Systematic Review of Trace Abstraction Techniques for Program Comprehension

Taranpreet Singh Bhatia
Department of Electrical and Computer Engineering, Concordia University
Montreal, Canada
Email: taranpreetsinghbhatia@gmail.com

***Abstract*—Over the years, there has been a lot of studies and experiment conducted in the field of program comprehension with dynamic analysis. Program comprehension can be linked-to testing, debugging, and software maintenance. Software maintenance and evolution go through a lot of difficulties when it comes to the understanding of the program. There are lots of technique which are involved in the understanding of the program. Execution traces is one of the most focused areas in this regard as it provides us with the most intimidate details of program behavior. Traces, however, are very large and this makes it difficult to effectively do the analysis. In this paper, we will discuss the various techniques used in execution traces along with their cons and pros in the past decade. These techniques have not only shown their advantages but also they are limited to some extent due to their disadvantages.**

***Index Terms*—Program comprehension, dynamic analysis, execution traces, software maintenance.**

I. Introduction

Software engineers face a lot of problems when it comes to the maintenance and evolution of software. Before doing any changes to the software, developers need to understand the behavior of the program.

Program comprehension is an important process in software development which helps in the understanding of the source code and maintain it. Research in the field of program comprehension started more than 30 years ago. At that time, many kinds of research were conducted on how developers understand the program and its source code. But the software was not that complex and was small as compared to the software written now.

Earlier, understanding of the program could be done by documentation or through visualization methods. But what if the documentation is poorly made or is not updated on time. What if visualizations are not up to the mark or their designs are not very well maintained. It will be difficult for software developers to maintain the software or to do any changes to its code. This problem also arises when the original developers of the code shift to the other projects or companies.

Dynamic analysis, especially trace analysis, has its perspective to showcase the behavior of the system for program comprehension. Effectively analyzing the traces become very difficult because of their vastness. Therefore, different researches have been carried out in the past decade in this regard. Several tools are employed to fetch important details during the run-time. However, these techniques have their advantages and limitations. In this paper, we will be discussing all the techniques which use execution traces to understand the behavior of the system. We will be extracting all the research papers between 2010 and 2020 and discuss their advantages and disadvantages.

Execution traces are very complex and to fetch them becomes even expensive. Execution traces contain information about method calls and returns, object construction, and destruction, etc. Usually, such events numbered in thousands in execution traces. This becomes too vast for the programmers to even look at it without evening thinking of understanding the system. So tools are introduced which extract the important information from the traces. Developers try to understand these traces to get the best picture of the behavior of the system.

The remaining part of this paper discusses the following sections. Section 2 discusses about the background and related work done in this field. Section 3 covers the article selection process that we undergo. Section 4 discusses the different trace abstraction techniques. In section 5, we discuss the problem faced while writing this survey.

II. Background and related work

Many strategies for program comprehension have been suggested in the literature to gain an understanding of source code and learn about the behavior of the program. From past researches, the most focused area in program comprehension is the source code and program behavior [23]. This section summarizes the number of strategies in program comprehension chosen by the developer. The first research on "program comprehension through dynamic analysis" can be traced back to 1972, in which Biermann and Feldman created finite-state machines from execution traces [24]. After then, many researches were conducted in this field to gain momentum during the 1980s and 1990s.

In 1988, Kleyn and Gingrich [25] worked on the behavior views of object-oriented programs. They used TraceGraph, a tool that uses trace information to create views of program structure.

A few years later, De Pauw et al. [26], [27], [28], in 1993, started their research on the program visualization method in which they used "execution pattern" to visualize traces in a scalable manner.

In 1995, Wilde and Scully [29] established the field of feature location with their Software Reconnaissance tool.

Wilde et al. continued their research in this field with more focus on evaluation [30][31][32].

Koskimies and Mo¨ssenbo¨ck [33] presented another visualization, in 1996, which involves reconstructing the scenario diagrams from the execution traces.

In 1997, Jerding et al. [34], [35] presented their famous ISVis tool which helped in the visualization of large execution traces. The AVID tool presented by Walker et al. [36] helped in visualizing the dynamic information at the architectural level.

In 1999, Ball [37] bring together the concept of frequency spectrum analysis. He presented the analysis of frequencies of different program entities in traces that helped software developers to decompose programs and recognize related computations.

Visualization techniques have become a more popular approach in the area of program comprehension as they conveyed information that is comfortably read by humans.

One popular visualization technique is the UML sequence diagram. These techniques were used by De Pauw et al. [38], Systa et al. [39], and Briand et al. [40]. Other popular trace compaction techniques were presented by Reiss and Renieris [41] and Hamou-Lhadj et al. [42], [43], [44].

Another research body has been conducting a study on the feature analysis of the program. Work on fundamental analyses of program features was done by Greevy et al. [45], [46], and Kothari et al. [47] until Wilde and Scully [48] presented their work which becomes extremely popular in this field.

III. Article Selection

This section mainly focuses on the initial article selection criteria and online resources to look for conference papers and journals.

A. SEARCH STRATEGY

The search strategy includes proper keywords to get relevant papers from digital libraries. However, as mentioned by Brereton et al. [1] that current digital libraries do not offer a sympathetic approach for the keywords to find relevant articles, still holds. Also, it is difficult to search for related keywords in program comprehension as this field further extends to debugging, testing, and software maintenance. We, therefore, moved on with our search using keywords like program comprehension and execution traces. This survey primarily focuses on the articles published between 2010 and 2020.

We employed a search strategy on different digital libraries and the results are as shown in the table. We found 176 articles(see Fig. 1) in this period starting from 2010 till 2020.

B. SELECTION STRATEGY

After all the above articles from different digital libraries, we employed two selection strategies to manually analyzing and handpicking the relevant articles.

1) The article must have a program comprehension as to its core objective. This must exclude articles focusing on debugging and testing.

2) The article must focus on trace analysis technique. To satisfy this approach, the article must use and explain one or more trace analysis techniques.

C. FINAL SELECTION RESULTS

Finally, we decided to manually go through each of these 176 papers. Most of the papers mentioned their technique in the abstract itself which helped us to quickly filter out the papers to the final count of 19. We already were careful to rule out any duplicate papers which could be available on different digital libraries.

IV. Trace Abstraction Techniques

A. Pattern Based Approach

The pattern-based approach operates by grouping events with similar patterns. Many pattern-based techniques were presented in the past 10 years.

1) Pree's Meta Pattern: Kunihiro et al. [2] present a method that abstracts the history of the interaction of objects using Pree's meta pattern usage. In their method, they focused on the template-hook structure of object-oriented programming to identify objects that are strongly related to each other. They then generate the sequence diagram of the system's behavior by grouping the objects.

Their research reduced the amount of information packed in the execution trace and to generate visual information that helps in the program comprehension process. This method helped developers to grasp the big picture of the system behavior because it visualizes the intergroup interactions. In their method, they present two grouping methods. Grouping objects based on meta patterns avoid excessive abstractions as it concerns with the correlations between objects. Grouping objects in GSOs per class reduce the number of objects which helped in effectively handle a large amount of information.

2) Inferring Hierarchical Motifs: Motifs are defined as patterns in the traces that are flexible to the minor changes in the execution. These motifs are further captured in the hierarchical model. These hierarchical models can provide us with the behavior at a high level while keeping intact the details of execution in a structured manner. The authors[3] designed a visualization to interact with these models. This improved the accuracy to comprehend the program by 54 percent.

Extracting motifs is a challenging task because of the dynamism, asynchrony, and non-determinism in the execution of the program. Like in JavaScript applications, understanding the results of user action is very difficult as it is often asynchronous to capture by analyzing the code. Furthermore, to define the behavior as a motif, it should recur during execution. Different execution may vary in details and they may not converge to one motif.

The Authors employ the tool SABALAN in their technique. SABALAN helped to improve the accuracy of the overview of the application's behavior as compared to other techniques. Also, it helped developers to better gain the understanding of inside details of interactions of components. This technique

SOURCE TYPE	ACRONYM	SOURCE TITLE	NO. OF ARTICLES 2010-2020
Conference	IEEEICPC	IEEE International Conference On Program Comprehension	30
	IEEEICOSMI	IEEE International Conference On Software Maintenance Icsm	19
	ACMICPS	ACM International Conference Proceeding Series	15
	IEEEACMICPC	2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)	2
	IEEEICPC	2011 IEEE 19th International Conference on Program Comprehension	3
	ICPC	2013 21st International Conference on Program Comprehension (ICPC)	2
	IEEEACMICPC	2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)	2
	IEEEICPC	2015 IEEE 23rd International Conference on Program Comprehension	2
	IEEEPCODA	2015 IEEE 6th International Workshop on Program Comprehension through Dynamic Analysis (PCODA)	1
	ICIT	2017 8th International Conference on Information Technology (ICIT)	2
	CITS	2012 International Conference on Computer, Information and Telecommunication Systems (CITS)	1
	QRS-C	2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)	1
	ISSREW	2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)	1
	ICSM	2011 27th IEEE International Conference on Software Maintenance (ICSM)	1
	ICECCS	2011 16th IEEE International Conference on Engineering of Complex Computer Systems	1
	ACIS	2010 Eighth ACIS International Conference on Software Engineering Research, Management and Applications	1
	ICSE	2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)	1
	WCRE	2012 19th Working Conference on Reverse Engineering	1
Journal	IEEETOSE	IEEE Transactions On Software Engineering	11
	JOSEAP	Journal Of Software Evolution And Process	15
	JOSAS	Journal Of Systems And Software	15
	IST	Information And Software Technology	12
	ESE	Empirical Software Engineering	13
	SCP	Science of Computer Programming	2

Fig. 1: Initial Article Selection

using SABALAN to focus more on the minute portion of the code that was relevant in the study of software maintenance.

3) Concept Mining: Concept mining is the task of finding and identifying concepts like domain concepts in the code. In this paper, the author [22] worked on a dynamic-programming algorithm that splits an execution trace into segments that represent concepts. They planned to use techniques derived from Latent Dirichlet Allocation (LDA) to assign meaning to segments that further help maintainer in concept assignment.

This approach improves the performance and scalability of the execution traces. This method takes much less time as CPU time was improved by 99

After splitting the traces, the authors provide two possibilities of assigning meaning to segments. If the trace is split into short segments, then applying LDA to every segment becomes problematic.

B. Execution Phases

Some of the techniques to analyze the traces depends on the execution phases. These phases can help the software developers to understand the trace content at a high level and dig more into the details. These execution phases may include initializing variables, specific computation, etc. Here are a few of these techniques discussed in the past 10 years.

1) Automatic Detection of Execution Phases: In 2010, Akanksha Agarwal in her work[4] proposed an approach that automatically detects the execution phases in the traces. Her phase detection approach is based on the fact when a certain set of events involved in one phase start to "fade" and enter into a new phase. Moreover, her approach is online-based which means one can work on traces the moment it gets generated. She focused on the traces of routine calls like function and procedure in her approach.

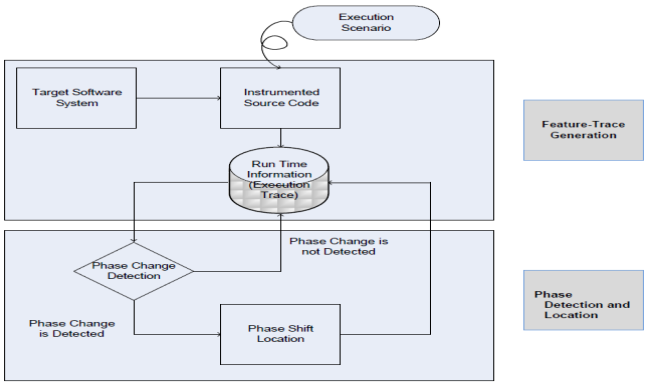

Fig. 2: Approach diagram [4]

This approach is online which means we do not need to extract the traces and use it offline rather we can work on these traces on the go. This approach is independent of the language that is used to develop the system. This approach is very effective in large traces to detect the execution phases. This approach is applied only to the object-oriented system. In her approach, she used JHotDraw which was used to capture routine calls. But this tool also captures noise like mouse movements and getter and setter methods. It's a challenge to explore a large set with several phases. This could also turn as a challenging task if the traces are complex.

2) Phase Detection Approach: In this paper, which was presented in 2010, Heider, Akanksha, and Hamou-Lhadj [5] worked on a novel approach that detects the execution phases that compose it and simplifies the analysis of large execution traces. Their approach divides the program into phases which makes it simple for software developers to easily grasp its content. They focused on traces of method calls which is an easier subject for the understanding of program comprehension.

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

Fig. 3: Pseudo code of phase finding algorithm [5]

This algorithm eliminates the need to save the traces as it reacts on the traces the moment it gets generated. This follows the online approach rather than offline which gives it advantage over the other approaches of similar type.

In this algorithm, authors are using a certain threshold to suggest if a new phase is taking place or not. They found out that the threshold is application specific and the tool in this approach should allow flexibility for the varying threshold.

3) Gestalt Psychology: In 2011, Heider and Hamou-Lhadj [6] presented a novel trace analysis technique inspired by Gestalt law. Gestalt psychology is based on similarity, proximity, and continuity which is a concept to showcase the operational laws of human behavior. The Authors model the concept of the Gestalt principle in the context of execution traces.

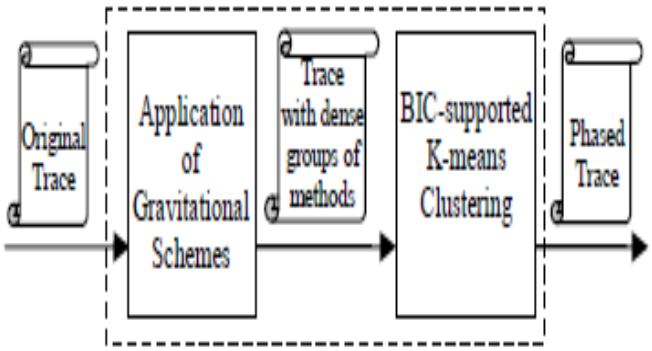

Fig. 4: Detailed view of the execution phase detection unit [6]

Their approach based on Gestalt Psychology determines the degree of similarity and repeated call sequences automatically in certain execution phases instead of efforts from the users. Even to detect phases this approach takes in to account the nesting level of methods which is missing in other approaches. This approach does not measure similarity and continuity among trace elements as it has its own issues. Moreover, this approach also helps maintainers to automatically distinguish between various phases using the K-means clustering algorithm.

The drawback of this approach is that to use the clustering algorithm, the user must know the number of phases and provide it the clustering algorithm to make it work. This problem can be addressed using BIC (Bayesian Information Criterion) which supports K-means clustering and locates the phases automatically.

4) Text Mining Techniques: In 2011, Heidar, Hamou-Lhadj, and Mohak Shah [7] exploits the text mining technique to analyze the execution traces. They proposed a trace exploration approach that automatically identifies the information needed about the phases. This approach uses the cosine similarity metric which detects redundant phases to provide a representation of the flow of phases.

The advantage of this approach is that it does not rely on efforts hypothesis to detect phases, their flow, and relevant information. Moreover, the important elements of execution traces in this approach can be represented in the UML sequence diagram and further can be used for redocumentation. This approach can be helpful in the areas of trace summarization.

Authors limit themselves to the type of utilities like accessing methods or language libraries explained by Hamou-Lhadj et al. [8] that can be easily detected without using advanced processing techniques.

5) Using Heuristic Search: Omar et al. [9] in their paper, published in 2014, proposed an approach that identifies feature

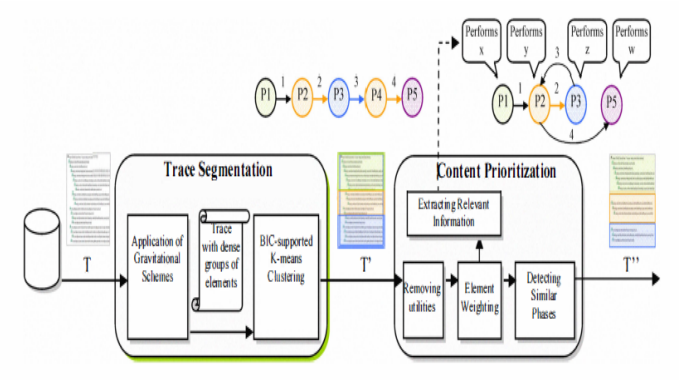

Fig. 5: Overview of proposed approach [7]

level phases from the events collected from execution traces. They applied their approach to the optimization problem to form phases using the dynamic information from execution traces thus minimizing coupling while maximizing cohesion. Their technique is based on a metaheuristic approach in which different phases involve different objects.

Authors contribute their research by addressing the high-level phase detection which comes from Watanabe et al [10]. Authors [10] carried out their research with some assumptions like different functionalities makes use of different sets of objects. However, Omar et al [9] are in advantage because their research does not assume any assumptions. In paper [9], the approach detects periods of the program execution itself that abstracts the external behavior.

However, their approach does not guarantee success in all the heuristic approaches as there may be differences in the representation of the problem and heuristic approach.

6) Using Probabilistic and Gaussian Models: In his paper, Mohammad Rejali [11] presented a novel trace analysis technique i.e. SumTrace which uses trace to segment it into more small and manageable groups to showcase the execution phases of the trace. SumTrace is a mixture of both probabilistic and Gaussian models.

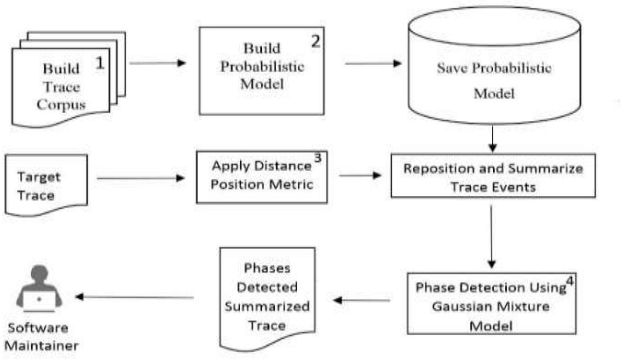

Fig. 6: The SumTrace process for extracting execution phases from traces [11]

According to the author, the SumTrace technique is fast because it needs only one pass through the trace to get meaningful segments. SumTrace assigns each function of the execution trace to a specific phase which comes out as an advantage of what the objective is i.e. trace segmentation or trace summarization.

The authors are using a probabilistic model that requires a data corpus. In their approach, they are using trace corpus. To provide good coverage, the authors argued that traces should cover multiple features of the system. Also, in case of system change like new patches, traces must be updated which becomes time-consuming.

7) High Performance Computing (HPC) Systems: Journal presented by Alawneh and Hamou-Lhadj [12] discussed the trace segmentation approach of HPC events to small and meaningful clusters. This approach can be widely used by software analysts who want to maintain software systems such as enhancing existing features. Authors focused on distributed memory applications particularly on MPI (Message Passing Interface).

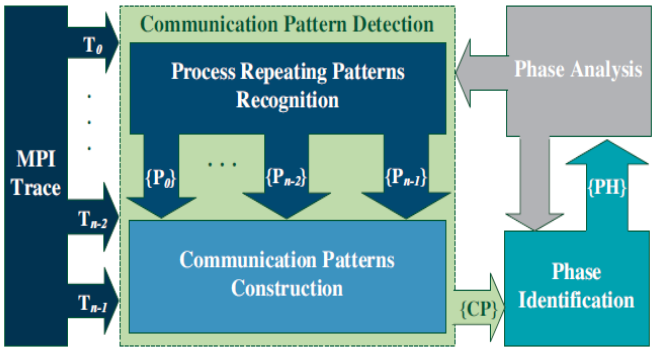

Fig. 7: Approach for segmenting traces of inter-process communication [12]

This approach can be used to verify the behavior of the system with the intended behavior during design. For those who do not maintain the systems but want to report the behavior system, this approach can be quite useful. One can also deduce the number of processes involved in a pattern by inferring some statistical data from this approach.

The above approach requires multiple passes through the execution traces until the longest pattern is formed which is time-consuming. There is no way to filter out low-level utilities with a high level which affects the scalability of the approach. The authors labeled the phases manually by referring to the documentation which is not desirable. The tool used to generate trace must handle indeterminism and other complex scenarios.

8) TRIADE: A Three-Factor Trace Segmentation Method: Raphaël and Hamou-Lhadj [13], in 2019, worked on enhancing the trace segmentation of method calls considering three factors: method names, method parameters, and method calling relationships. This approach helped in understanding the trace segmentation by the behavior of the program rather than the existing algorithm. The authors demonstrated experi-

mentally the importance of extracted elements from the trace analysis.

The experiments conducted by authors show that the use of method parameters with the method name and calling relationship improve the accuracy of the trace segmentation process. This 3-factor approach is better considering the 2-factor approach as authors deduce that the methods present are behavior-specific than utilities and the behavior segment was not assigned in wrong segments which are the case in the 2-factor approach.

This trace analysis fails in two conditions. First, if every method in the summary is a utility and secondly if the behavior-specific method is not under its actual behavior. They call them mis-assigned methods.

C. Database approach

The database approach focuses on the understanding of database access behavior of the program that is not only largely ignored but has also emerged as an important aspect of program comprehension. This approach shows how students/developers try to understand the interactions between various databases and the application program. This approach mainly focuses on the data manipulation behavior of the program. We found two concepts that were mentioned in the papers between 2010 and 2020. These are mainly, SQL execution traces and relational database concepts.

1) SQL execution traces: In SQL execution traces, the interactions between a program and databases produce a conceptual interpretation that usually relies on dynamic SQL. These SQL queries are dynamically constructed before sending it to a database server.

Nesrine and Anthony [14] presented a technique in which they initially captured SQL trace and then they analyze the dependencies among the queries of the SQL trace. At last, they produce the conceptual interpretation of SQL trace execution. The goal was to understand the interaction between program and database, rather than interactions between source code entities. Therefore, the input was SQL execution traces instead of source code fragments.

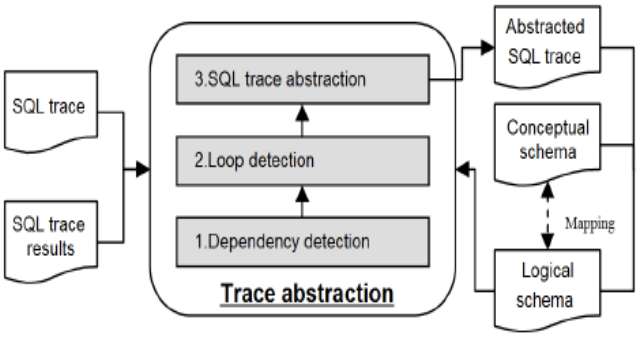

Fig. 8: Overview of approach [14]

The plus point of their technique is they started their abstraction with the SQL queries and the result of their queries. This phase does not depend on the SQL trace capturing techniques. Moreover, their approach conceptualizes the interpretation of SQL execution traces in terms of platform-independent and domain-specific models. The main aim was to understand the SQL data manipulation language (DML) queries that are allowing programs to change the contents of the database. But to move forward with this technique, it must contain at least one trace of SQL queries corresponding to the program execution scenario.

2) Relational database approach: Sahel et al. [15] present an approach which helps in measuring and analyzing the amount of information present in execution traces. The approach was based on the relational database concept which consists of two methods. Firstly, they defined suitable database operations to apply on execution trace files. Secondly, they transformed execution trace files into compatible database files. The above proposal helps a lot in evaluating software security vulnerabilities using log files. Using relational database operations can help in reducing the complexity of trace analysis thus providing the information of interest.

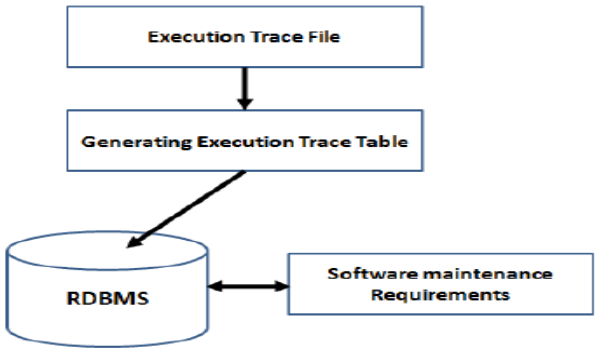

Fig. 9: Overview of approach [15]

Their technique produced very good results but with the compression of information which is no longer readable by humans.

D. Based on Framework

1) Using Ontologies: Newres et al. [16] represented a framework using ontologies to represent the semantics of execution traces and dynamic analysis. They used ontologies to enhance the traces at the conceptual level. Their approach uses the knowledge base to capture the concepts concerned with traces and analysis and then experts can explore and analyze traces within this framework. To formally represent knowledge in the domain, Ontologies provide a way in the form of relationships between them.

They have used a modest amount of traces in their test cases, but the framework can capture and work with millions of lines of traces. Using ontologies can help in integrating many different types of knowledge. Trace analysis can be done both by human and software in the form of program comprehension.

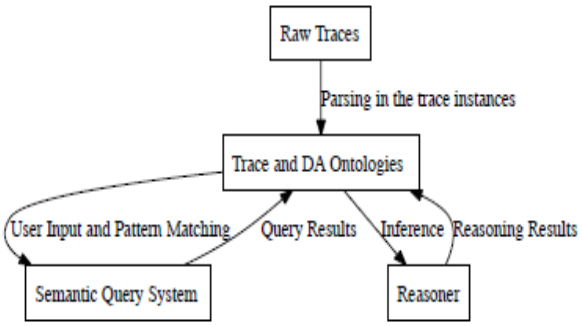

Fig. 10: Overview of approach [16]

While combining dynamic analysis and ontologies, one of the main problems faced is a large number of ordered facts. Here authors dealt with sets of finite ordered facts which were known beforehand. The author suggested increasing the integration of traces derived through queries with a framework.

2) Instruction-level Tracing: Authors in this paper [17] have described a framework that is based on dynamic binary translation and does not require the static instrumentation of the program. This framework collects the detailed user-mode execution traces of the programs and re-simulates them with full fidelity to the instruction level.

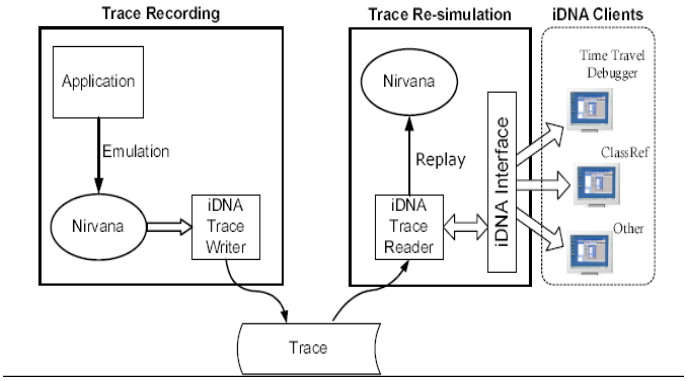

Fig. 11: Tracing framework [17]

The traces produced can be transmitted to other machines for the purpose of re-simulation and analysis because of their task-independent and self-controlled nature. This approach is fully based on software and does not require special tools. Traces can be captured on demand by linking it to the running process and it does not affect the performance of the tool when tracing is off.

For Trace compression, value prediction-based compression (VPC) can result in better compression than this scheme.

3) Utility Classes Detection Metrics: Hasan Abualese et al.[18][19] focused on easing the complexity of the execution traces by detecting and then removing the utility classes. This technique becomes very useful in the software comprehension process. This technique consists of two utility detection metrics that are based on dynamic coupling measures that detect the utility component in the traces. To reduce the complexity of traces, decoupling proved to be very useful as it keeps trace components while simplifying the traces. This framework includes filtering the scope, detecting utility and utility decoupling components.

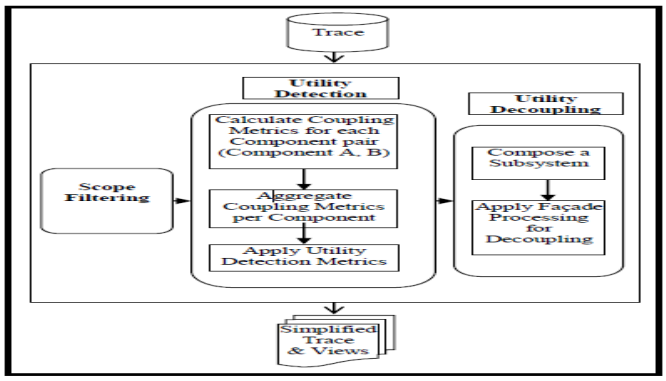

Fig. 12: Proposed Trace Simplification Framework [19]

The advantage of this technique is that it reduces the complexity of the traces by filtering out the utility components that disturb the relationship of the components in traces. The dynamic coupling has its advantage of detecting utility class as the more widespread use of late binding and polymorphism. The dynamic coupling used by the author has a large scope more than the system. Moreover, decoupling helps in reducing the complexity without touching trace components.

The authors insisted on defining the utility detection metrics which is not limited to system scope. They also insist on a modified module façade to ensure the separation of utility components. Techniques presented by authors remove some components and events from execution traces which results in creating gaps and holes in trace structure. Detecting utility class becomes a difficult task when software passes through several maintenance cycles.

E. UCM approach

1) Generating Software Documentation: Edna Braun et al. [20] in their paper described a unique technique that automatically extracts and visualizes software behavioral models from the execution traces. They filtered out low-level software components via algorithms to summarize the lengthy traces. They used the Use Case Map (UCM) scenario notation to visualize the traces that further can be used to document the software.

The limitation of this technique is that the authors did not test this methodology with non-open source and non-Java systems. The UCM approach was only common with the representative of the University of Ottawa than the average software which adds some bias to the results. Tests were conducted with a non-production system that could contain bugs and can affect the results of the experiment. During the filtering step, useful data may get filtered out by the Small and Simple filter.

This approach could benefit from more specific handling of multithreaded or distributed applications. The tool they used TraceToUCM did better on the average than the other program comprehension techniques which exploit dynamic analysis.

2) Recovering System Availability Requirements: J. Hassine and A. Hamou-Lhadj [21] proposed a dynamic analysis approach that recovers availability requirements from execution traces of the system. Recovered availability requirements are then described and visualized by using the Use Case Maps (UCM) language of the ITU-T User Requirements Notation (URN) standard.

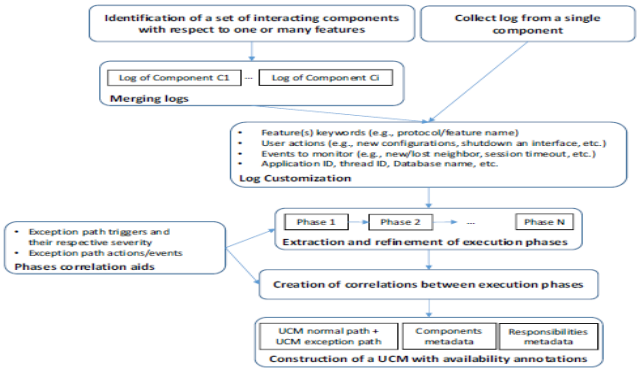

Fig. 13: Proposed Trace Simplification Framework [21]

This technique requires prior knowledge of any possible involvement between components in the case of multiple components. Additional semantic information about the running system is required to establish accurate correlations between execution phases. To present their work, authors have used simple configuration which does not comply with the production environment.

The UCM approach they have used in this paper offers a flexible way that represents non-functional requirements at different levels of abstractions using the plugin/stub concept.

V. Problems Faced

Several problems were faced while writing the survey. Digital libraries do not provide adequate functionality in fetching journals about specific topics like execution traces, in this case. ACM digital library yields more than 16000 search results, many of them not even relevant to the current subject.

Another issue we faced is that this survey is focused on software maintenance but execution traces are also involved in testing and debugging.

Acknowledgment

I would really like to recognise Wahab Hamou-Lhadj from Concordia University's Department of Electrical and Computer Engineering for his insightful feedback that enabled me tremendously in the improvement of this document.

References

[1] P. Brereton, B.A. Kitchenham, D. Budgen, M. Turner, and M.Khalil, "Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain," J. Systems and Software, vol. 80, no. 4, pp. 571-583, 2007.

[2] Kunihiro Noda, Takashi Kobayashi, and Kiyoshi Agusa," Execution Trace Abstraction based on Meta Patterns Usage".

[3] Saba Alimadadi, Ali Mesbah, Karthik Pattabiraman," Inferring Hierarchical Motifs from Execution Traces".

[4] Akanksha Agarwal,"Trace Abstraction Based on Automatic Detection of Execution Phases".

[5] Heidar Pirzadeh, Akanksha Agarwal, Abdelwahab Hamou-Lhadj,"An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension".

[6] Heidar Pirzadeh, Abdelwahab Hamou-Lhadj,"A Novel Approach Based on Gestalt Psychology for Abstracting the Content of Large Execution Traces for Program Comprehension".

[7] Heidar Pirzadeh, Abdelwahab Hamou-Lhadj, Mohak Shah,"Exploiting Text Mining Techniques in the Analysis of Execution Traces".

[8] A. Hamou-Lhadj and T.C. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," In Proc. ICPC'06, 1 8 1-190, 2006.

[9] Omar Benomar, Houari Sahraoui, and Pierre Poulin,"Detecting Program Execution Phases Using Heuristic Search"

[10] Watanabe, Y., Ishio, T., Inoue, K.: Feature-level phase detection for execution trace using object cache. In: Proc. Intl. Work. on Dynamic Analysis, WODA, pp. 8–14. ACM (2008)

[11] Mohammad Reza Rejali,"Effective Segmentation of Large Execution Traces Using Probabilistic and Gaussian Mixture Models".

[12] Luay Alawneh, Abdelwahab Hamou-Lhadj, Jameleddine Hassine,"Segmenting large traces of inter-process communication with a focus on high performance computing systems".

[13] Raphaël Khoury, Abdelwahab Hamou-Lhadj and Mohamed Ilyes Rahim, Sylvain Hallé, Fabio Petrillo,"TRIADE: A Three-Factor Trace Segmentation Method to Support Program Comprehension"

[14]] Nesrine Noughi, Stefan Hanenberg, Anthony Cleve," An Empirical Study on the Usage of SQL Execution Traces for Program Comprehension,"

[15] Sahel Alouneh1, Sa'ed Abed, Bassam Jamil Mohd, Ahmad Al-Khasawneh, "Relational Database Approach for Execution Trace Analysis"

[16] Newres Al Haider, Benoit Gaudin, and John Murphy,"Execution Trace Exploration and Analysis Using Ontologies"

[17] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray,Milenko Drini´c, Darek Miho˘cka, Joe Chau,"Framework for Instruction-level Tracing and Analysis of Program Executions".

[18] Hasan Abualese and Putra Sumari, Thamer Al-Rousan and Mohammad Rasmi AL-Mousa, "Utility Classes Detection Metrics for Execution Trace Analysis"

[19] Hasan Abualese and Putra Sumari, Thamer Al-Rousan and Mohammad Rasmi AL-Mousa, "A Trace Simplification Framework"

[20] Edna Braun, Daniel Amyot, and Timothy C. Lethbridge, "Generating Software Documentation in Use Case Maps from Filtered Execution Traces"

[21] Jameleddine Hassine and Abdelwahab Hamou-Lhadj, "Toward a UCM-Based Approach for Recovering System Availability Requirements from Execution Traces"

[22] Soumaya Medini, "Scalable Automatic Concept Mining from Execution Traces"

[23] Janet Siegmund Ivonne Schrter, Jacob Krger and Thomas Leich. Comprehending studies on program comprehension. In IEEE 25th International Conference on Program Comprehension (ICPC), pages 308–311. IEEE, 2017.

[24] A.W. Biermann, "On the Inference of Turing Machines from Sample Computations," Artificial Intelligence, vol. 3, nos. 1-3, pp. 181-198, 1972.

[25] M.F. Kleyn and P.C. Gingrich, "Graphtrace—Understanding Object-Oriented Systems Using Concurrently Animated Views," Proc. Third Conf. Object-Oriented Programming Systems, Languages, and Applications, pp. 191-205, 1988.

[26] W. De Pauw, R. Helm, D. Kimelman, and J.M. Vlissides, "Visualizing the Behavior of Object-Oriented Systems," Proc. Eighth Conf. Object-

Oriented Programming Systems, Languages, and Applications, pp. 326-337, 1993.
[27] W. De Pauw, D. Kimelman, and J.M. Vlissides, "Modeling Object-Oriented Program Execution," Proc. European Object-Oriented Programming Conf., pp. 163-182, 1994.
[28] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman, "Execution Patterns in Object-Oriented Visualization," Proc. Fourth USENIX Conf. Object-Oriented Technologies and Systems, pp. 219-234, 1998.
[29] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," J. Software Maintenance: Research and Practice, vol. 7, no. 1, pp. 49-62, 1995.
[30] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds, "A Comparison of Methods for Locating Features in Legacy Software," J. Systems and Software, vol. 65, no. 2, pp. 105-114, 2003.
[31] N. Wilde, M. Buckellew, H. Page, and V. Rajlich, "A Case Study of Feature Location in Unstructured Legacy Fortran Code," Proc. Fifth European Conf. Software Maintenance and Reeng., pp. 68-76, 2001.
[32] N. Wilde and C. Casey, "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension," Proc. Int'l Conf. Software Maintenance, pp. 312-318, 1996.
[33] K. Koskimies and H. Mo¨ssenbo¨ ck, "Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs," Proc. 18th Int'l Conf. Software Eng., pp. 366-375, 1996.
[34] D.F. Jerding and S. Rugaber, "Using Visualization for Architectural Localization and Extraction," Proc. Fourth Working Conf. Reverse Eng., pp. 56-65, 1997.
[35] D.F. Jerding, J.T. Stasko, and T. Ball, "Visualizing Interactions in Program Executions," Proc. 19th Int'l Conf. Software Eng., pp. 360-370, 1997.
[36] R.J. Walker, G.C. Murphy, B.N. Freeman-Benson, D. Wright, D.Swanson, and J. Isaak, "Visualizing Dynamic Software System Information through High-Level Models," Proc. 13th Conf. Object-Oriented Programming Systems, Languages and Applications, pp. 271-283, 1998.
[37] T. Ball, "The Concept of Dynamic Analysis," Proc. Seventh European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng., pp. 216-234, 1999.
[38] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J.M. Vlissides,and J. Yang, "Visualizing the Execution of Java Programs," Proc.ACM 2001 Symp. Software Visualization, pp. 151-162, 2001.
[39] T. Systa¨, K. Koskimies, and H.A. Mu¨ ller, "Shimba: An Environment for Reverse Engineering Java Software Systems," Software,Practice and Experience, vol. 31, no. 4, pp. 371-394, 2001.
[40] L.C. Briand, Y. Labiche, and J. Leduc, "Toward the ReverseEngineering of UML Sequence Diagrams for Distributed Java Software," IEEE Trans. Software Eng., vol. 32, no. 9, pp. 642-663, Sept. 2006.
[41] S.P. Reiss and M. Renieris, "Encoding Program Executions," Proc.23rd Int'l Conf. Software Eng., pp. 221-230, 2001.
[42] A. Hamou-Lhadj and T.C. Lethbridge, "A Survey of Trace Exploration Tools and Techniques," Proc. 2004 Conf. the Centre for Advanced Studies on Collaborative Research, pp. 42-55, 2004.
[43] A. Hamou-Lhadj and T.C. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System," Proc. 14th Int'l Conf. Program Comprehension,pp. 181-190, 2006.
[44] A. Hamou-Lhadj, T.C. Lethbridge, and L. Fu, "Challenges and Requirements for an Effective Trace Exploration Tool," Proc. 12th Int'l Workshop Program Comprehension, pp. 70-78, 2004.
[45] O. Greevy, S. Ducasse, and T. Gırba, "Analyzing Feature Traces to Incorporate the Semantics of Change in Software Evolution Analysis," Proc. 21st Int'l Conf. Software Maintenance, pp. 347-356,2005.
[46] O. Greevy, S. Ducasse, and T. Gırba, "Analyzing Software Evolution through Feature Views," J. Software Maintenance and Evolution: Research and Practice, vol. 18, no. 6, pp. 425-456, 2006.
[47] J. Kothari, T. Denton, A. Shokoufandeh, and S. Mancoridis,"Reducing Program Comprehension Effort in Evolving Software by Recognizing Feature Implementation Convergence," Proc. 15th Int'l Conf. Program Comprehension, pp. 17-26, 2007.
[48] N. Wilde and M.C. Scully, "Software Reconnaissance: Mapping Program Features to Code," J. Software Maintenance: Research and Practice, vol. 7, no. 1, pp. 49-62, 1995.