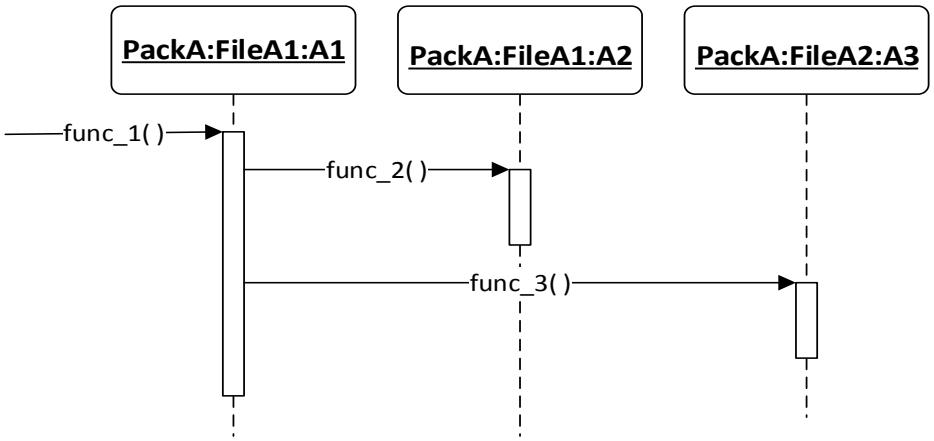

Figure 3- 18 Sequence diagram scenario at class level

The figure 3.18 shows sequence diagram with class level of abstraction. To understand the abstraction concept the naming of the object lifelines is different from the previous example. The naming of the object lifeline is represented by all the levels of the system. The above figure shows interaction between the classes “A1”, “A2” and“A3”. The classes “A1” and “A2” are defined in the file “fileA1” and classes “A3” in the file “fileA2”. Finally all the files “fileA1” and “fileA2” are present in a directory “PackA”. At the file level, classes of the same file are replaced by only one representation of that file. This is done in order to reduce the volume of information in the sequence diagrams and is useful to comprehend which files implement which application functionalities.

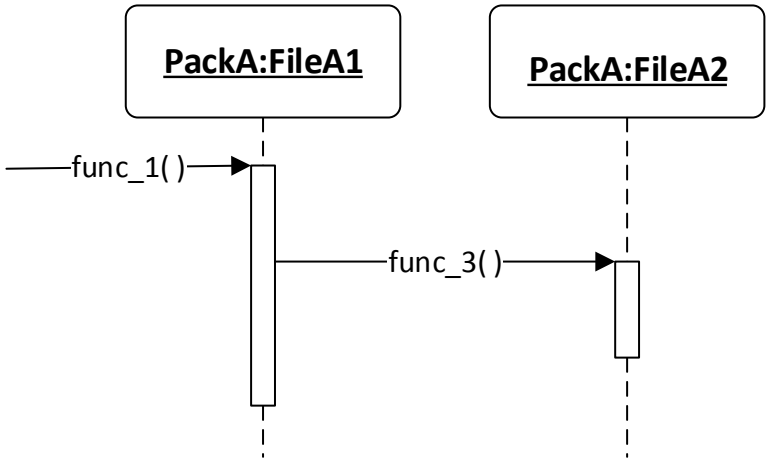

Figure 3- 19 Sequence diagram scenario at file level

The figure 3.19 shows interaction between the different files of the system. The interactions between the classes are hidden from the above figure. This shows that the calls become internal with respective to the files are hidden. The function “func_2 ()” is now removed from the figure since it was an interaction between classes of same file. The same concept can be applied for directory level abstraction. Files of the same directory are replaced by only one representation of that directory.

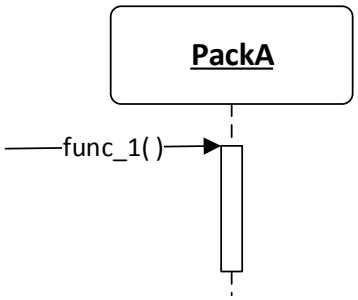

Figure 3- 20 Sequence diagram scenario at directory level

The figure 3.20 shows interaction between different directories of the system. All the interactions between the files are hidden showing only the interaction between the directories. The figure is also

the highest level of the system behavior. In abstraction process no data is lost for a user. The data's are just hidden.

Information hiding is a difficult concept to be applied to the current trace file analyzer step. To implement the various levels of abstraction, a separate user interface tool needs to be developed over the current tool. The proposed solution analyzed during the course of the project was found to be time consuming and unfit to produce desired results thereby mandating alternative visualization tool, choices of which will be explained in Section 4.4.

3.4.2 Extraction of Hierarchical structure of the system

This is the fourth step in the revised architecture which is an essential prerequisite to support the concept of information hiding. Information hiding can be achieved by retrieving the hierarchical structure of the systems which necessitates the need for incorporating static analysis. The structure of the system describes the levels of the application. The outcome of the static analysis represents the hierarchical structure of the given application: directories, files, classes and also describes the association of each file with their respective directories and likewise for each class with their respective file. The analysis produces a static information file consisting of the description of the above hierarchical relation. In the previous architecture, the format conversion step required only the information of the dynamic analysis. Since the current visualization tool support the information hiding it requires details of both the details of the hierarchical structure of the system and dynamic analysis. The data structure of the file format of the tool will be explained in detail in the Section 4.5.

3.5 Alternative architecture

The alternative architecture that has been analyzed is to reverse engineer a sequence diagram using a complete static approach. In our current architecture the source code transformation and generation of execution trace is completely based on a dynamic approach which can be replaced by a static approach. In this approach the information of method calls are retrieved by parsing the source code and sequentially tracking the path of the execution. It basically searches the source repeatedly to track the path of the execution till an end point is reached. So the required information for a sequence diagram can also be retrieved by this approach. The advantage of static approach is that there is no need for modifying any source code. Also the static analysis covers all possible code executions. It checks even the code fragments that get control very rarely. The main disadvantage of the static approach is that it is not as precise as the dynamic approach in determining the behavior of the application. Dynamic approach actually executes the system under test. This has the key advantage of having precise information available.

Chapter 4 Implementation of the tool

In this chapter, implementation for the each of the design steps in the architecture is explained. The selection of the platform to develop the tool in each of the steps is also explained in detail.

4.1 Implementation of source code transformation

As discussed earlier, fragments of code are inserted in parts of the application in order to retrieve data during runtime. By inserting these code fragments the original source code is transformed into an instrumented source code. The instrumentation process is implemented using Python programming language. The choice for Python is because the application that needs to be transformed is programmed in Python as well. Python language has many inbuilt libraries that can aid the process of source code transformation. Python also has a relatively simple syntax in comparison with other programming languages and hence it is less time consuming for the programmer to comprehend. Another reason is that it is a very powerful scripting language where you could parse any given application with ease. The parts of the application that needs to be instrumented are at the start and exit of a function in the application. These specific parts of the application are identified by parsing each file in an application. Figure 4-1 is a model of implementation to extract information and using this to identify the occurrence of functions in the application.

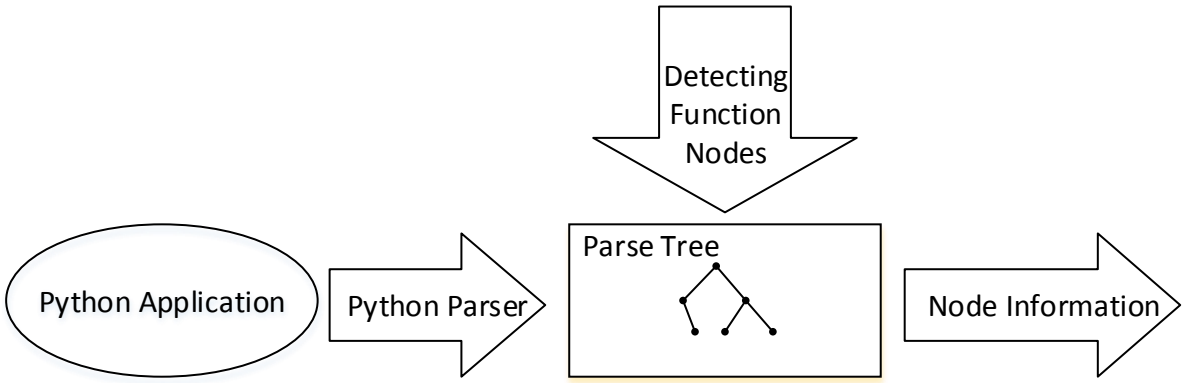

Figure 4 - 1 Shows a process of identify the specific part of an application

In this case, each file of the application was parsed and transformed into a parse tree. Figure 4-2 shows the transformation of a Python function statement into a parse tree. This particular statement is chosen as an example since all the methods or functions in Python language are similar to this statement. Additional information like line number of the program instruction, indentation offset of the program instruction are stored in each node of the parse tree during transformation. Once the parse tree is constructed, each node is inspected for the occurrences of Python function ("def") and if found, the respective node's line number information is retrieved.. The instrumentation process becomes easier once these line numbers are extracted.

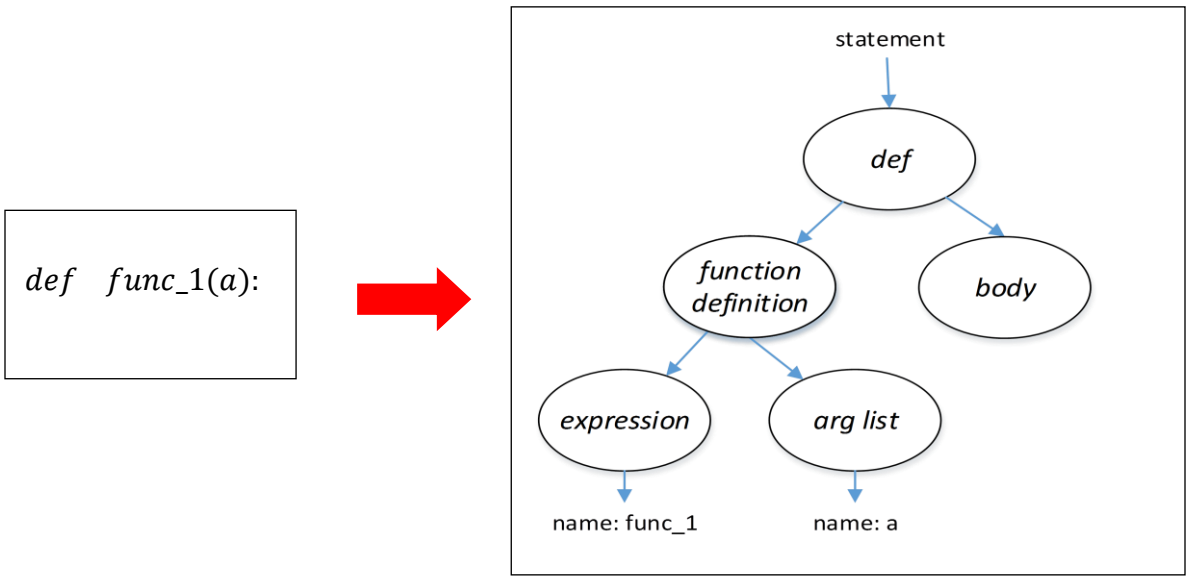

Figure 4 - 2 Shows a Python function transformed into a parse tree

Using the line numbers, the corresponding functions are tracked and code fragments are added at the starting and ending of that function. Figure 4-3 shows the result of a source code transformation of small fragment of a Python code. The top part of the Figure 4-3 is the original code and the bottom part of the Figure 4-3 is the transformed code. The transformed code clearly shows the addition of 'Enter' and 'Exit' line of code being added in the transformed code. The inserted code represents the information (callee details, caller details, method signature) being instrumented. The Figure 4-3 shows that during instrumentation none of the details of the data are directly written. Instead these data's are calculated during runtime by inspecting the call stack this process is called introspection. The reason for this approach is because the caller details can't be written directly during instrumentation process. This leads to only writing details of the callee, method signature and method logging type and subsequently generating a trace file during runtime consisting of only these details in each entry. After which for each entry in the trace file the caller detail can be calculated from the previous trace entries. This is a very lengthy process than calculating the details during runtime so the later approach was used. The call stack consists of all the method calls that have been invoked from start till the point of the current execution. So the callee, caller and method signature details can be retrieved from the call stack. With this the implementation part of the source code transformation is finished.

```
Class    A :
  def   func_1():
        b = B()
         b.func2()

Class    B:
  def    func_2():
         print "hello"
```

```
Class    A :
  def   func_1():
        Enter =' Enter',  stack(function), stack(parameters), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
        b = B()
        b.func2()
        Exit =' Exit',  stack(function), stack(return), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
Class    B:
  def    func_2():
       Enter =' Enter',  stack(function), stack(parameters), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
         print "hello"
       Exit =' Exit',  stack(function), stack(return), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
```

Figure 4 - 3 Source code transformation for small Python code

4.2 Implementations of trace file generation

A trace file is a simple text file generated when the instrumented application is executed. The file consists of information that is written during the execution of the inserted codes in the application. As discussed before, this information consists of data that can later on be used to generate sequence diagram. A python program is written to compile the instrumented code and subsequently generate a trace file. Apart from the trace file generation, the instrumented application produces an output same as the output of the non-instrumented application

4.2.1 Trace file format

The trace file consists of the collection of all the entry and exit statements of a method retrieved during runtime. Each line in a trace file consists of 8 different fields.

Table 4 - 1 Format of the trace file

Method Logging Type	Method Name	Method parameters/ return value	Caller Class	Caller File	Callee Class	Callee File	Timestamp

The above Table 4-1 shows the format of the generated trace file. The details of the each field have been discussed in Section 3.1. Table 4-2 is a sample of a trace file consisting of execution statements produced for the code fragment shown in the Figure 3-2.

Table 4 - 2 A sample of a trace file

Method Logging Type	Method Name	Method parameters/ return value	Caller Class	Caller File	Callee Class	Callee File	Timestamp
Entry	func_1	None	C	File1.py	A	File2.py	14:45:59:445000
Entry	func_2	None	A	File2.py	B	File2.py	14:45:59:880000
Exit	func_2	None	B	File2.py	A	File2.py	14:46:00:130000
Exit	func_1	None	A	File2.py	C	File1.py	14:46:00:145000

The first row in the Table 4-2 shows that a function "func_1" was entered because the Method Logging Type field was "Entry". Since the logging type was "Entry" the third column in the table represents "Names of the method parameter" that were passed. If the logging type was "Exit" then the third column would represent "return value". In this case no parameters are passed and so a value "None" is written in this field. Indeed, the code fragment in the Figure 3-2 shows that function "func_1" has no parameters. The function "func_1" was defined in the class "A" and this class is present in the file "File2.py". This information represents the callee details which are written in the columns 6 and 7. The columns 4 and 5 represent the caller details showing that the class C located in the file "File1.py" was used to initiate the function "func_1". The final column shows the exact time the function was entered. Likewise every row in the table is filled with trace entries similar to the first row.

4.3 Implementation of Trace File Analyzer

Section 3.4.1 discusses in detail design of various reduction rules. These reduction rules are applied to the trace file in order to reduce the number of lines. In this section, the implementation of each

reduction rules is explained in detail. The algorithms designed for each rule, discussed below, were implemented in a Python program.

4.3.1 Implementation of redundant loop reduction

The main purpose of the rule is to reduce all the repetitive statements due to the execution of loops. Also the presence of "if-else" statement may sometimes influences the execution resulting in non-repetitive statements between iterations. The present trace file does not contain information that differentiates statements that occur due to loops from the rest of the statements. In order to identify execution statements due to loops additional data requirements are updated which are retrieved dynamically during runtime.

Table 4 - 3 Revised Requirements

To identify caller details(caller class)
To identify callee details(callee class)
To identify message details (Method signature)
To identify method logging type(Entry/Exit, timestamp)
To identify loop details (Loop Entry, Loop Exit)

The Table 4-3 shows updated requirements of the Table 3-1, by including identifying loop details as one of the requirements. The significance of the added requirement is to identify details of the loops statements in the code which can subsequently support in identification of repetitive statements occurring due to loops in the trace file. Loop Entry and Loop Exit are the information that needs to be identified in the loop statements in the code. Again instrumentation process was used to retrieve the above details. The current source code transformation implementation is slightly modified so as to include the instrumentation of loops as well. To instrument loop statement, identifying the line number of the occurrence of the loops in the code is necessary. The process of parsing each file of an application and transforming into a parse tree remains the same. Now each node of the parse tree is inspected to find the occurrence of both the loop statements ("while", "for") as well as the occurrence of each function".

Using the line numbers, the corresponding loops are tracked and few statements are inserted. "Entry of loop" is added just before the start of a loop condition statement, "Exit of Loop" is added immediately after the body of the loop ends and "Iteration Number" is added immediately after the loop condition statement.

```
Class    A :
  def   func_1():
        b = B()
        iteration = 0
        "Entry of Loop_X"
        while  (iteration < 4):
             "Iteration Number_X    Y"
            if (iteration%2 == 0):
                b.even()
           else:
                b.odd()
           iteration = iteration + 1
       "Exit of Loop_X"
Class    B:
  def    even():
         print "Even number"
  def    odd():
         print "Odd number"
```

Figure 4 - 4 A sample of a Python code after inserting statements with respective to the loops

Figure 4-4 shows the result of a source code transformation for a code fragment. The Figure 4.4 clearly shows the addition of "Entry of loop_X", "Exit of Loop_X" and "Iteration Number_X Y" statements in the above code. The "X" present in the added statements represents a unique integer value to differentiate each loop that is instrumented in the application. The unique integer value helps in identifying a particular loop's corresponding "Entry of loop", "Exit of Loop" and "Iteration Number" statements when a trace file is generated. The "Y" present in the "iteration Number_X Y "is an integer value that starts from '1' and is incremented by '1' at each iteration of the loop. The "Y" value helps differentiate the statements executed at different iterations of the loop.

Table 4 - 4 A sample of a trace file with inserted loop statements

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
"Entry of Loop_1"			
"Iteration Number_1 1"			
Entry	even	A	B
Exit	even	B	A
"Iteration Number_1 2"			
Entry	odd	A	B
Exit	odd	B	A
"Iteration Number_1 3"			
Entry	even	A	B

Exit	even	B	A
"Iteration Number_1 4"			
Entry	odd	A	B
Exit	odd	B	A
"Exit of Loop"			
Exit	func_1	A	C

Table 4-4 shows the generated trace file for the code shown in the Figure 4-4. The "Entry of loop_X" and "Exit of loop_X" statements act as external boundaries for the statements executed by the entire loop. The "Iteration NumberX Y" statements act as a boundary for the statements executed during each iteration of the loop. These boundaries aid in tracking execution statements in each iteration.

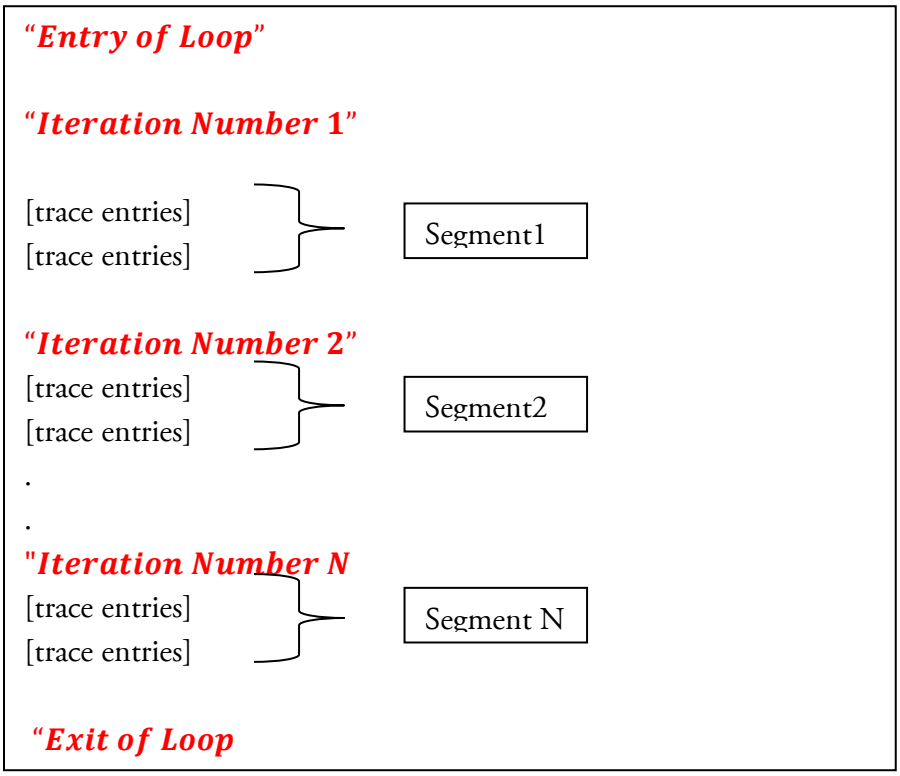

Figure 4 - 5 A generic diagram of a trace file with loops statements

Figure 4-5 shows a general diagram of a fragment of the trace file consisting of statements executed by the loops. The trace entries are call statements executed during the loop execution. The significance of the Figure 4-5 is to explain the algorithm designed to reduce the trace entries that are repetitive due to loops. The basic step of the algorithm is that all the trace entries in single iteration of a loop are considered as one single segment. All the trace entries under "Iteration Number1" are represented as "segment1" and so on until "Iteration Number N". After the trace entries are classified into segments, a comparison step is incorporated to compare all the segments and remove the redundant ones. In this step, one of the segments needs to be a reference segment to compare all

the other segments. The comparison step starts with the first segment (Segment 1) being the reference segment and is compared with rest (Segment2, Segment3….. SegmentN). If any of the comparisons yielded 'True' means that both the segments are the same. Then the segment that was compared with the reference segment (segment1) would be removed. Once the reference segment is compared with rest of the N segments and then the reference segment is assigned to the next immediate segment that was not removed during comparison. This new reference segment is now compared with rest of the available segments. In segment comparison each trace entries of a segment are compared with the corresponding trace entries of the other segment. The attributes that are used to compare the trace entries are "Method logging type ","Method Name, "Caller class" and "Callee class".

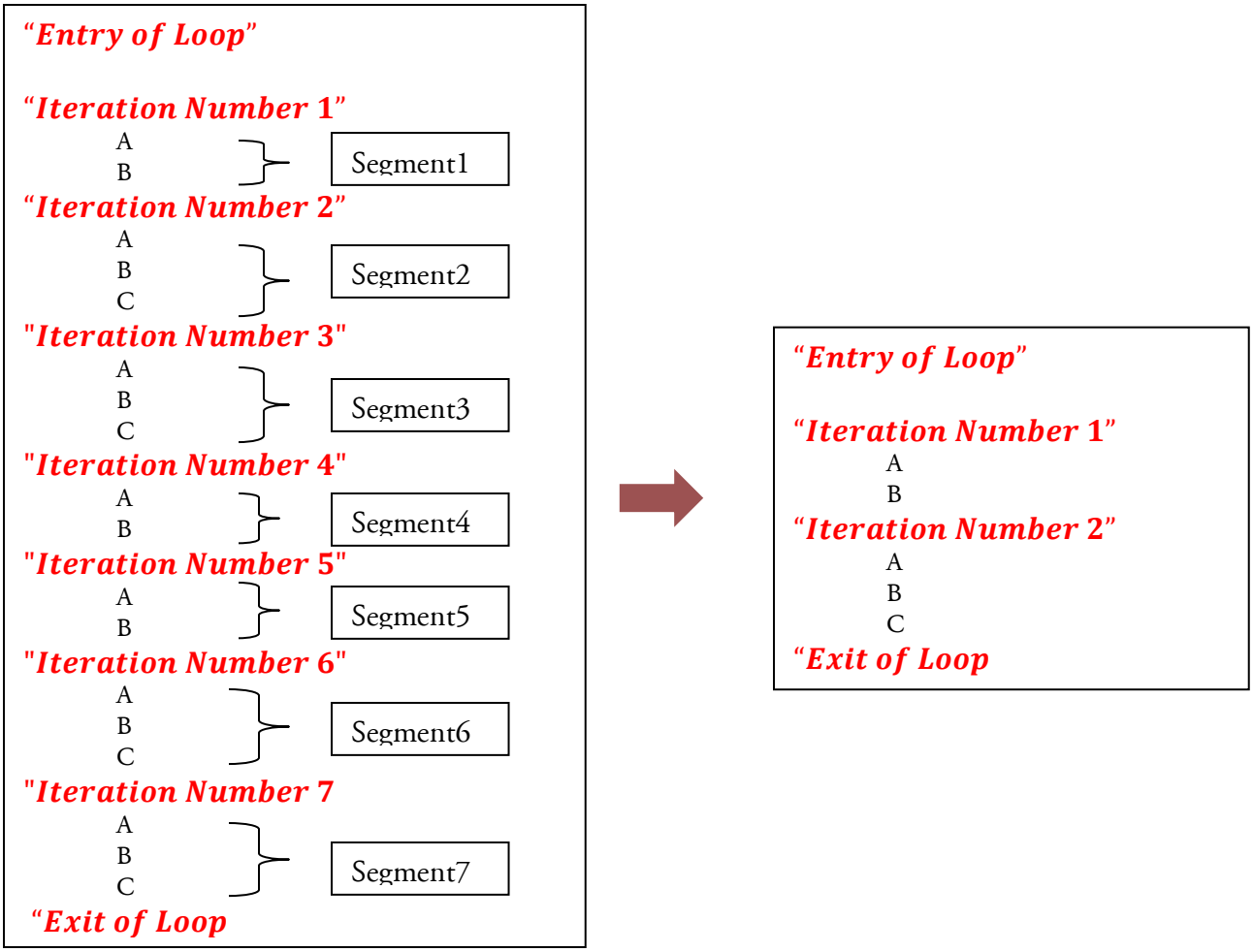

Figure 4 - 6 An example of transformation of the trace file due to the reduction algorithm

The reduction algorithm explained above is explained using an example as shown in the Figure 4-6. The left side of the Figure 4-6 shows the non-reduced trace file and the right side of the Figure 4-6 shows a reduced trace file when reduction algorithm was applied. The trace entities in this example are symbolically represented as A, B and C. The algorithm starts by assigning the first segment as the reference segment. The first segment (Segment1) consists of trace entries A and B. Now this segment is compared with second segment (Segment2) consisting of trace entries A, B and C. The comparison yields 'False' as the segments are not the same. Likewise the first segment starts

comparing the rest of the segments. When the first segment is compared with the 4th segment (Segment4) in this case the comparison yields 'True' because both the segment consists of the same trace entries A and B. When it is true the 4th segment is removed from the trace file. The comparison would yield 'True' for the 5th segments as well and thereby the segment being removed. The 2nd, 3rd, 6th and 7th segment when compared would yield 'False' and so these segments are not removed. The right side of the figure shows no presence of segments 4 and 5. After the first segment is finished comparing with the final segment the reference segment is now assigned to 2nd segment because that was the next immediate segment that was not removed. This process of comparison continues until the reference segment is assigned to the final non-removed segment.

The algorithm works slightly differently for the repetitive statements occurring due to nested loops. In nested loops, all the execution that are occurring due to an inner loop will always be within the scope of the outer loop execution. In this case, all the execution traces starting from "Entry of Loop_X" till the "Exit of Loop_X" of an inner loop will always reside within the "Entry of Loop_Y" and "Exit of Loop_Y" execution traces of the outer loop. The same concept applies for a nested loop of any depth. So the algorithm starts by traversing to the inner most loop's "Entry of Loop" statement and applies the reduction algorithm to the repetitive statement occurring due to the inner most loop. When the reduction is done it moves to the immediate outer loop's "Entry of Loop" statement and applies the same reduction rule. This process continues until the algorithm applies the reduction to the outer most loop.

Table 4 - 5 The result of a trace file when subjected to redundant loop reduction

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
"Entry of Loop_1"			
"Iteration Number_1 1"			
Entry	even	A	B
Exit	even	B	A
"Iteration Number_1 2"			
Entry	odd	A	B
Exit	odd	B	A
"Exit of Loop"			
Exit	func_1	A	C

The Table 4-5 shows the reduced trace file when redundant loop deduction algorithm is applies. The non-reduce trace file of the above table is shown in the Table 4-4.

4.3.2 Implementation of Filtering self-calls.

This rule removes all the messages that are sent and received by the same class. The implementation of this rule is very trivial since it just needs to compare the caller and the callee class. If classes are same then that particular execution trace entry is removed. Table 4-6 represents a sample trace file

for a code fragment. When the rule is applied to this trace file it compares the caller and callee classes of each trace entries in the table and removes the entries which have the same caller and callee class. In this case the function "func_3" is the only function that has the same caller and callee class "A". So the corresponding trace entries are removed from the table.

Table 4 - 6 A sample of a trace file to understand self-calls

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
Entry	func_3	A	A
Exit	func_3	A	A
Entry	func_2	A	B
Exit	func_2	B	A
Exit	func_1	A	C

After removing those trace entries, the reduced Table 4-7 is shown below.

Table 4 - 7 The resulting trace file after filtering self-calls

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
Exit	func_2	A	B
Exit	func_2	B	A
Exit	func_1	A	C

4.3.3 Implementation of filtering constructor calls.

The main purpose of this rule is to reduce all the message interaction statements that are occurring when an object is created. This rule removes all constructor calls. In Python these constructors are represented as "__int__" functions which would be invoked when a particular class object is created. The implementation of this rule is also very trivial since it needs to compare the Method Name of each execution trace to see if it is a "__init__" function and removes that particular execution trace when it is 'True'. Table 4-8 represents a sample trace file for a code fragment. The rule filters all the execution traces that have Method Name "_init__". In this case the second and the third entries of the Table 4-8 are removed because the Method Name column consists of __init__ function.

Table 4 - 8 A sample of a trace file to understand constructor calls

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
Entry	__init__	A	B
Exit	__init__	B	A
Entry	func_2	A	B
Exit	func_2	B	A
Exit	func_1	A	C

After removing those trace entries, the reduced Table 4-9 is shown below.

Table 4 - 9 The resulting trace file after filtering constructor calls

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_1	C	A
Entry	func_2	A	B
Exit	func_2	B	A
Exit	func_1	A	C

4.3.4 Implementation of Extraction of Unique Method calls

This rule removes all the messages that are repeatedly occurring between two classes. The implementation of this rule is by comparing each trace entry's caller details(class, file), callee details (class, file)and message details(method name, parameters) with previous each trace entries' corresponding caller, callee and message details. When respective details match then the current trace entry which is being compared is removed from the trace file. Table 4-10 represents a sample trace file. In this case, the sixth entry in the Table 4-10 is compared with each of the previous entries starting from first. Since the sixth entry is same as the second entry in the Table 4-10, during comparison the sixth entry is removed from the trace file. The resulting reduced trace file is shown in the Table 4-11.

Table 4 - 10 A sample of a trace file consisting of repetitive statements

Entry/Exit of a method	Method Name	Method parameters	Caller class	Callee class
Entry	func_1	None	D	A
Entry	func_2	None	A	B
Exit	func_2	None	B	A
Entry	func_3	None	A	B
Exit	func_3	None	B	A
Entry	func_2	None	A	B
Exit	func_2	None	B	A

Table 4 - 11 The resulting trace file after extraction of unique method calls

Entry/Exit of a method	Method Name	Method parameters	Caller class	Callee class
Entry	func_1	None	D	A
Entry	func_2	None	A	B
Exit	func_2	None	B	A
Entry	func_3	None	A	B
Exit	func_3	None	B	A

4.4 ExTraVis

This chapter discusses the tool chosen to visualize trace executions (even in the orders of hundreds of thousands). It also discusses various features incorporated by the chosen tool. There are two main criteria that were considered during the selection of the visualization tool. The first criterion is that the tool needs to visualize the entire sequence of interactions in a single view. The second criterion is that the tool needs to support the concept of information hiding. Through information hiding the identification of interesting part of the trace could be very effective thereby making it an important criterion. *ExTraVis* [8] is the visualization tool chosen which meets the above criteria and also adds additional features to support program comprehension.

ExTraVis was implemented to visualize large execution traces more effectively. The tool incorporates two views: circular bundle view and massive sequence view. The circular bundle view shows the structure of the systems consisting of hierarchical elements. This view also shows interaction between the elements and the connection between the elements are depicted as bundle splines. The view also offers the user with the feature of collapsing the elements thereby focusing on the interaction between specific parts of the system. The view also adds few other properties such as the color and thickness of the splines indicating the direction and number of calls between the elements. Another view that was implemented is the massive sequence view which outlines the entire trace execution in a compressed manner. The view mainly provides the feature of effective navigation through the trace file which helps the user in finding the interesting parts of the trace. The users are able to zoom in on particular parts of the trace which could provide a closer inspection on the specific part [8].

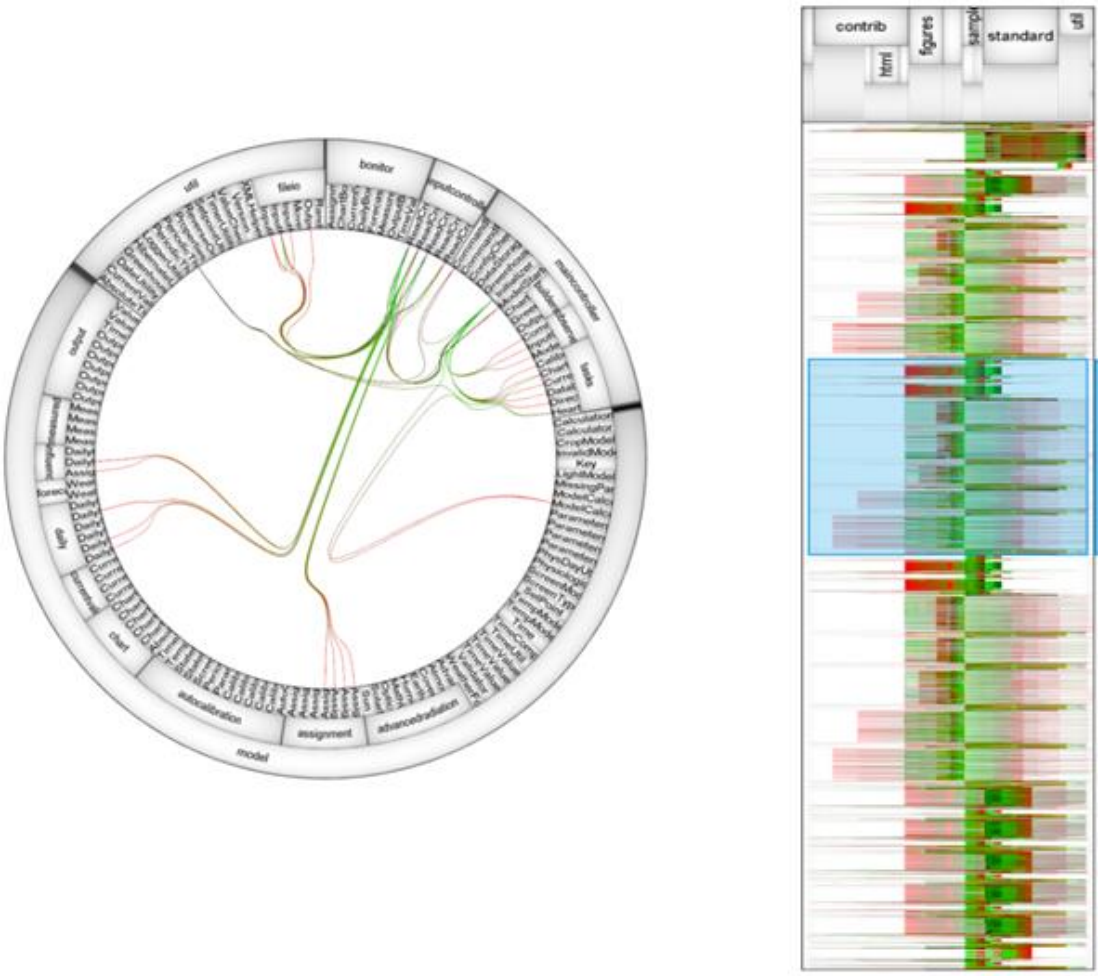

Figure 4 - 7Circular view and the massive sequence view of the ExTraVis Visualization tool (image taken from [8]).

Figure 4-7 shows an example of the circular and massive sequence view supported in *ExTraVis*. Both the views together make the tool very effective for visualization. Since the massive sequence consists of only information about the caller and callee details and excludes the method signature details there by making this view look like some form of sequence diagram. This void is filled by the circular view which provides all the details of a message interaction. A change in one of the views would directly reflect in the other view. If elements are collapsed in the circular view the same element shown in the other view would also collapse. Same way when user zooms into specific sets of traces in the massive sequence view then only the details of the selected traces are shown in the circular view. Due to these effective features, apart from the above mentioned criteria, *ExTraVis* is a more suitable visualization tool for the project.

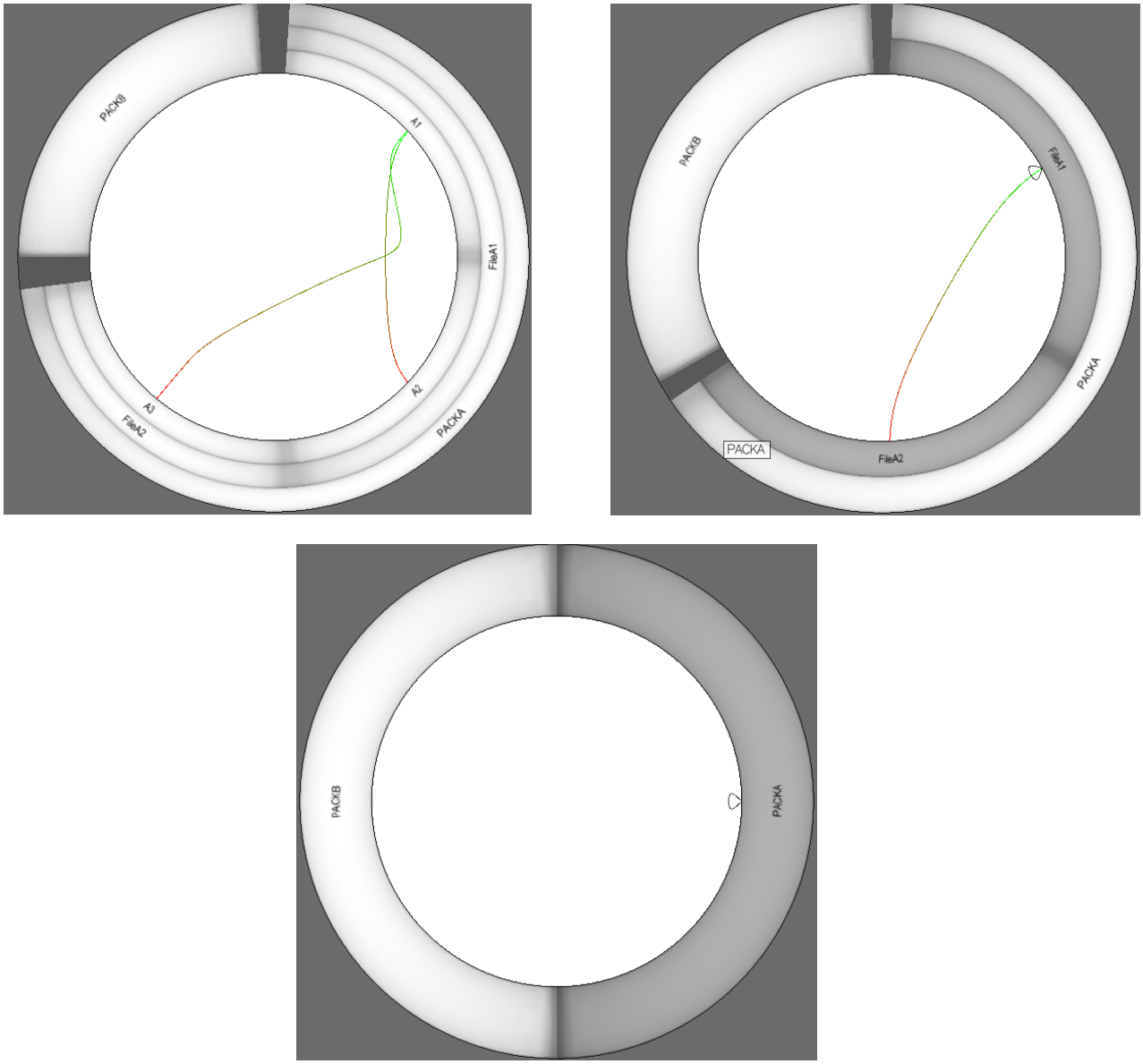

Figure 4 - 8 Various examples of an output of the circular view describing the various levels of abstraction

Figure 4-8 shows a sample of circular views in *ExTraVis* applying the feature of information hiding. The top left side of the Figure 4-8 shows message interaction occurring at class level. The top right side of the Figure 4-8 shows message interaction occurring at file level. It shows that, one of the message interacted between two classes of the same file was hiding during file level abstraction. Finally, the bottom of the Figure 4-8 shows message interaction occurring at package (directory) level. In this case, none of the message interaction was visible since all the message interaction occurred within the same directory. The significance of this figure is to show that the chosen tool supports concept of information hiding.

4.5 Implementation of extraction of hierarchical structure of a system

The information regarding the structure of the system acts as a backbone for applying the concept of information hiding. *ExTraVis* requires information regarding the hierarchical structure of the system to support the information hiding process. A Python script is written to extract this information from the application. The script takes application's root path as an input and searches for the files with extension '.py'. The search begins from the very first directory of the root path. For each file found, it reports the path of the file and filename. Next the files found are parsed to extract the names of the classes. The name of the directory can be extracted from the retrieved path of a file. The retrieved information is stored in a list which is later used during format conversion. This approach would work as long as every class can be contained only in one file.

Table 4 - 12 A sample of a list consisting of systems structure data

Directory Name	File Name	Class Names
PACKA	FileA1	A1,A2
PACKA	FileA2	A3

Table 4-12 shows an example of hierarchical structure information stored in list. Each row in table represents each element in a list. This information is directly processed during the implementation of the format conversion.

4.6 Implementation of format conversion

This section of the chapter explains in detail about the process of generating a file of a specific format which is compatible with *ExTraVis* tool. This file is generated using the information provided from trace file and the extracted hierarchical structure. The first and foremost step in the implementation of the format conversion process is to understand the structure of the *ExTraVis* input file.

4.6.1 ExTraVis Input File Format

Figure 4-9 shows an example of an *ExTraVis* input file. The input file is categorized as *ExTraVis* Header data, *ExTraVis* hierarchical structure data and *ExTraVis* call relation data. Before explaining how the trace file and hierarchical structure information is being converted to this format, each data categories are explained in detail as follows:

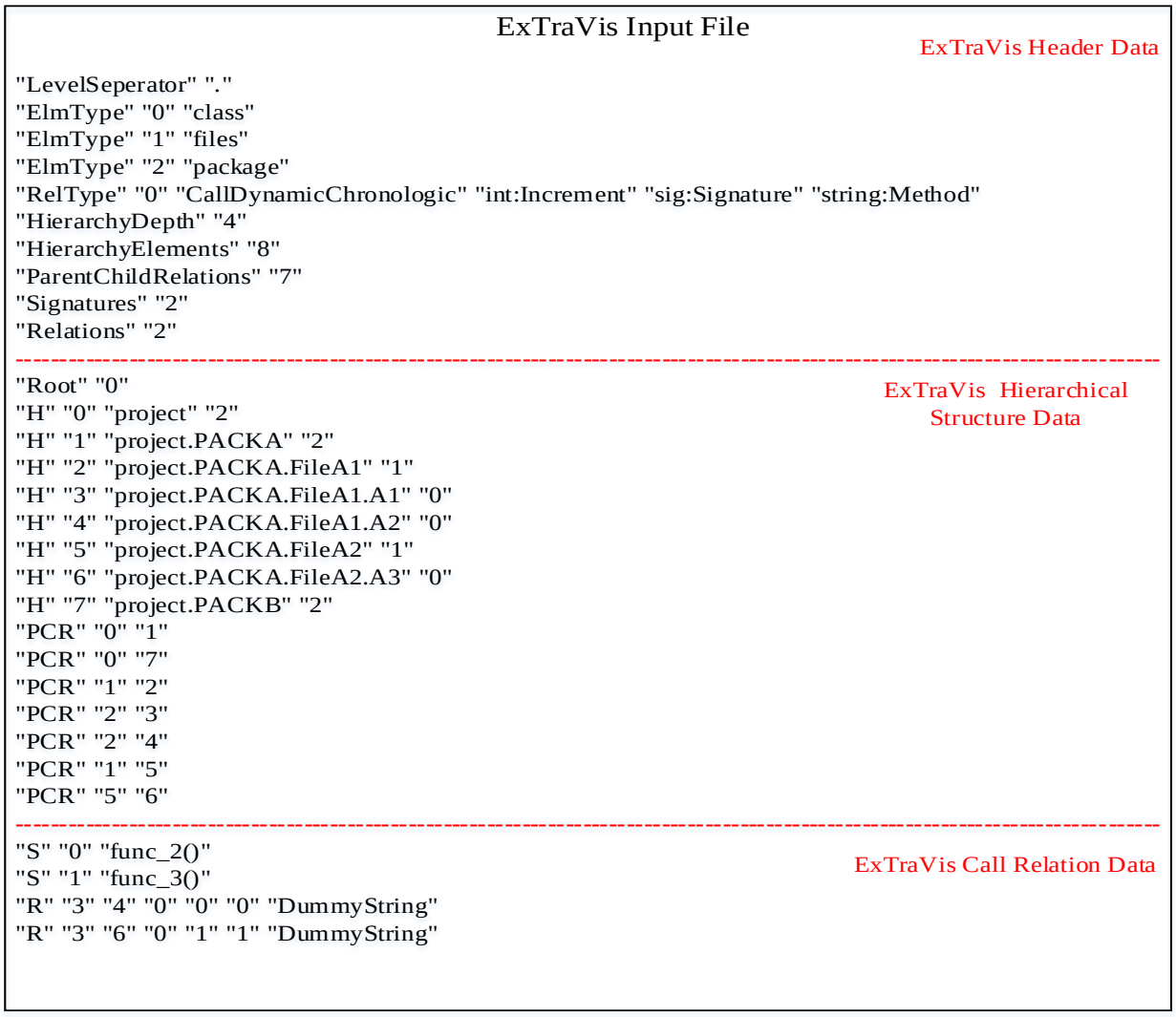

Figure 4 - 9 ExTraVis input file format

4.6.1.1 ExTraVis Header data

This is the first section of the input file consisting of the information to build the other existing data in order to visualize using *ExTraVis*. Each line in the header data section is explained in detail as follows:

```
"LevelSeparator" "."
```

This statement indicates the kind of separator used for separating levels entries later in the ExTraVis hierarchical structure data category of the file. For example, "`project.PACKA.FileA1.A1`" shown in figure represents a particular path of a hierarchical structure where the symbol "**.**" separates various levels of the path.

```
"ElmType" "0" "Class"
"ElmType" "1" "Files"
"ElmType" "2" "Package"
```

These statements indicate various classifications of levels that are defined in the application associating to a unique number (ID). For example "class" is type of a hierarchical element that is associated to a unique ID "0". These unique ID's will later on be used to classify each elements of the hierarchical structure to a particular type.

```
"RelType" "0" "CallDynamicChronologic" "int:Increment" "sig:Signature"
"string:Method"
```

This statement of the header file is to indicate the structure of the call relation statement. This statement is not important since the call relation statements are structured in the default way specified.

```
"HierarchyDepth" "4"
```

This statement represents the Maximum depth of the hierarchy. In this example the maximum depth is "4". An example of an element having depth "4" is `"project.PACKA.FileA1.A1"`.

```
"HierarchyElements" "8"
```

This statement represents the number of elements in the hierarchical tree. In this example the total number of nodes in the tree is "8".

```
"ParentChildRelations" "7"
```

This statement represents the number of parents-node relationships in the hierarchical tree. In this example the value is "7".

```
"Signatures" "2"
```

This statement represents the total number of unique message signature present in the trace file. In this case the value is "2" since only two signatures are defined in *ExTraVis* call relation data section.

```
"Relations" "2"
```

This statement represents the total number of number of execution traces (method calls). This can also be represented as the number of relations between the elements in the hierarchical tree excluding the parent child relation. The value is "2" in this case since only two relations are present which is defined in *ExTraVis* call relation data section.

4.6.1.2 ExTraVis Hierarchical Structure Data

This is the second section of the input file consisting of the information that defines each hierarchical element in the tree. The data also defines each element's parents-child relation.

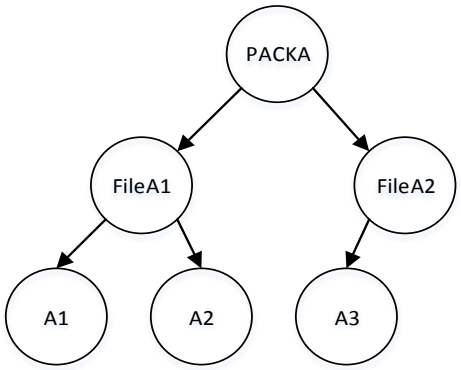

Figure 4 - 10 A hierarchical tree structures (without call relation)

The Figure 4-10 shows as sample of tree consisting of only hierarchical structure. Each level in the tree represents elements of a specific type. The Top level of the tree consists of hierarchical elements of type packages and the lowest level consists of hierarchical elements of type class. The tree shows only the parent – child relation for each element.

```
"Root" "0"
```

This section of the input file always starts with the above "root" statement. This statement explains that a hierarchical element consisting of unique ID "0" acts as the root for the entire hierarchical structure. More explanation about the hierarchical elements and its unique IDs would be explained below. The tree starts from the root element from which all the other elements are define.

```
"H" "0" "project" "2"
"H" "1" "project.PACKA" "2"
"H" "2" "project.PACKA.FileA1" "1"
"H" "3" "project.PACKA.FileA1.A1" "0"
"H" "4" "project.PACKA.FileA1.A2" "0"
"H" "5" "project.PACKA.FileA2" "1"
"H" "6" "project.PACKA.FileA2.A3" "0"
"H" "7" "project.PACKB" "2"
```

The first statement shows that unique ID'0' was defined to "project". This means that "project" is the root element. The above statements are representation of list of hierarchical elements. Each of the lines is described using 4 fields. The first field "H" represent that the statements are about defining the Hierarchical element. The second field is unique ID for the hierarchical element. These IDs should start from 0 for the first element and increment numerically for each element defined. The third field is the name of the element and it should be defined using the full path name. For example, if an application "project" consists of a package "PACKA" within which there is a file "FileA1" consisting of a class "A1". Then the name of the element "A1" in defines as "project.PACKA.FileA1.A1". The symbol "." separates various levels in the name. The final field is a unique integer value that defines the type of the hierarchical element. These unique integers are elements type Ids which was defined earlier in the ExTraVis header data to indicate whether the element- in this case- either class or file or package.

```
"PCR" "0" "1"
"PCR" "0" "7"
"PCR" "1" "2"
"PCR" "2" "3"
"PCR" "2" "4"
"PCR" "1" "5"
"PCR" "5" "6"
```

The above statements are lists of parent-child relations for the above defined elements. These statements together with the hierarchical elements defined above form the hierarchical tree structure shown in Figure 4-10. The statements are described using three fields. The first field "PCR represents that the statements are parent child relations. The second and the third fields are unique IDs of a 2 hierarchical elements. Where the first element represents the parent and the second element represents a child. These unique IDs are obtained from the list of hierarchical elements defined above. For example "PACKA" is the parent of "FileA1". The unique ID number defined for the element "PACKA" is "1" and for "FileA1" is "2". So the parent-child relation statement would be written as `"PCR" "1" "2"`.

4.6.1.3 ExTraVis Call Relation Data

This is the final section of the input file consisting of the information of the method calls that are executed during runtime. This data together with the above *ExTraVis* hierarchical structure data forms the complete tree (includes the hierarchical tree structure with calls among the tree elements).

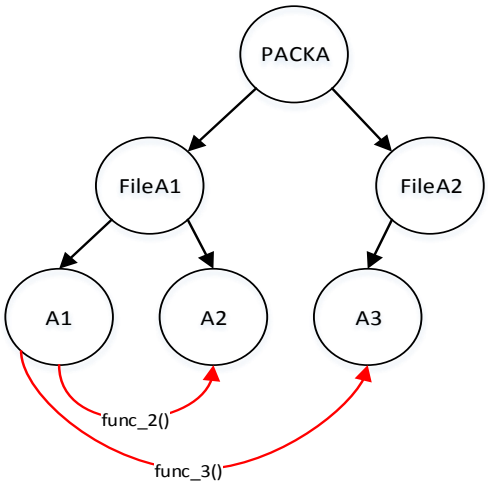

Figure 4 - 11 A hierarchical structure tree (with call relation)

The Figure 4-11 shows a sample of tree consisting of both hierarchical structure as well as its call relation between elements. The tree shows the parent – child relation and the call relation for each element.

```
"S" "0" "func_2()"
"S" "1" "func_3()"
```

These statements represent a list of unique function (signatures) that was executed during runtime. Each of the lines is described using 3 fields. The first field "S" represents that these statements are about defining unique functions. The second field is a unique ID for a function defined. These IDs should start from 0 for the first function and increment numerically for each unique function defined. The third field is the name of the unique function.

```
"R" "3" "4" "0" "0" "0" "DummyString"
"R" "3" "6" "0" "1" "1" "DummyString"
```

These statements represent the call relations between hierarchy elements. Each of the lines is described using 7 fields. The first field "R" represents that these statements are about call relation between elements. The second and the third field represent two unique IDs of a two hierarchical elements. Where the first element represents a caller and the second represents the callee. These unique IDs are obtained from the list of hierarchical elements mentioned above. The fourth field is an integer value to represent the type of call relation we have defined earlier in header data. In this case it is "0" in all the places since same relation type was used everywhere in the data set. The fifth field represents a unique ID for each relation statements defined. These IDs should start from 0 for the first call relation statement and increment numerically for each relation statement. The sixth field is an integer values representing a unique ID of a function name. The unique IDs are obtained from the list of unique signatures mention above. The final field represents a string value which is not of much importance so a dummy value given.

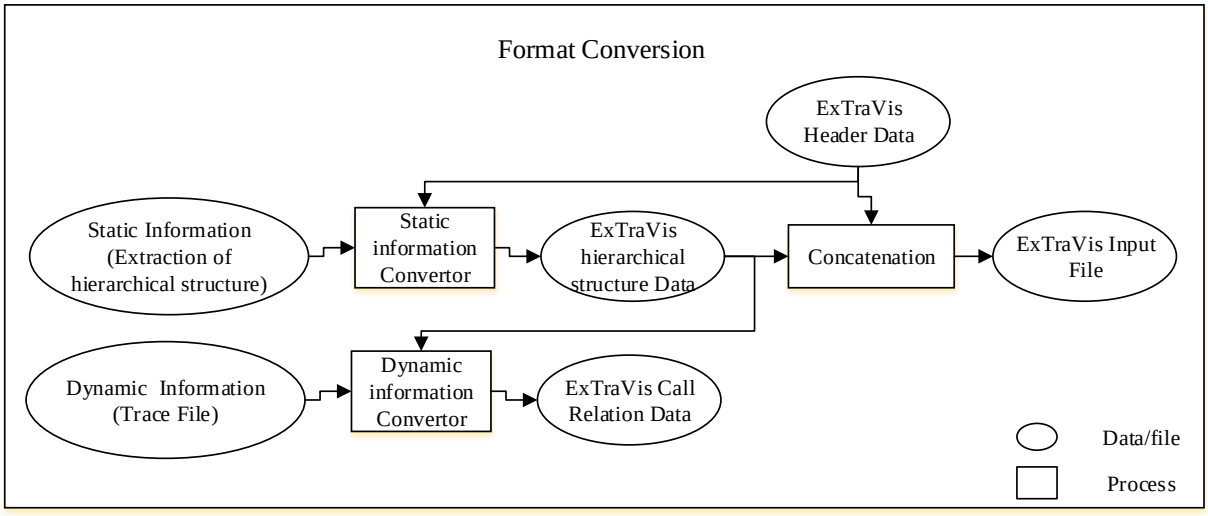

Figure 4 - 12 An internal structure of the format conversion process

Based on the file format a conversion process transforms the exiting hierarchical structure information and the trace file information into the *ExTraVis* file format. Figure 4-12 shows an

implementation process of the proposed format conversion step. There are three main processes that are involved in the conversion step namely Static information conversion, dynamic information conversion and concatenation. Before the implementation of the various processes an ExTraVis header data needs to be created, since some of the information in the header data would be used in the Static information conversion process. The information regarding the number of hierarchical elements, number of unique signature, number of parent child relation and number of call relation can be determined from the already existing hierarchical structure data (List) and from the trace file. Using this information a Python program is written which automatically creates a *ExTraVis* header data file consisting of the above information. The working and implementation of each of these processes are explained in detail as follows

4.6.2 Static information conversion

The Figure 4-12 shows that the process takes information from extraction of hierarchical structure of a system as well as ExTraVis header data and converts it into ExTraVis hierarchical structure data. The format of the ExTraVis hierarchical structure data is already discussed in the Section 4.6.1.2. The entire hierarchical structure of the system is stored in a list the process is explained in the Section 4.5. A python program is written that converts each element in the list into ExTraVis hierarchical structure data. Figure 4-13 shows the conversion process from an element in the list to the specified format.

Directory Name	File Name	Class Names
PACKA	FileA1	A1,A2

```
"H" "0" "project" "2"
"H" "1" "project.PACKA" "2"
"H" "2" "project.PACKA.FileA1" "1"
"H" "3" "project.PACKA.FileA1.A1" "0"
"H" "4" "project.PACKA.FileA1.A2" "0"

"PCR" "0" "1"
"PCR" "1" "2"
"PCR" "2" "3"
"PCR" "2" "4"
```

Figure 4 - 13 Conversion of array elements into ExTraVis hierarchical structure data format

The top part of the Figure 4-13 is a representation of an element in the list. The representation shows that a package “PACKA” has a file “FILEA1” which consisting of classes “A1” and “A2”. The bottom part of the Figure 4-13 is the transformed ExTraVis hierarchical structure data for the array

element. For the conversion process the Python program identifies first part of the list element as a package “PACKA”. Since “PACKA” is of type packages (directory) the unique ID for this element type is retrieved from the ExtraVis header data in this case its “2”. Then the program automatically writes this information into a form of ExTraVis format in this case "H" "1" "project.PACKA" "2". Still the parent child relation statement for the same element needs to be written. The package “PACKA” resides under the root “project”. The unique IDs for element “PACKA” and element “project” are “1” and “0”, respectively. The IDs can be retrieved for the list of already defined hierarchical elements. Using the retrieved element IDs the parent child relation statement "PCR" "0" "1" is automatically written by the program. The same conversion happens for the rest of the parts of the list element which is a file and its corresponding class elements. Once the conversion process completes for one element, it moves on to the next element in the list. This process continues until the end of the list. Finally, the program transforms the entire hierarchical structure into ExTraVis format and stores in a file “ExTraVis hierarchical structure data” file.

4.6.3 Dynamic information conversion

The Figure 4-12 shows that the process takes information from trace file data and converts it into ExTraVis call relation data. The process also takes the output of the Static information conversion as an input. The reason is because to transform a trace file into a call relation data the information about the list of hierarchy elements are need. The format of the ExTraVis call relation data is already discussed in the Chapter 4.6.1.3. A Python program is written that converts each entry in the trace file into ExTraVis call relation data. The Figure 4-14 shows the conversion process from trace entries to the specified format

Entry/Exit of a method	Method Name	Caller class	Callee class
Entry	func_2	A1	A2
Entry	func_3	A1	A3

```
"S" "0" "func_2()"
"S" "1" "func_3()"
"R" "3" "4" "0" "0" "0" "DummyString"
"R" "3" "4" "0" "1" "1" "DummyString"
```

Figure 4 - 14 Conversion of trace file entries into ExTraVis call relation data format

The ExTraVis call relation data consists of two types of statements. The first statement represents the list of unique function. These statements start with “S” in the call relation data. A Python program is written which takes the trace file as input and retrieves all the unique functions that were executed in the trace file. For example, if the Python program takes the sample trace file shows in the Figure 4-

14 as input then the output of the program would be func_2 and func_3 as the unique sets of function present in the trace file. After retrieval of the unique sets of functions the program converts all the functions into lists of the format explained in ExTraVis call relation data. In this case, the retrieved functions are “func_2” and “func_3” which is converted into `"S" "0" "func_2()"` and `"S" "1" "func_3()"` statements respectively. Once the list of unique functions is created, the program continues with the conversion of the second type of statement in *ExTraVis* call relation data.

The second statement represents the call relation between hierarchy elements. These statements start with “R” in the call relation data. For each entry in the trace file a corresponding call relation conversion needs to be done. For conversion of “R” statements, three sets of unique IDs are required. Two IDs represents the caller and callee element and other ID represent the unique function called. The unique ID’s of caller and callee elements are retrieved by searching for these elements name in the list of hierarchy element defined earlier. Once these elements are matched in the list their corresponding unique IDs would be retrieved. For example, the first trace entry in the Figure 4-14 has caller and callee class as “A1” and “A2”. The program now matches the presence of “A1” and “A2” in the list of hierarchy element and retrieves the assigned unique ID for those two elements. In this case, the IDs are “3” and “4” respectively. Apart from these element IDs the “R” statements requires unique signature ID for the function in the trace entry. So now the program is extended by searching for the match for the same function defined in the list of functions defined earlier. In the same example considered above “func_2” was the function that was called. So the program match’s the presence of “func_2” in the list of function and if a match is found it retrieves the unique ID. In this case the ID is “0”. Once all the required IDs are retrieved the program converts the trace entry into a call relation format. For the first trace file entry the converted call relation format is `"R" "3" "4" "0" "0" "0" "DummyString"`. The program continues to convert all the trace entries by same approach explained above.

4.6.4 Concatenation

This is a very trivial process where after the generation of ExTraVis header data, ExTraVis hierarchical structure data, ExTraVis call relation data, they are integrated together to make one whole ExTraVis input file. The order of concatenation is important where header data comes first followed by hierarchical structure data and finally the call relation data. A python script was written to achieve this process.

Chapter 5: Evaluation of the tool

In this chapter, the results obtained by applying the tool to the DiagnosticFramework application are discussed. Initially, the description of the application and a specific use case used for evaluation are briefly discussed. Later on, results generated during various stages of the tool are validated.

The application has been already introduced in Section 3.3. The application consists of 200 files with 189 classes. Executing the application presents the user a GUI with 3 different views Extraction, Processing and Visualization. The application during start up loads all the data present in a CSV file. When a GUI is presented, the user can select few fields like sensor, paper handling, etc. from a large list such that only the details of the selected fields are extracted from the loaded data and this can be viewed in the extraction view. Later on, the extracted data can be represented in various diagrams such as pie chart, line chart etc. present in the visualization view. The process above explained is a basic working of the DiagnosticFramework application.

The evaluation of the tool is done using a specific use case scenario of the application. The use case is "*load CSV"* where a user loads a separate CSV file in the visualization view by clicking the button *"load".* All the actions (clicking button, loading a life) the user did with respective to the use case were performed on the visualization view.

5.1 Evaluation of Source code transformation process

The automatic tool was successfully able to instrument all 1904 functions and 134 loops distributed over 200 files. The trace that results from executing the specific "load CSV" use case scenario contains 126,490 function calls. Among the traces a significant part of the trace entries are generated due to the application's "startup" use case. To understand the "load CSV" use case it is very essential to differentiate the trace due to "startup" use case and that of the "load CSV" use case. The details of the traces are already discussed in the Section 4.2.1. Regarding the performance of the instrumentation application, it runs roughly 1.5 times slower than the execution of the original application. This small lag is mostly seen during the startup of the application and not much during the user interaction with the application. To conclude, the performance of the instrumented application was only reduced by a very small amount which is one of the positives of the tool.

5.2Evaluation of Trace file analyzer

This section of the chapter discusses the efficiency of the reduction rules when applied to the trace file generated for the *"load CSV"* use case. The performances of the reduction rules were calculated by initially applying each rule individually and then applying all the rules together. The maximum reduction in trace entries was around 89 % when applied with all the reduction rules. The Figure 5-1 shows a column chart diagram representing the percentage in reduction of the trace file when subjected to the rules. An important remark that was observed during the evaluation process is that

the reduction due to loops alone reduces a significant part of the trace file (around 85%). Due to this, it gives the user more flexibility in terms of filtering self or constructor calls. The performance of *extraction of unique method calls* rule is not analyzed during evaluation of the trace file analyzer. This is a specific reduction rule which needs to be applied very carefully. Since it retains only the unique message interacted between classes, there is a possibility that when the rule is applied to a large number of traces some interesting message interactions could be removed. A trace file in general contains sets of message interactions pertaining to different execution scenarios of an application. There is a possibility that some of the messages interacted in a scenario can be the same with respective to another. So when the *extraction of unique method calls* rule is applied those common messages interacted would be retained only in one of the scenarios. This could lead to lower understanding the scenario behavior.

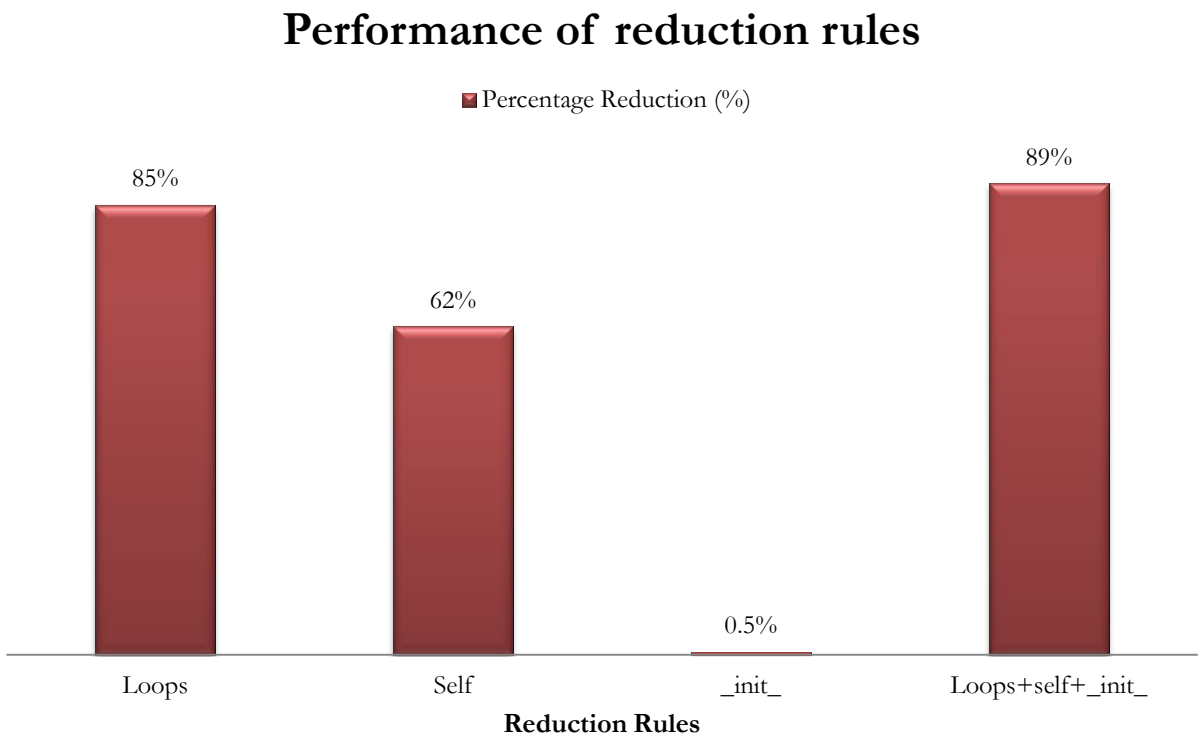

Figure 5 - 1 The performance of reduction rules when applied to a trace file.

5.3 Evaluation of Format conversion

A reduced trace file consisting of 19245 function calls was used as a dynamic input file for the format conversion process. To understand and validate *"load CSV"* use case the self-call and the constructors call statements are preserved in the trace file thereby only the repetitive statements due to loops being removed. The trace file is converted into an *.rsf* file and fed into ExTraVis.

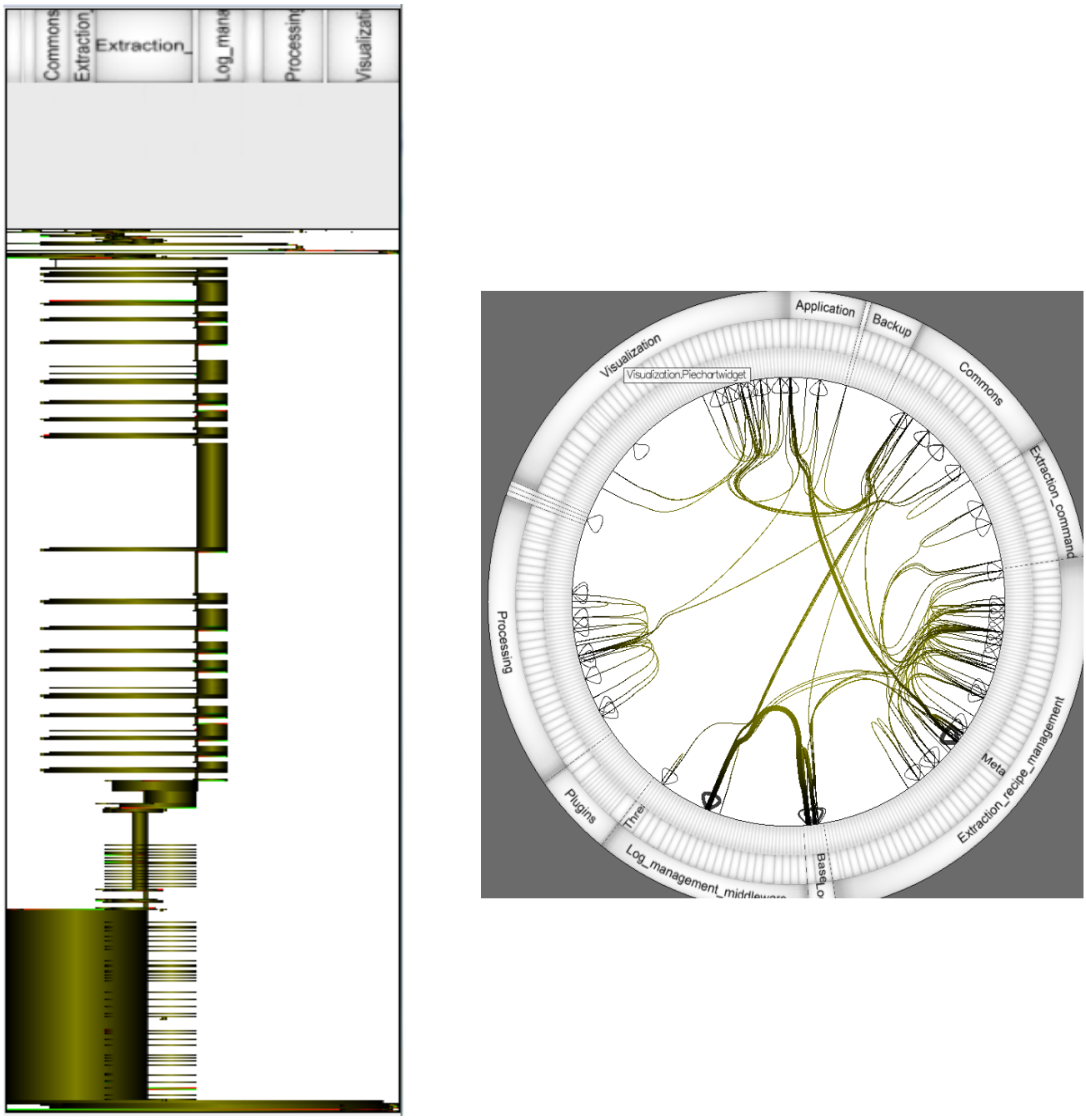

Figure 5 - 2 Left: Output of a massive sequence view for the use case "load CSV" and "startup" and right: Output of a circular view for the use case "load CSV" and "startup".

5.4 Validation of the Result

The main purpose of the tool is to support program comprehension. This section of the chapter discusses how the user is able to understand the *"load CSV"* use case of the application using the

final results of the tool. Few queries are formulated by the user based on the use case which needs to be validated using the final result. The following are the queries raised by the user:

- Was there any threads executed for this use case scenario? If so, can we know when a thread starts and ends?
- What all classes share the data object for the loaded csv file?
- How is the dataflow through the classes?

First step in validation is to identify the section of the traces that was generated due to the *"load CSV"* use case. The Left hand side of the Figure 5-3 shows an output of the massive sequence view for the entire trace consisting of "startup" and "load CSV" use case. All the action of the *"load CSV"* use case was executed in the visualization view of the application. This gives an important remark that the user is interested in set of message interactions that involve only the visualization package. The top part of a massive sequence view shows the structure of the system along the horizontal axis. Below this, is the visualization of call relations which are ordered along the vertical axis based on the time of execution from the earliest to the latest call being executed. The direction of the call relation is color coded where the green is the caller and the red is the callee.

One of the important features of massive sequence view is that it provides visualization of interaction patterns. Concentrating on specific interaction patterns aids the user in achieving abstraction and focusing on low-level system design with fewer traces. The user is interested in all the interaction with respective to the visualization package and so upon close observation the block of interaction pattern that is highlighted in the red box shown in the left hand side of the Figure 5-3 consists of call relations that are related to the visualization package. The right hand side of the Figure 5-3 shows a zoomed view of the selected interaction pattern. This pattern consists of 295 calls which were abstracted from 19245 calls.

The disadvantage of the massive sequence view is that it only shows few details of a message interaction. The call relation excludes the method signature details (method name, parameters). Some parts of the caller and callee details (like classes and files) are not clearly visible while viewing in massive sequence view. Missing details can be viewed using the circular view. The circular view provides visualization of all the details of a call relation and along with system's structural elements and their interrelationships. The disadvantage of the circular view is that the order of call execution with respective to the time is not visualized. The collaboration of both the views is required in order to view the details of a call relation and its occurrence in time unlike in a UML sequence diagram where you can view both in a single diagram.

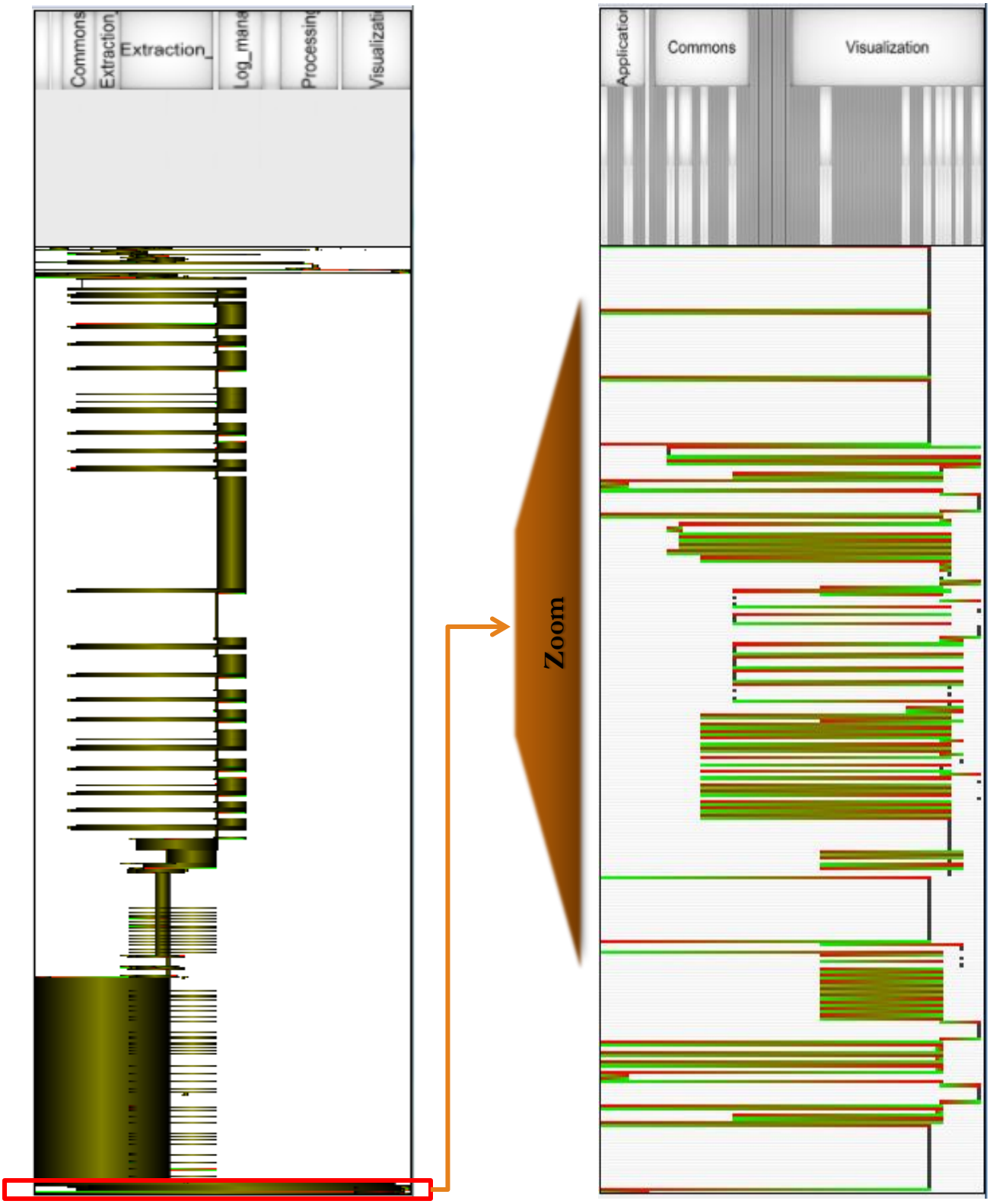

Figure 5 - 3 Left: Output of a massive sequence view for the "load CSV" and "startup" use case and Right: Zooming in on the highlighted part of the trace representing the "load CSV" use case.

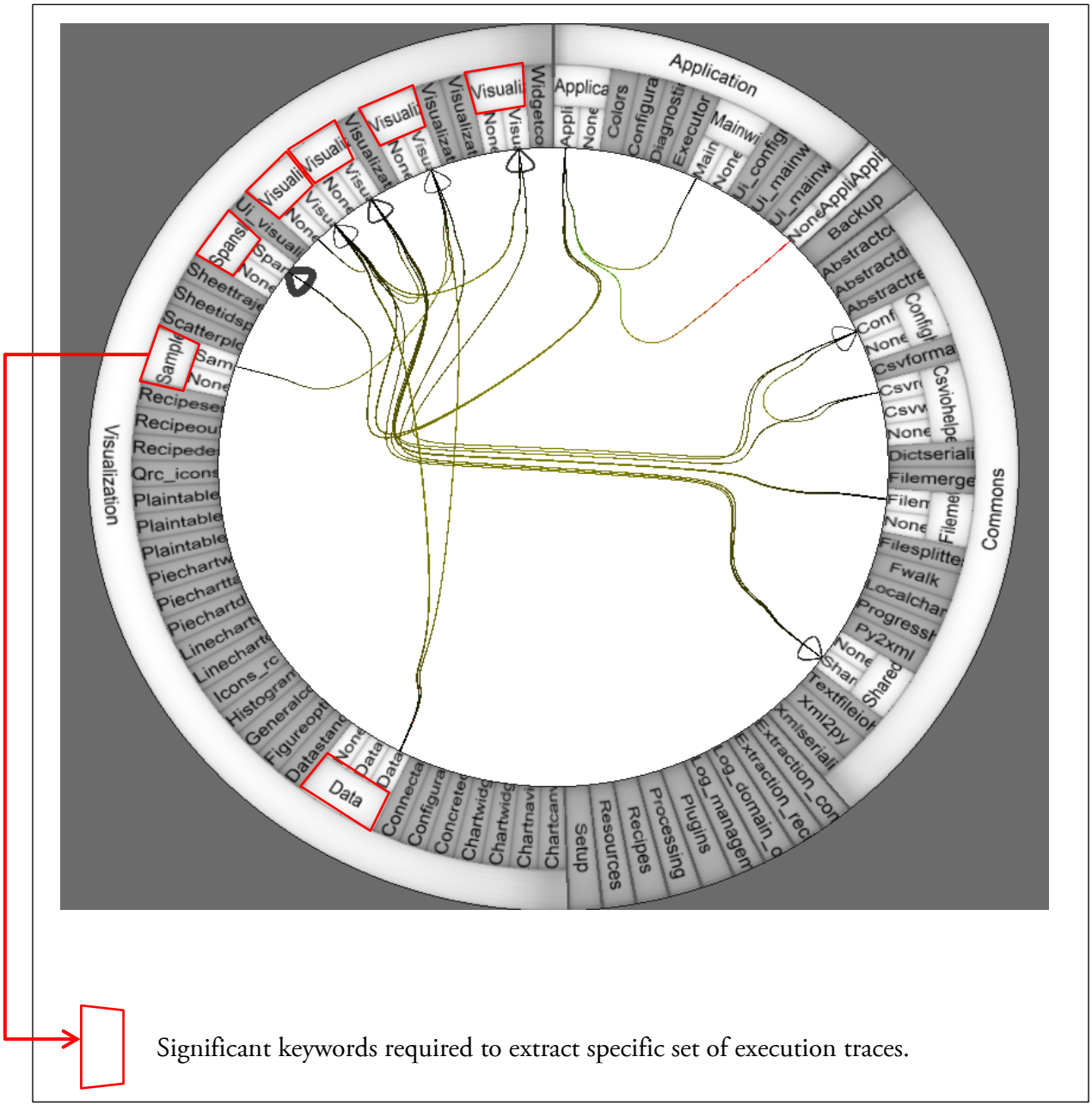

Figure 5 - 4 Output of a circular view for the zoomed part of the trace with significant keywords highlighted

In Section 3.3 a conclusion was made that it is not feasible to view large number of execution trace using a sequence diagram. This lead to the search for suitable visualization tool called ExTraVis that can visualize the entire trace in a single view. Now by using the massive sequence view the user is able to focus on specific parts of the trace consisting of just around 300 call statements. Viewing this amount of call statement using a sequence diagram is easily feasible so it helps the user understand the behavior of the use case. The only challenge remaining is to extract only these specific call statements from the entire reduced trace file. This can be achieved using the circular view in ExTraVis. The circular view always visualizes only the selected part of the call statements in the

massive sequence view. Here, the selected part contains all the call relations pertaining to visualization package.

Figure 5-4 shows an output of a circular view for the zoomed part of the massive sequence view. Using this view, significant keywords can be identified that will assist in retrieving the specific execution trace from the trace file. The Figure 5-4 shows red colored boxes highlighting the significant keywords. These keywords can be a file name, a class name or a directory name. For this use case, all the file names present in the visualization package that are involved in the interaction pattern are chosen as the keywords. A small Python program is written to retrieve those interesting parts of the trace selected by the user from the already existing reduced trace file using these keywords. The program retrieves only those trace entries which have either caller detail or callee detail that matches any of the identified keywords. Finally a separate trace file consisting of those approximate 300 call relation is generated. The trace file is further reduced by using the *extraction of unique method calls* reduction rule. This rule removes all the repetitive statements apart from the loops and finally generates a trace file around 130 trace entries. So the final reduced trace file is now converted into an MSc-generator file format which was introduced and explained in Section 3.3. The converted file is now loaded into Msc-generator and a sequence diagram is generated. Figure 5-5 shows the generated sequence diagram in MSc generator for the use case *"load CSV".* Some of the message interactions were removed from the sequence diagram manually so that the diagram focuses on details that support program comprehension by understanding and justifying the queries formulated by the user. The list of questions is addressed using the generated sequence diagram as follows:

Was there any threads executed for this use case scenario? If so, can we know when a thread starts and ends?

There are two classes "*VisualizationReaderThread"* and *"VisulaizationFileDataLoader"* which are the threads that were executed in this use case. The "*VisualizationReaderThread"* class itself contains the word "Thread" thereby making it easy to identify. Other than the naming the function *"start()"* invoked by the class "*VisualizationController"* is always executed whenever a thread is being initiated. From the sequence diagram it can be explicitly seen that the "*VisualizationController"* invokes *"start()"* whenever the two classes "*VisualizationReaderThread"* and *"VisulaizationFileDataLoader"* is being initiated. The duration of the execution of the method *"run()"* occurring as a self-call in the classes "*VisualizationReaderThread"* and *"VisulaizationFileDataLoader"* represents the duration of the thread execution. It can be seen from the sequence diagram that in the lifetime of the two threads ("*VisualizationReaderThread", "VisulaizationFileDataLoader")* there is a start of the *"run()"* function and after few interaction there is end of the *"run()"* function.

What all classes share the data object for the loaded csv file?

User always has the flexibility regarding the amount of details required to generate a sequence diagram. The choices depend upon the type of use case that needs to be comprehended. In this case the inclusion of self, constructor calls *("__init__ ()"*) and parameters passed during method invocation are the choices. Parameters (*sharedDataObject, shared_object*) are tracked in the sequence diagram to have a better understanding of how the data object being shared. The data loaded during the execution of the use case is reference to a data object called *"sharedDataObject"*. The first occurrence of this parameter is when class *"ApplicationController"* invokes a method *"__init__()"* of class *"VisualizationControler"* with one of the parameter passed *("sharedDataObject")*. This shows that class *"ApplicationController"* initially had the referenced data object and during the method invocation the data object is now shared with the class *"VisualizationControler".* Same as above class *"VisualizationControler"* now invokes the function *"__init__()"* and shares the data object by passing as a parameter to the classes *"VisulaizationFileDataLoader"* and " *VisualizationReaderThread".*

Apart from addressing the queries, the current tool also supports in improving the source code of the application. To answer the third question, a new and improved sequence diagram needs to be generated using the improved source code. A specific method call *"readlineInBuffer()"* highlighted using a box shown in the Figure 5-5 was included during the improvement process of the source code. Initially the *"readlineInBuffer()"* method call was not present in the trace file. While understanding the behavior of the class *"VisualizationFileDataLoader"* there was only one function being executed which is *"_clear_extraction_buffer()".* This function removes data from a buffer. Further analysis showed that there is no other function executed that adds data to the buffer. This led to a manual inspection of the source code and it was found that the code was badly written. The process of adding data into the buffer was done using few variables rather than invoking a function that does the same. The code was later improved by writing a separate function *"readlineInBuffer()"* that does the above operation.

How is the dataflow through the classes?

The inclusion of *"readlinesInBuffer()"* function into the source code assists in answering the above question. The *"readlinesInBuffer()"* function represents that the data is now read from a *csv* file and stored in a buffer through the class *"VisulaizationFiledataLoader"*. The retrieved data is referenced to the data object of class *"VisulaizationFiledataLoader"* which is also shared with the class *"VisulaizationReaderThread"* using the *sharedDataObject*. Later on, *"VisulaizationReaderThread"* class load the data into the class "Data" by invoking a function *"loadData()"*. Finally the class "Visualization controller gets the data from class "Data" by invoking the function *"getData()".*

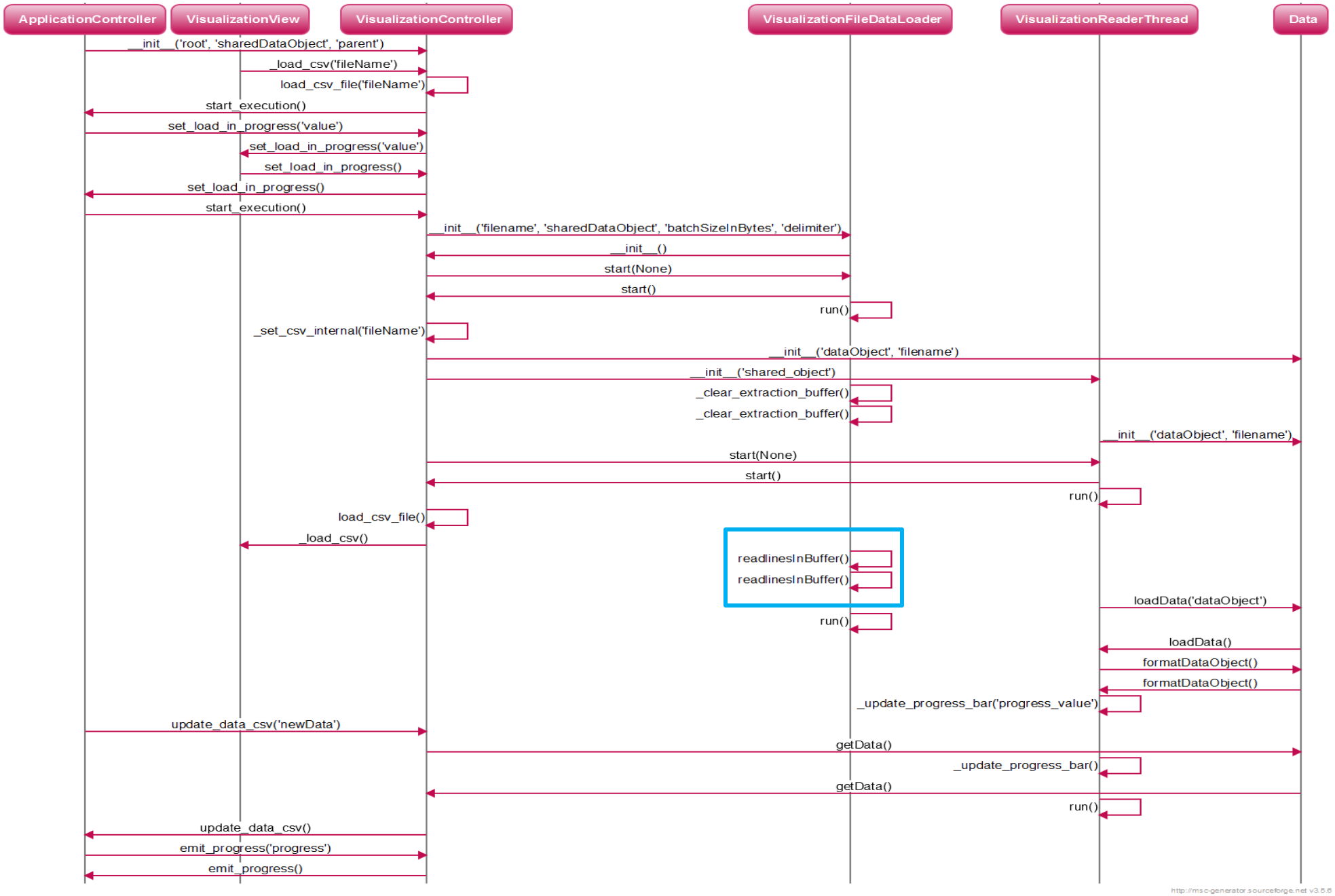

Figure 5 - 5 A Sequence diagram representing the reduced trace file generated using Msc-generator.

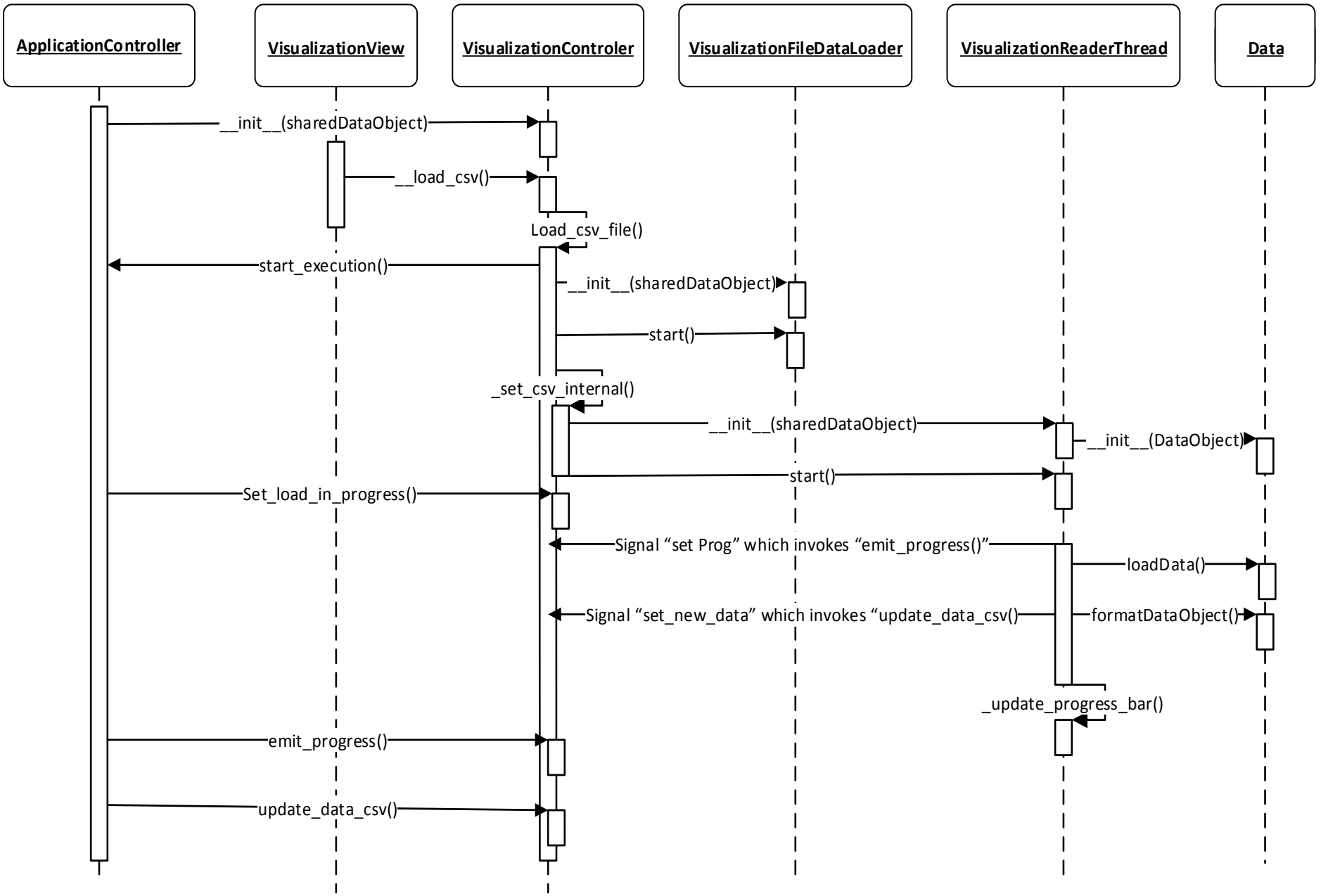

Figure 5 - 6 A Sequence diagram manually sketched by the user for the “Load CSV” use case.

A similar sequence diagram as the one in Figure 5-5 was sketched manually by the user of the application through source code inspection as in Figure 5-6. The manual process took around 16 hours to generate the sequence diagram whereas the designed automatic tool took around 1 hour with the inclusion of improvement to the source code. All the function that was sketched manually in Figure 5-6 can be mapped onto the generated sequence diagram in Figure 5-5. The only difference is that there is an additional detail "signal" in the sketched sequence diagram. In a GUI application it is common to have signals that are used for communication between objects. The basic operation of a signal is that it invokes set of functions that were already connected to that particular signal. The absence of the signals in the generated sequence diagram is acceptable since was not defined in the requirement in Section 3.1. This shows that the tool produces consistent result.

As a result of the evaluation of the tool there is a significant remark that can be made. It was found that using the result from the ExTraVis tool was difficult to facilitate program comprehension. So by combining the features of Msc-generator with ExTraVis, it was possible to achieve the goal. The challenging part was to extract the set of traces from the trace file that was selected by the user in the ExTraVis. To make this simpler, the following possibilities can be adopted:

- Implement a user interface button in ExTraVis that automatically retrieves the selected trace in massive sequence view instead of identifying significant keywords from circular view and then using them to retrieve the traces. Since ExTraVis software was not an open source, any customization to the software was not feasible.
- To identify software that can be able to cope with large amounts of traces. Also provides a sequence diagram as a separate view for the selected set of trace.
- To identify software similar to ExTraVis (which is an open source) and later on extending the software by customizing it.

The feedback of the user regarding the tool is that it is not too user friendly. The reasons are listed as follows:

- The current tool does not have the capability of reversing the instrumentation process. Once an application is instrumented, the inserted code always resides in that application. So each time a copy of the original source code is instrumented instead of the original source code.
- A "One-click" process is still not yet adapted to the current tool. Each analysis is implemented separately, so the user needs to execute each analysis (instrumentation, reduction rules and format conversion) each time separately.

Apart from the user friendliness the current tool is a prototype that just implements various proofs of concepts. The current tool still needs to be improved in order to make it practically useable. This can be achieved by validating the tool with many use case scenarios of the application. Since validating

with different use case scenarios can result in different behaviors of the tool. This can support the tool's enhancement if any irregularities were found. For example the redundant loop reduction algorithm in particular performs slower when the amount of trace increases. So there is a need for optimization for the algorithm.

Chapter 6: Conclusion

In this chapter, the contributions of the project and answers to the three design questions are summarized.

6.1 Contribution

In order to facilitate program comprehension, an approach was proposed to reverse engineer a sequence diagram from execution traces for a Python application (DiagnosticFramework). This approach is entirely based on dynamic analysis of the source code. In this project, a tool is implemented which automatically transforms a given application using an *instrumentation* process. During execution of the transformed application, information regarding method executions is collected and stored as a trace file. The main issue with dynamic analysis is that large amount of traces are collected during the execution of the program. To solve this, two approaches were proposed: Applying reduction rules, choosing effective trace visualization software (ExTraVis). The first approach was achieved by implementing four reduction rules which when applied reduces the amounts of execution traces in the trace file. The second approach is different from the reduction rules since none of the information in the trace file are reduced instead a visualization software is chosen that can compactly visualizes the large amounts of traces. ExTraVis being the chosen tool is able to support visualization of large amount of traces. The two approaches are implemented in such a way that the user has flexibility over the amounts of data that needs to be visualized. The user has the option of selecting various reduction rules that needs to be applied. Finally, the tool converts the trace file into a file format compatible with ExTraVis.

The implemented tool was evaluated using *"load CSV"* use case of the DiagnosticFramework application. The tool was successfully able to instrument all the "1904" functions of the application and also able to generate a trace file consisting of "126490" calls. Based on the feedback of the user, the trace file was subjected to only redundant loop reduction rule and was reduce to "19245" calls. The file was visualized in ExTraVis and using the massive sequence view the user was able to select the interesting parts of the trace. Later on, a sequence diagram is visualized using Msc-generator software for the selected part of the trace. Few questions were framed by the user in Section 5.4, were answered using the generated sequence diagram. Apart from the questions answered, improvement was also made in the original source code of the application based on the understanding of the behavior of the system using the sequence diagram. The precision of the generated sequence diagram was also verified using a manually sketch diagram by a user through manual code inspection. It showed that the generated diagram was in fact same as the manually sketched one.

6.2 Design Questions Answered

This thesis attempts to answer the following three design questions:

What are the approaches implemented in this project to effectively visualize large amounts of information retrieved during dynamic analysis of the source code?

The different reduction rules designed in trace file analyzer is one of the approaches taken to aid visualization of large amounts of traces. These rules help in reducing the amounts of traces in the trace file and thereby increasing the clarity of the sequence diagram. But obtaining the desired clarity to visualize a readable sequence diagram depends upon the effectiveness of these rules. Choosing suitable software that can support visualization of large amounts of trace is another approach used in this project and also an answer to the first design question. ExTraVis being the software can visualize traces up to the order of millions. This approach independently can provide a precise sequence diagram without any loss of information or details in the behavior.

If more than one approach is used to effectively visualize large amounts of information, how can it be best combined to achieve the desired goal?

The follow from the implementation of reduction rules in Section 4.3.1 to the implementation of format conversion Section 4.6 shows the integration of two approaches that is designed to aid visualization of large amounts of traces. The important remark is that trace file analyzer is a step newly included into the original design architecture. This means that even without the implementation of reduction rules the chosen visualization tool (ExTraVis) is enough to support visualization of large amounts of traces. But the important goal is to support program comprehension for which these reduction rules play an important role. The reduction rules when applied increases the productivity in helping the user focus on the interesting part of the traces.

How does sequence diagram representing a system behavior facilitate program comprehension?

During the validation of the results a conclusion was made that UML sequence diagram facilitates program understanding much better than a circular view, a massive sequence view or when combined. The important feature of a sequence diagram is that it shows message interaction between classes with respect to the time of occurrence. This feature enables to understand the lifetime of a function being executed. In this case, the lifetime of the function *"run()"* in Figure 5.6 helped in understanding the lifetime of a thread being executed as well as the all the interesting message interacted by the thread. Apart from this, a sequence diagram help in understanding the behavior of each classes individually. A user can focus on all the interaction pertaining to a particular class for understanding. In this case, understanding the class *"VisualizationFileDataLoader"* aided in identifying a badly written code and improving it by defining a function *"readlinesInBuffer()"*. This led to the understanding of the flow of data in the system. If it can help identify a badly written code then it can also help in identifying any bugs in the code. This shows that a sequence diagram always aids in understanding the behavior of the program.

Chapter 7: Future work

The project presents various prospects for the future work. The instrumentation approach used in this project can be improved so as to get additional information during runtime. The additional information that needs to be included are listed as follows

- The details regarding *"signals"* that are used in GUI Python application can be included. The importance of "signals" is already explained at the end of Section 5.4.
- The details about local variables names which are initialized during an object instantiation can also be included. Since using this information it could facilitate the understanding of the system as the mapping between the sequence diagrams and the source code would be more direct.
- Third details about destructors in Python application can be included. The reason is in Python, objects are automatically destroyed even without the invocation of the destructor function. So the details of the object destruction can also be represented in a sequence diagram.

At present only dynamic analysis is used as an approach to understand program execution during runtime. This approach does not compute all the possible executions in a program. In the future, the combination of both dynamic and static analysis can be used to enumerate all possible execution and thereby ensuring completeness of the sequence diagram. Furthermore, the role of threads during program execution needs to be investigated. An effective technique needs to be implemented to extract details of specific threads and their interactions and finally visualize them. Finally, a detailed research can be done in finding suitable visualization software that has combined features of the MSc-generator and ExTraVis tools.

Reference

[1] V.R. Gibson and J.A. Senn, "System structure and software maintenance performance", Commum. ACM, Vol. 32, pp. 347-358, 1989.

[2] K. Fjeldstad and W.T. Hamlen," Application program maintenance study: Report to our respondents", Tutorial on Software Maintenance, IEEE Computer Society Press, pp. 13 – 30, 1982.

[3] J. Chiksfsky and J.H. Cross II, "Reverse engineering and design recovery: a taxonomy", IEEE software, pp. 13-17, 1990.

[4] L.C. Braind, Y. Labiche and J. Leduc, "Towards the Reverse Engineering of UML Sequence Diagram for distributed Java software", IEEE Tran. on Software engineering, col.32, pp. 642-663, 2006.

[5] T. Systa, "Static and Dynamic Reverse Engineering Technologies for Java software systems", Phd Thesis , 2000.

[6] Y.G. Gueheneuc and T. Ziadi, "Automated Reverse-Engineering of UML v2.0 Dynamic Models", Proc Workshop Object –Oriented Reengineering, 2005.

[7] C. Larman ," Applying UML and Patterns", Prentice –Hall , 2nd edition, 2001.

[8] E. Stroulia and T. Systa, "Dynamic analysis for reverse engineering and program understand", ACM SIGAPP Applied Computing Review, pp. 8-17, 2002.

[9] C. Bennett, "Tool Features for understanding large Reverse Engineering Sequence diagrams" Thesis, Dept. of Computer Science, 2008.

[10] O. Rainer and S. Thomas, "Javavis : Automatic program visualization with object and sequence diagram using the Java debug interface", in Revised Lectures on software visualization. pp 176-190, 2002.

[11] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto and K. Inoue, "Extracting sequence diagram from execution trace of Java Program", In 8th International workshop on principles of software evolution", IEEE computer society, pp. 148-151, 2005.

[12] A. Hamou-Lhadji and T.C. Lethbridge "Techniques for reducing the complexity of object-oriented execution traces", In proc 2nd IEEE international, pp. 35-40, 2003.

[13] M.H. Alalfi, J.R. Cordy Thomas and R. Dean "Automated Reverse engineering of UML sequence diagram for dynamic web application", pp.287-294, 2009.

[14] D. Jerding and S. Rugaber, " Using visualization for architecture localization and extraction", In Proc 4th Working conference on reverse engineering, pp 56-65, 1997

[15] D.F. Jerding, J.T. Stasko and T. Ball, "Visualizing interaction in program execution", In proc 19th International conference on software engineering, pp 360-370, 1997

[16] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J.J. Van Wijk and A. Van Deursen" Understand execution traces using massive sequence and circular bundle views", 15th IEEE international conference on program comprehension, pp. 49-58, 2007.

[17] W. De Pauw, E. Jenson, N. Mitchell, G. Sevtisky, S.M. Vlissider and J. Yang, "Visualizing the executions of Java program", Lecture notes in Computer science, pp 151-162,2001.

[18] W. De Pauw, D. Lorenz, J.vlissides and M. Wegman "Execution patterns in object-oriented visualization", In proc, USENIX, pp 219-234, 1998.

[19] T. Systa, K. Koshimer and H. muller," Shimba- An environment for reverse engineering Java software", software practice, vol. 31, pp 371-394, 2001.

[20] M.M Tikir and J.K. Hollingsworth, "Efficient instrumentation for code coverage testing", In Proc of ACM International symposium on software testing and analysis, pp 86-96, 2002.

[21] GNU operating system. Online. Accessed 15th March 2013. http://www.gnu.org/licenses/gpl.html/

[22] Msc-generator. Online. Accessed 15th March 2013. http://www.mcternan.me.uk/mscgen/

APPENDIX A: Python Decorators

Decorators in Python can be used as a sophisticated method to instrument a given application. A decorator is a callable function that takes any function as an argument and returns a replaced function. In other words it can modify or decorate any function when called. The Figure A1 shows the definition and the working of a decorator. The decorator function is defined as "modify" which always takes function as an arguments. When a function calls the decorator function "modify" it automatically replaces the called function to a newly modified function defined as "changedfunction1" in Figure A1.

Figure A2 shows the calling of a decorator function. Since decorators are unique functions that behave slightly different from a regular function, the process of calling a decorator is also very distinctive. A "@" symbol together with the decorator function name should be written one line above a regular function. In this case, "@modify" is written one line above the actual function"function1" that needs to be modified is shown in Figure A1. During the program execution, when the program encounters a "@" symbol it understands that the following function needs to be passed as a parameter to the decorator. In this case, the function "function1" is passed to the decorator function "modify" and later on the decorator replaces the function"function1" with "changedfunction1()" and executes it. The scope of the project is to insert Entry and Exit logging traces in the start and at the end of a function. Figure A1 show that the function's "function1" code "Original code" is now embedded between a "Start code" and "End code". In this project the "Start code" and "End code" is actually statements that extract the information about caller details, callee details and the message detail during runtime. The added Entry and Exit logging traces are shown in Figure A3.

```
def modify(f):
    def changedfunction1():
        [start code]
        [Original code]
        [end code]
        return
    return changedfunction1
```

Figure A1 Definition of a Python decorator

```
@modify
def function1():
    [Original code]
    return
```

Figure A2 Calling a decorator function

```
def modify(f):
    def changedfunction1():
        Enter =' Enter', stack(function), stack(parameters), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
        Original code
        Exit =' Exit', stack(function), stack(return), stack(caller class),
              stack(caller file), stack(callee class)stack(callee file), timestamp
    return
return changedfunction1
```

Figure A3: Implementing instrumentation using decorators

APPENDIX B: Source code for Observer (Source code Transformation)

Identification and Instrumentation process (Methods)

```
//// Instrumentation.py
////////////////////////////////////////////////////////////////////////////////////////////
parser[j+0]='import sys \n'
                parser[j+1]='import os \n'
parser[j+2]='sys.path.append(os.path.dirname(\'C:\\\\kaushik\\\\DiagnosticFramework_R5+_src\\\\
Log_dynamic\\\\instrumentation.py\'))\n'

                parser[j+3]='from instrumentation import * \n'
                pfile3=open(a,'w')
                pfile3.writelines(parser)
                pfile3.close()
                with open(a) as f:
                    tree = ast.parse(f.read())
                taskfunction=[]
                taskloops=[]
                for node in ast.walk(tree):
                    if isinstance(node, (ast.FunctionDef)):
                        m=node.lineno-1
                        taskfunction.append(m)
                taskfunction.sort()
                taskfunction.reverse()
                f=open(a,'r')
                parser=f.readlines()
                k=0
##                pdb.set_trace()
                while k<len(taskfunction):
                      if ('my_autopct'in parser[taskfunction[k]]) or ('@' in
                      parser[taskfunction[k]]):
                        k=k+1
                    else:
                        count_number=count_number+1
                        if (taskfunction[k]>j+4):
                            b=parser[taskfunction[k]][0:parser[taskfunction[k]].index('def')]
                              if ('self' in parser[taskfunction[k]]):
                              if('self'==parser[taskfunction[k]][parser[taskfunction[k]].index(
                              '(')+1:parser[taskfunction[k]].index(')')]):
                                   parser.insert(taskfunction[k],'')
                                   parser[taskfunction[k]]=b+'@entryExitOnlySelf \n'
                                   k=k+1
                                else:
                                   parser.insert(taskfunction[k],'')
                                   parser[taskfunction[k]]=b+'@entryExit \n'
                                   k=k+1
                            else:
                                parser.insert(taskfunction[k],'')
                                parser[taskfunction[k]]=b+'@entryExitfunc \n'
                                k=k+1
                        else:
                            k=k+1
```

Identification and Instrumentation details (Loops)

```
//// Instrumentation.py
////////////////////////////////////////////////////////////////////////////////////////////
while i<len(taskline):
                        Number_of_Loop=Number_of_Loop+1
                        if ('for' in parser[taskline[i]]) or ('while' in parser[taskline[i]]):
                              b=parser[taskline[i]][:len(parser[taskline[i]])-
                              len(parser[taskline[i]].lstrip())]
                            k=taskline[i]+1
                            while(k<len(parser)):
                                if (parser[k].lstrip().startswith('#')==True) :
                                    k=k+1
                                elif (parser[k].isspace()==True):
                                    k=k+1
                                elif ('' == parser[k]):
                                    k=k+1
                                else:
                                    c=parser[k][:len(parser[k])-len(parser[k].lstrip())]
                                    break
                                if ('visualization' in a and 'ui' in a and 'CW.py' in a and 'for' in
                                parser[taskline[i]] and 'selectedY' in parser[taskline[i]] and
                                'self._selectedYLst' in parser[taskline[i]]):
                                    pass
                        else:
                            parser.insert(taskline[i],'')
                            parser.insert(taskline[i],'')
                            parser.insert(taskline[i]+3,'')
                            parser.insert(taskline[i]+3,'')
                              parser[taskline[i]]=b+'count_%s=0\n'%str(Number_of_Loop)
                              parser[taskline[i]+1]=b+'Myfile.pfile3.write(\'Entry_of_Loop_%s\\n\')\n'%
                              str(Number_of_Loop)
                            parser[taskline[i]+3]=c+'count_%s=count_%s+1
                            \n'%(str(Number_of_Loop),str(Number_of_Loop))
                            parser[taskline[i]+4]=c+'Myfile.pfile3.write(\'Iteration_Number_%s
                            %s\\n\'%str(count_%s))\n'%(str(Number_of_Loop),'%s','%s',str(Number_of_Loop
                            j=taskline[i]+3
                            while j<len(parser):
                                d=len(parser[j])-len(parser[j].lstrip())
                                if (parser[j].isspace()==True):
                                    if j==len(parser)-1:
                                        parser.insert(j,'')
                                       parser[j]=b+'Myfile.pfile3.write(\'Exit_of_Loop_%s
                                      \\n\')\n'%str(Number_of_Loop)
                                        break
                                    else:
                                        j=j+1
                                elif (parser[j].lstrip().startswith('#')==True):
                                    j=j+1
                                elif parser[j]=='':
                                    j=j+1
                                elif (parser[j].lstrip().startswith('return')==True):
                                    if d>len(b):
                                        calculate=parser[j][:len(parser[j])-len(parser[j].lstrip())]
                                        parser.insert(j,'')
                                         parser[j]=calculate+'Myfile.pfile3.write(\'Exit_of_Loop_%s
                                         \\n\')\n'%str(Number_of_Loop)
                                        j=j+2
                                    elif d<=len(b):
                                        parser.insert(j,'')

                                      parser[j]=b+'Myfile.pfile3.write(\'Exit_of_Loop_%s
                                      \\n\')\n'%str(Number_of_Loop)
                                        break
```

Runtime calculation and code insertion details

```
/////RuntimeCalculation.py
////////////////////////////////////////////////////////////////////////////////////////////////

///////
//Calculation of caller, callee and Message details
////////////////////////////////////////////////////////////////////////////////////////////////

dt = datetime.now()
File_Name=inspect.getfile(f).split("\\")
timestamp=str(dt.year)+'-'+str(dt.month)+'-'+str(dt.day)+''+str(dt.hour)+  ':'+  str(dt.minute)+
':'+str(dt.second)+':'+str(dt.microsecond)
caller_file=sys._getframe(1).f_code.co_filename
caller_file= caller_file.split('\\')
caller_class=str(get_class_from_frame(sys._getframe(1)))

//Code inserted in the Entry of the function
////////////////////////////////////////////////////////////////////////////////////////////////
Enter="Entering"+'\t'+str(f.__name__)+'\t'+str(self.__class__.__name__)+'\t'+str(inspect.getarg
spec(f).args[1:])+ '\t'+ str(id(self))+'\t'+str(timestamp)+ '\t'+ File_Name[len(File_Name)-1] +
'\t'+ File_Name[3] + '\t'+ caller_class + '\t'+caller_file[len(caller_file)-1]+'\n'

//Code inserted in the Exit of the function
////////////////////////////////////////////////////////////////////////////////////////////////

Exit_kau="Exited"+'\t'+str(f.__name__)+'\t'+str(self.__class__.__name__)+'\t'+str(ret).replace(
'\n','')  +  '\t'+  str(id(self))+'\t'+str(timestamp)+'\t'+  File_Name[len(File_Name)-1]+  '\t'+
File_Name[3]+ '\t'+ caller_class+ '\t'+caller_file[len(caller_file)-1]+'\n'
```

APPENDIX C: Source code for Observer (Reduction Rules)

Self and constructor call reduction rule

```
////// ReductionRule.py
////////////////////////////////////////////////////////////////////////////////////////////
i=0
count=0

while i<len(parser):
    if((i%5000)==0):
            print i
    call_Verification=parser[i].split('\t')
    if 'Iteration_Number' in parser[i]:
        i=i+1
    elif 'Exit_of_Loop' in parser[i]:
        i=i+1
    elif 'Entry_of_Loop' in parser[i]:
        i=i+1
    elif (call_Verification[1]=='__init__' and reduce_init_calls==1):
        del parser[i]
        count=count+1
    elif (call_Verification[2]==call_Verification[8] and reduce_self_calls==1):
        del parser[i]
        count=count+1
    else:
        i=i+1
```

Redundant Loop Reduction

```
////// ReductionRule.py
//////////////////////////////////////////////////////////////////////////////////
/////// Identify the loop statements and creation of two segments (segment1 and segment2)
/////// for comparison
//////////////////////////////////////////////////////////////////////////////////////
if ('Entry_of_Loop' in parser[k]):
                    match=parser[k][parser[k].rindex('_'):len(parser[k])-1]
                    m=k+1
                    while m<len(parser):
                        if ('Iteration_Number'+match in parser[m]):
                            segment_1=[]
                            n=m+1
                            while n<len(parser):
                                if('Exit_of_Loop'+match in parser[n]):
                                    break
                                elif ('Iteration_Number'+match not in parser[n]):
                                    segment_1.append(parser[n])
                                    n=n+1
                                elif('Iteration_Number'+match in parser[n]):
                                    break
                            while n<len(parser):
                                if ('Exit_of_Loop'+match not in parser[n]):
                                    p=n+1
                                    segment_2=[]
                                    while p<len(parser):
                                        if('Exit_of_Loop'+match in parser[p]):
                                            break
                                        elif ('Iteration_Number'+match not in parser[p]):
                                            segment_2.append(parser[p])
                                            p=p+1
                                        elif('Iteration_Number'+match in parser[p]):
                                            break
////////////////////////////////////////////////////////////////////////////////////////
/////// Comparing two segments (segment1 and segment2)
////////////////////////////////////////////////////////////////////////////////////////
                                    if(len(segment_1) == len(segment_2)) and
                                    (len(segment_1)>0 and len(segment_2)>0):
                                        compare=0
                                        while ( compare<len(segment_1)):
                                            if ('DummyString'in segment_1[compare]):
                                                segLine_1=segment_1[compare]
                                             elif('Iteration_Number_' in
                                             segment_1[compare]):
                                                segLine_1=segment_1[compare]
                                            elif('Exit_of_Loop' in segment_1[compare]):
                                                segLine_1=segment_1[compare]
                                            else:

                                             segLine_1=segment_1[compare].split('\t')
                                             segLine_1=''.join((segLine_1[0],segLine_1[1],
                                             segLine_1[2],segLine_1[8]))

                                            if ('DummyString'in segment_2[compare]):
                                                segLine_2=segment_2[compare]
                                             elif('Iteration_Number_' in
                                             segment_2[compare]):
                                                segLine_2=segment_2[compare]
                                            elif('Exit_of_Loop' in segment_2[compare]):
                                                segLine_2=segment_2[compare]
                                             else:
                                             segLine_2=segment_2[compare].split('\t')
                                             segLine_2=''.join((segLine_2[0],segLine_2[1],
                                             segLine_2[2],segLine_2[8]))
                                            if(segLine_1==segLine_2):
                                                Identical_Flag=True
```

Extraction of Unique Method calls

```
////// ReductionRule.py
//////////////////////////////////////////////////////////////////////////////////////
while i<len(parser):
    if 'Entry_of_Loop_' in parser[i] or 'Iteration_Number_' in parser[i] or
'Exit_of_Loop_' in parser[i]:
        pass
    else:
        collection=parser[i].split('\t')
##        collection[3]='dummy'
        collection[4]='dummy'
        collection[5]='dummy'
        collection[7]='dummy'
        collection='\t'.join(collection[:])
        parser[i]=collection
    i=i+1


print len(parser)
lines = (line.rstrip() for line in parser)
unique_lines = OrderedDict.fromkeys( (line for line in lines if line) )
unique_collection=unique_lines.keys()
first='\n'.join(unique_collection[:])
```

APPENDIX D: Source code for Observer (Extraction of hierarchical structure)

Extraction of hierarchical structure of a system

```
////// Extraction_of_hierarchical_structure_of_system.py
//////////////////////////////////////////////////////////////////////////////////
root =dirname
pattern = "*.py"
for path, subdirs, files in os.walk(root):
    for name in files:
        if fnmatch(name, pattern):
            a=os.path.join(path, name)
            a=a.replace('/','\\')
            file_name=a.split("\\")[len(a.split("\\"))-1]
            file_name=file_name.replace('.py','').capitalize()
            file_name=file_name.replace('.py','')
            directory_name=a.split("\\")[3]
            directory_name=directory_name.capitalize()
##          directory_name=directory_name.replace('.py','')
            f=open(a,'r')
            p = ast.parse(f.read())
            classes = [node.name for node in ast.walk(p) if isinstance(node, ast.ClassDef)]
            if classes!=[]:
                j=0
                classes.append('None')
                while(j<len(classes)):
                    complete= directory_name.split()+file_name.split()
                    complete.append(classes[j].capitalize())
                    print complete
                    t.append(complete)
                    j=j+1
            else:
                classes=['None']
                complete= directory_name.split()+file_name.split()+classes
                t.append(complete)
```

APPENDIX E: Source code for Observer (Format Conversion)

Static information conversion

```
////// FormatConversion.py
////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////
///// H-elements conversion process for ExTraVis
////////////////////////////////////////////////////////////////////////////////////////
t.sort()
count=1
while(k<len(t)):
    if (t[k][0] in trace[len(trace)-1]):
        if(t[k][0] in trace[len(trace)-1])and (t[k][1] in trace[len(trace)-1]):
               statement='\"H\" \"%s\" \"DiagnosticFramwork.%s.%s.%s\"
               \"0\"\n'%(str(count),t[k][0].replace('.py',''),t[k][1],t[k][2])
            trace.append(statement)
            count=count+1
        else:
               statement='\"H\" \"%s\" \"DiagnosticFramwork.%s.%s\"
               \"1\"\n'%(str(count),t[k][0].replace('.py',''),t[k][1])
            trace.append(statement)
            count=count+1
               statement='\"H\" \"%s\" \"DiagnosticFramwork.%s.%s.%s\"
               \"0\"\n'%(str(count),t[k][0].replace('.py',''),t[k][1],t[k][2])
            trace.append(statement)
            count=count+1
    else:
        statement='\"H\" \"%s\" \"DiagnosticFramwork.%s\"
        \"2\"\n'%(str(count),t[k][0].replace('.py',''))
        trace.append(statement)
        count=count+1
        statement='\"H\" \"%s\" \"DiagnosticFramwork.%s.%s\"
        \"1\"\n'%(str(count),t[k][0].replace('.py',''),t[k][1])
        trace.append(statement)
        count=count+1
        statement='\"H\" \"%s\" \"DiagnosticFramwork.%s.%s.%s\"
        \"0\"\n'%(str(count),t[k][0].replace('.py',''),t[k][1],t[k][2])
        trace.append(statement)
        count=count+1
    k=k+1

////////////////////////////////////////////////////////////////////////////////////////
///// PCR-elements conversion process for ExTraVis
////////////////////////////////////////////////////////////////////////////////////////

PCR_elements=[]
m=2
##pdb.set_trace()
while(m<len(trace)):
    current_level=trace[m].count('.')
    n=m-1
    while(n>0):
        previous_level=trace[n].count('.')
        if (previous_level==current_level-1):
            previous_rel_number=trace[n].split(' ')[1]
            current_rel_number=trace[m].split(' ')[1]
            statement='\"PCR\" %s %s\n'%(previous_rel_number,current_rel_number)
            PCR_elements.append(statement)
            break
        else:
            n=n-1
    m=m+1
```

Dynamic information conversion

```
////// FormatConversion.py
//////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////
///// S-elements conversion process for ExTraVis
//////////////////////////////////////////////////////////////////////////////////////

pfile=open(file_path.name,'r')
signature_trace=pfile.readlines()
signature=[]
i=0
while(i<len(signature_trace)):
    if 'Entry_of_Loop_' in signature_trace[i] or 'Iteration_Number_' in signature_trace[i] or
'Exit_of_Loop_' in signature_trace[i]:
        pass
    else:
        parser=signature_trace[i].split('\t')
        method_signature=parser[7]+'.'+parser[6].replace('.py','')+'.'+parser[1]+'()'+'\n'
        signature.append(method_signature)
    i=i+1
pfile.close()
lines = (line.rstrip() for line in signature)
unique_lines = OrderedDict.fromkeys( (line for line in lines if line) )
take=unique_lines.keys()
method_statements=[]
j=0
while(j<len(take)):
    statment='\"S\" "%s" "%s"\n'%(str(j),take[j])
    method_statements.append(statment)
    j=j+1
//////////////////////////////////////////////////////////////////////////////////////
///// R-elements conversion process for ExTraVis
//////////////////////////////////////////////////////////////////////////////////////
pfile=open(file_path.name,'r')
method=method_statements
relat=[]
checker=trace
call_relation=pfile.readlines()
i=0
count=0
##pdb.set_trace()
while(i<len(call_relation)):
    if((i%2000)==0):
        print i
    parser=call_relation[i].split('\t')
    if(parser[0]=='Entering'):
        caller_detail='.'+parser[8].capitalize()
        callee_detail='.'+parser[2].capitalize()
        j=1
//////////////////////////////////////////////////////////////////////////////////////
///// Retrieving the unique caller ID
//////////////////////////////////////////////////////////////////////////////////////
        while(j<len(checker)):
            if (checker[j].count('.')==3):
                if(caller_detail == '.None') or (caller_detail == '.Test'):
                    caller_check=parser[9].replace('.py\n','').capitalize()+caller_detail
                    if(caller_check in checker[j]):
                        relation_caller=checker[j].split(' ')[1]
                        break
                    else:
                        j=j+1
                else:
                      caller_check=checker[j].split(' ')[2]
                      caller_check=caller_check[caller_check.rindex('.'):caller_check.rindex('\"')]
                      if(caller_detail == caller_check):
                        relation_caller=checker[j].split(' ')[1]
```

Dynamic information conversion

```
////// FormatConversion.py
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
///// Retrieving the unique callee ID
//////////////////////////////////////////////////////////////////////////////////
while(k<len(checker)):
            if (checker[k].count('.')==3):
                if(callee_detail == '.None') or (callee_detail == '.Test'):
                    callee_check=parser[6].replace('.py','').capitalize()+callee_detail
                    if(callee_check in checker[k]):
                        relation_callee=checker[k].split(' ')[1]
                        break
                    else:
                        k=k+1
                else:
                    callee_check=checker[k].split(' ')[2]
                    callee_check=callee_check[callee_check.rindex('.'):callee_check.rindex('\"')]
                    if(callee_detail == callee_check):
                        relation_callee=checker[k].split(' ')[1]
##                        print callee_detail+'\t'+callee_check
                        break
                    else:
                        k=k+1
            else:
                k=k+1

//////////////////////////////////////////////////////////////////////////////////
///// Retrieving the unique Signature ID
//////////////////////////////////////////////////////////////////////////////////
l=0
        while(l<len(method)):
            method_signature=parser[7]+'.'+parser[6].replace('.py','')+'.'+parser[1]+'()'
            method_checker=method[l].split(' ')[2]
            method_checker=method_checker.replace('\"','')
            if(method_signature in method_checker):
                signature_number=method[l].split(' ')[1]
##                print signature_number
                break
            else:
                l=l+1

//////////////////////////////////////////////////////////////////////////////////
///// R element conversion
//////////////////////////////////////////////////////////////////////////////////
statment='\"R\" %s %s \"0\" \"%s\" %s \"DummySignature\"\n'%
(relation_caller,relation_callee,str(count),signature_number)

relat.append(statment)
count=count+1
```

Concatenation of all the converted data

```
////// FormatConversion.py
//////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////////////
///// Concatenation of Header + static information conversion + dynamic information conversion
//////////////////////////////////////////////////////////////////////////////////////////

Header_file=['\"LevelSeperator\" \".\"\n','\"ElmType\" \"0\" \"python.package\"\n','\"ElmType\"
\"1\" \"python.file\"\n','\"ElmType\" \"2\" \"python.class\"\n','\"RelType\"
\"0\",'\"HierarchyDepth\" \"4\"\n','\"HierarchyElements\" \"%s\"\n','\"ParentChildRelations\"
\"%s\"\n','\"Signatures\" \"%s\"\n','\"Relations\" \"%s\"\n']
Header_file[6]=Header_file[6]%(str(len(trace)-1))
Header_file[7]=Header_file[7]%(str(len(PCR_elements)))
Header_file[8]=Header_file[8]%(str(len(method_statements)))
Header_file[9]=Header_file[9]%(str(len(relat)))
input_file=Header_file+Hierachical+PCR_elements+method_statements+relat
```