

Reverse Engineering Behavioural Models by Filtering out Utilities from Execution Traces

Edna Braun

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Ph.D. in Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science



uOttawa

University of Ottawa
Ottawa, Ontario, Canada
September 2013

© Edna Braun, Ottawa, Canada, 2013

Abstract

An important issue in software evolution is the time and effort needed to understand existing applications. Reverse engineering software to recover behavioural models is difficult and is complicated due to the lack of a standardized way of extracting and visualizing knowledge. In this thesis, we study a technique for automatically extracting static and dynamic data from software, filtering and analysing the data, and visualizing the behavioural model of a selected feature of a software application. We also investigate the usefulness of the generated diagrams as documentation for the software.

We present a literature review of studies that have used static and dynamic data analysis for software comprehension. A set of criteria is created, and each approach, including this thesis' technique, is compared using those criteria.

We propose an approach to simplify lengthy traces by filtering out software components that are too low level to give a high-level picture of the selected feature. We use static information to identify and remove small and simple (or uncomplicated) software components from the trace. We define a *utility method* as any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program. *Utilityhood* is defined as the extent to which a particular method can be considered a utility. Utilityhood is calculated using different combinations of selected dynamic and static variables. Methods with high utilityhood values are detected and removed iteratively. By eliminating utilities, we are left with a much smaller trace which is then visualized using the Use Case Map (UCM) notation. UCM is a scenario language used to specify and explain behaviour of complex systems.

By doing so, we can identify the algorithm that generates a UCM closest to the mental model of the designers. Although when analysing our results we did not identify an algorithm that was best in all cases, there is a trend in that three of the best four algorithms (out of a total of eight algorithms investigated) used method complexity and

method lines of code in their parameters. We also validated the algorithm results by doing a comparison with a list of methods given to us by the creators of the software and doing precision and recall calculations. Seven out of the eight participants agreed or strongly agreed that using UCM diagrams to visualize reduced traces is valid approach, with none who disagreed.

Acknowledgment

I would like to thank my supervisors, Dr. Daniel Amyot and Dr. Tim Lethbridge, for their incredible patience, guidance and support. This work would not have been possible without their help. I also thank my committee members for accepting to review this work and for their valuable comments at the proposal stage of this thesis. This thesis would not have been possible without the funding provided by NSERC.

I am indebted to my fellow graduate students for having listened to my presentations and provided invaluable comments. To all the researchers, graduate students and software architects, who cannot be named (you know who you are), thank you for participating to my case studies. I would also like to express my gratitude to Nick Cartwright and ex-colleagues and fellow PhD students Mohammad Alhaj and the late Dr. Rasha Tawhid, for their thought-provoking discussions and pointers on research methodologies and data analysis techniques.

I would like to convey my appreciation to my mother Zenebech Tesfahun, my sister Dr. Ruth Abraham and my brothers Yonathan Teferawork, Isaac Abraham and Henock Abraham for all their support over the years.

Many thanks to my husband, Allen Braun, who has always been very supportive and accommodating during my studies. His comments were invaluable from a software engineering practitioner's point of view. I also thank my children Rebecca, Samuel, Naomi and Matthias, who have brought joy and purpose to my life. They have contributed positively to every aspect of my life, except maybe to the speed of the progress of my PhD studies. Finally, many thanks to Mylène Bergeron and Allen Braun for proof reading the thesis, and to Lucie Lalonde and the staff of the Académie de la Capitale for being so flexible and accommodating and for taking great care of my children while I was finishing the last years of my PhD.

Table of Contents

Abstract	i
Acknowledgment	iii
Table of Contents	iv
List of Figures.....	ix
List of Tables	xi
List of Acronyms	xiii
Chapter 1. Introduction	1
1.1. <i>Motivation</i>	3
1.2. <i>Research Hypothesis and Definitions</i>	4
1.3. <i>Research Method</i>	5
1.4. <i>Thesis Contributions</i>	7
1.4.1 Major Contributions	7
1.4.2 Minor Contributions	8
1.5. <i>Publications Based on this Thesis</i>	9
1.6. <i>Thesis Outline</i>	9
Chapter 2. Background.....	10
2.1. <i>Program Comprehension</i>	10
2.2. <i>Reverse Engineering</i>	11
2.3. <i>Program Analysis</i>	12
2.3.1 Dynamic Analysis	12
2.3.2 Static Analysis	14
2.4. <i>Program Visualization</i>	15
2.5. <i>Use Case Maps</i>	16
2.6. <i>Data Compression</i>	19
2.7. <i>Chapter Summary</i>	20
Chapter 3. Literature Review	21
3.1. <i>Evaluation Criteria for Papers on Program Comprehension</i>	21

3.2.	<i>Program Comprehension and Feature Analysis</i>	24
3.2.1	Dynamic Fan-in and Fan-out Metrics for Program Comprehension.....	25
3.2.2	Scenario-Driven Dynamic Analysis for Comprehending Large Software.....	26
3.2.3	Integrating Static and Dynamic Analysis to Improve Comprehension.....	28
3.2.4	Understanding the Behaviour of Java Programs.....	29
3.2.5	The CONCEPT Project	30
3.2.6	Mapping Program Features to Code	31
3.2.7	Aiding Program Comprehension by Static and Dynamic Feature Analysis.....	33
3.2.8	Recovering High-Level Views from Static and Dynamic Analysis	34
3.2.9	Analysis of Execution Traces for Program Comprehension	36
3.2.10	Automatic Generation of Abstract Views for Software Comprehension	37
3.2.11	Solving the Feature Location Problem by Measuring the Component Modification Impact.....	39
3.2.12	Summary of Program Comprehension and Feature Analysis Studies.....	40
3.3.	<i>Chapter Summary</i>	42
Chapter 4. Data and Methodology Used in the Empirical Studies.....		43
4.1.	<i>Design of the Study</i>	43
4.1.1	Systems Studied.....	43
4.1.2	Feature Selection.....	45
4.1.3	Tools Used to Collect Data from the Systems	46
4.1.4	Validation Participants	46
4.2.	<i>Dynamic Data Collected</i>	47
4.2.1	Gathering Execution Traces	47
4.2.2	CPU Usage	48
4.2.3	Fan-in/Fan-Out	49
4.2.4	Depth of Call	53
4.2.5	Segmentation and Clustering.....	54
4.3.	<i>Static Data Collected</i>	54
4.3.1	McCabe Cyclomatic Complexity.....	55
4.3.2	Method Lines of Code (MLOC)	57
4.4.	<i>Summarization Process</i>	58
4.5.	<i>Visualization of Reduced Traces using UCM</i>	59
4.6.	<i>Chapter Summary</i>	61
Chapter 5. Utility Detection Algorithms		62
5.1.	<i>Algorithm 1: Fan-in</i>	63
5.2.	<i>Algorithm 2: Fan-in and Fan-out</i>	64
5.3.	<i>Algorithm 3: McCabe</i>	65
5.4.	<i>Algorithm 4: MLOC</i>	67
5.5.	<i>Algorithm 5: Segment Presence</i>	68
5.6.	<i>Algorithm 6: High Fan-in and High Fan-out (Yuying)</i>	69
5.7.	<i>Algorithm 7: Average Call Tree Depth</i>	70

5.8.	<i>Algorithm 8: Calculate Utilityhood CPU</i>	72
5.9.	<i>Chapter Summary</i>	73
Chapter 6. Tool Support		74
6.1.	<i>Overview</i>	74
6.2.	<i>Inputs</i>	74
6.2.1	Execution Trace Input File	74
6.2.2	Static Data Input File	76
6.2.3	CPU Input File	79
6.2.4	Percent Distribution Input File	80
6.2.5	Exclusion Input File	82
6.2.6	Inclusion Input File	83
6.3.	<i>Output Files</i>	83
6.4.	<i>Final Product of the Tool</i>	86
6.4.1	Sample Condensed Trace File	86
6.4.2	Sample Use Case Map	87
6.4.3	Message Sequence Chart.....	89
6.5.	<i>Summary</i>	91
Chapter 7. Reverse Engineering Case Studies		92
7.1.	<i>Participants</i>	92
7.2.	<i>Empirical Study</i>	92
7.3.	<i>Case Study 1 – jUCMNav</i>	93
7.3.1	Description of Feature Selected	93
7.3.2	jUCMNav Traces	94
7.4.	<i>Case Study 2 – Umple</i>	97
7.4.1	Description of Feature Selected	98
7.4.2	Umple Traces.....	100
7.5.	<i>Case Study 3 – UCed (Use Case Editor)</i>	101
7.5.1	Description of Feature Selected	102
7.5.2	UCed Traces	102
7.6.	<i>Validity of UCM Diagrams for Behavioral Model</i>	105
7.7.	<i>Algorithm Comparison Process</i>	106
7.7.1	Precision and Recall.....	106
7.7.2	Comparison of Algorithms	112
7.7.3	Algorithm Parameter Comparison	114
7.7.4	Possible Further Improvement of the Algorithms.....	115
7.8.	<i>Summary</i>	115
Chapter 8. Case Study on Reverse-Engineered Behavioural Diagrams as Code Documentation		117
8.1.	<i>Description of the Study</i>	117

8.2.	<i>Validation of UCM Diagrams</i>	<i>119</i>
8.3.	<i>Lessons Learned</i>	<i>121</i>
8.4.	<i>Summary.....</i>	<i>123</i>
Chapter 9.	Discussion.....	124
9.1.	<i>Validation of the Trace Filtering Step</i>	<i>124</i>
9.1.1	Validation Approach	125
9.1.2	Cyclomatic Complexity of Methods	126
9.1.3	Size of Methods	127
9.1.4	Data Access	128
9.1.5	Possible Improvement of the Trace Filtering Step.....	130
9.2.	<i>Validation of the Utility Detection Step</i>	<i>131</i>
9.2.1	Which Algorithm Works Best	131
9.2.2	Precision and Recall.....	132
9.3.	<i>Validation of the Visualization Step.....</i>	<i>132</i>
9.3.1	Validation Approach	132
9.3.2	Use of UCM for Visualization.....	133
9.4.	<i>Limitations and Threats to Validity</i>	<i>134</i>
9.4.1	External Validity Threats	134
9.4.2	Internal Validity Threats	134
9.4.3	Construct Validity Threats	135
9.4.4	Conclusion Validity Threats.....	136
9.5.	<i>Summary.....</i>	<i>137</i>
Chapter 10.	Conclusion	138
10.1.	<i>Thesis Contributions</i>	<i>138</i>
10.1.1	Major Contributions.....	138
10.1.2	Minor Contributions.....	139
10.2.	<i>Comparison with Similar Work</i>	<i>140</i>
10.3.	<i>Future Work.....</i>	<i>143</i>
10.4.	<i>Closing Remarks.....</i>	<i>147</i>
References.....		148
Appendix A:	Questionnaire	157
Appendix B:	Behavioural Maps for jUCMNav.....	159
Appendix C:	Behavioural Maps for Umple.....	169
Appendix D:	Behavioural Maps UCED	178
Appendix E:	Behavioural Maps for the TraceToUCM Tool.....	188

Appendix F: Algorithms.....	197
Appendix G: jUCMNav – Open UCM Principal Diagram.....	200
Appendix H: Umple – Principal Methods for the Compile Feature.....	201
Appendix I: Data Correlation.....	203

List of Figures

Figure 1.	Filtering out utility methods.	2
Figure 2.	Summary of the research method.....	6
Figure 3.	Summary of UCM notation (from [59]).....	17
Figure 4.	A UCM diagram of a filtered JHotdraw execution trace.	18
Figure 5.	Summary of the algorithm comparison process.	59
Figure 6.	Suite of tools and their interactions.....	75
Figure 7.	Input parameter 1 – Execution trace.	76
Figure 8.	Static data extracted using Metrics 1.3.6.....	77
Figure 9.	CPU data extracted using TPTP.	80
Figure 10.	Input parameter 4 – Filtering information.	81
Figure 11.	Optional input parameter 5 – List of components that must be excluded.	82
Figure 12.	Optional input parameter 6 – List of components that must be included.....	83
Figure 13.	Clustering report – List of methods that tend to occur together.	85
Figure 14.	Output artefact 1 – A condensed trace file.	86
Figure 15.	Use case map of the TraceToUCM feature.	87
Figure 16.	A UCM diagram without stubs.....	88
Figure 17.	A UCM diagram with stub (left), with its plugin UCM (right).	89
Figure 18.	Message Sequence Diagram of the TraceToUCM feature.....	90
Figure 19.	Use case map created to exercise the “Open a UCM diagram” feature.....	94
Figure 20.	Behavioural map of jUCMNav using Algorithm 8.....	95
Figure 21.	Behavioural map of Umple using Algorithm 3.	98
Figure 22.	Program used to run Umple.	99
Figure 23.	Use case example opened using UCEd feature [111].	102
Figure 24.	Behavioural map of UCEd using Algorithm 1.	103
Figure 25.	Precision versus recall graph for algorithms used on jUCMNav methods.....	109
Figure 26.	F-Measures of the eight algorithms for jUCMNav.....	109
Figure 27.	Precision versus recall graph for algorithms used on Umple methods.	111
Figure 28.	F-Measures of the eight algorithms for Umple.....	111
Figure 29.	TraceToUCM diagram after small and simple methods are filtered.....	121
Figure 30.	Size of filtered traces, in percentage of the original size.....	124
Figure 31.	Size of filtered traces (algorithms only), in percentage of the original size.....	131
Figure 32.	Multiple layers of UCM stubs to visualize execution traces.	145
Figure 33.	UCM for TraceToUCM filtered using Algorithm 1.	188
Figure 34.	UCM for TraceToUCM filtered using Algorithm 2.	189
Figure 35.	UCM for TraceToUCM filtered using Algorithm 3.	190
Figure 36.	UCM for TraceToUCM filtered using Algorithm 4.	191
Figure 37.	UCM for TraceToUCM filtered using Algorithm 5.	192

Figure 38. UCM for TraceToUCM filtered using Algorithm 6.	193
Figure 39. UCM for TraceToUCM filtered using Algorithm 7.	194
Figure 40. UCM for TraceToUCM filtered using Algorithm 8.	195
Figure 41. UCM for TraceToUCM after small and simple methods are filtered.	196
Figure 42. Pseudocode for dynamic and static data processing.	198
Figure 43. Class description for TraceItem.	199

List of Tables

Table 1	Summary scores for reviewed papers.....	41
Table 2	List of software systems studied.	44
Table 3	List of dynamic data derived from an execution trace.	48
Table 4	Correlation of Umple fan-in score with other static and dynamic variables.....	50
Table 5	Correlation of static and dynamic variables with fan-in, for all systems.	51
Table 6	Correlation of static and dynamic variables with fan-out, for all systems.	52
Table 7	Correlation of static and dynamic variables with average call hierarchy, for all systems.	53
Table 8	Static data collected by using Metrics 1.3.6.	55
Table 9	Correlation of static and dynamic variables with McCabe, for all systems.	56
Table 10	Correlation of static and dynamic variables with MLOC, for all systems.	58
Table 11	Mapping of trace elements to UCM notation elements.....	60
Table 12	Types of parameters collected and used for the algorithms.	62
Table 13	Summary of the eight algorithms.....	63
Table 14	Trace segmentation table.	69
Table 15	Input parameter 2 – List of static data collected using Metrics 1.3.6.	79
Table 16	jUCMNav execution trace line counts.	96
Table 17	Summary of responses from jUCMNav designers (algorithms’ selection of methods).	96
Table 18	Summary of responses from jUCMNav designers (UCM diagram comparison).	97
Table 19	Umple execution trace line counts.	100
Table 20	Summary of responses from Umple designers (algorithms’ selection of methods)	101
Table 21	Summary of responses for Umple designers (UCM diagram comparison).	101
Table 22	UCEd execution trace line count.	104
Table 23	Summary of responses from UCEd designers (for each algorithm’s selection of methods).	104
Table 24	Summary of responses from UCEd designers (UCM diagram comparison).	105
Table 25	Participants’ answers to part II of the questionnaire.....	105
Table 26	Recall values of the eight algorithms at selecting jUCMNav’s main methods.....	107

Table 27	Precision and Recall values of the eight algorithms at selecting jUCMNav's main methods.	108
Table 28	Recall values of the eight algorithms at selecting Umple's main methods.....	110
Table 29	Precision and Recall values of the eight algorithms at selecting Umple's main methods.	110
Table 30	Summary of responses from all participants (algorithm's selection of methods).	113
Table 31	Summary of parameters used for each of the eight algorithms.	113
Table 32	First choice of algorithms by the three sets of designers.	114
Table 33	Top 3 choices of algorithms by the 3 sets of designers.....	114
Table 34	TraceToUCM execution trace line count.	118
Table 35	Summary of results: UCM visualized trace as documentation.....	119
Table 36	Data table (O1) with static and dynamic data on all methods in a trace.	126
Table 37	Correlation of static and dynamic variables with McCabe.....	127
Table 38	Correlation of static and dynamic variables with MLOC.	128
Table 39	Correlation of static and dynamic variables with data access methods.	129
Table 40	Data table for methods in jUCMNav trace.....	130
Table 41	Replies of participants for part II of the questionnaire.....	133
Table 42	Comparison of thesis methodology with reviewed papers.....	141
Table 43	Number of lines in trace files.....	145
Table 44	Correlation of static and dynamic variables with leaf.....	204
Table 45	Correlation of static and dynamic variables with AverageCallHierarchy....	205
Table 46	Correlation of static and dynamic variables with DataAccessMethod.	206
Table 47	Correlation of static and dynamic variables with AverageCalledByDistance.....	207
Table 48	Correlation of static and dynamic variables with averageCalledByDistanceBetweenMethodsCalled.....	208
Table 49	Correlation of static and dynamic variables with EncounteredinTrace.	209
Table 50	Correlation of static and dynamic variables with fan-in.	210
Table 51	Correlation of static and dynamic variables with fan-out.	211
Table 52	Correlation of static and dynamic variables with McCabe.....	212
Table 53	Correlation of static and dynamic variables with MLOC.	213

List of Acronyms

Acronym	Definition
BCP	Built Client Pages
BOON	Buffer Overrun detection
CA	Afferent Coupling
CE	Efferent Coupling
CONCEPT	Comprehension Of Net-Centered Programs and Techniques
CPU	Central Processing Unit
CTF	Compact Trace Format
DIT	Depth of Inheritance Tree
GRL	Goal-oriented Requirement Language
jUCMNav	Java Use Case Map Navigator
LIP	Linear Independent Path
LOC	Lines Of Code
MLOC	Method Lines of Code
MSC	Message Sequence Chart
NBD	Nested Block Depth
NOC	Number of Classes
NOF	Number of Attributes
NOI	Number of Interfaces
NOM	Number of Methods
NOP	Number of Packages
NORM	Number of Overridden Methods
NSC	Number of Children
NSF	Number of Static Attributes
NSM	Number of Static Methods
PAR	Number of Parameters
RMD	Normalized Distance
RMI	Instability
SAAT	Software Architecture Analysis Tool
SDK	Software Development Kit
SIX	Specialization Index
STOBO	Systematic Testing of Buffer Overflows
TPTP	Test & Performance Tools Platform
UCEd	Use Case Editor

UCM	Use Case Map
UML	Unified Modeling Language
URN	User Requirements Notation
VG	McCabe Cyclomatic Complexity
WA	Web Application
WANDA	Web ApplicationNs Dynamic Analyzer
WARE	Reverse Engineering Web Application
WebSSARI	Web application Security by static Analysis and Runtime Inspection
WMC	Weighted Methods per Class

Chapter 1. Introduction

One of the main issues in software maintenance is the time and effort needed to understand software. Understanding legacy software is often very difficult due to incomplete, outdated, or non-existent documentation. One frequently needs to extract structural and behavioural information directly from the code. A number of techniques and methods have been developed in order to facilitate program comprehension [19][71][80], and several others are listed in a survey by Cornelissen et al. [24]. We propose to contribute a solution to this problem by developing algorithms to recover the behavioural model of a software system from a combination of dynamic analysis and static analysis of the code. We chose to collect both static and dynamic information to get two different and complementary perspectives. The dynamic description of a program differs from the static description of the source code due to runtime behaviours such as dynamic binding, extending emergent behaviour, and usage patterns in object-oriented applications [114]. We use a combination of these two perspectives to reverse engineer a software system with the goal of understanding it.

Reverse engineering software to recover behavioural models is a difficult task and it is further complicated by the lack of a standardized way of representing and visualizing the extracted knowledge. Such knowledge needs to be represented in an intuitive and concise manner so that software engineers can effectively perform their maintenance and modernization tasks with minimal effort.

In order to build a behavioural model of a software system that contains only the important components for the feature being studied, we need a way of filtering out the irrelevant code, leaving behind only the code that is relevant for that feature. Execution traces provide a list of the relevant code statements invoked during an execution. However, for many features, the traces run from tens of thousands to millions of steps and hence are virtually impossible to effectively read. Furthermore, traces are too detailed to provide a high-level understanding of the feature before the programmer dives into the

source code. Therefore, this thesis aims to develop a set of algorithms to reverse engineer a very high level behavioural model of the software for the specific feature the programmer is interested in. We propose to iteratively filter out details to provide the user with a high level visual abstraction of the code as shown in Figure 1. The abstracted diagram can be used as documentation.

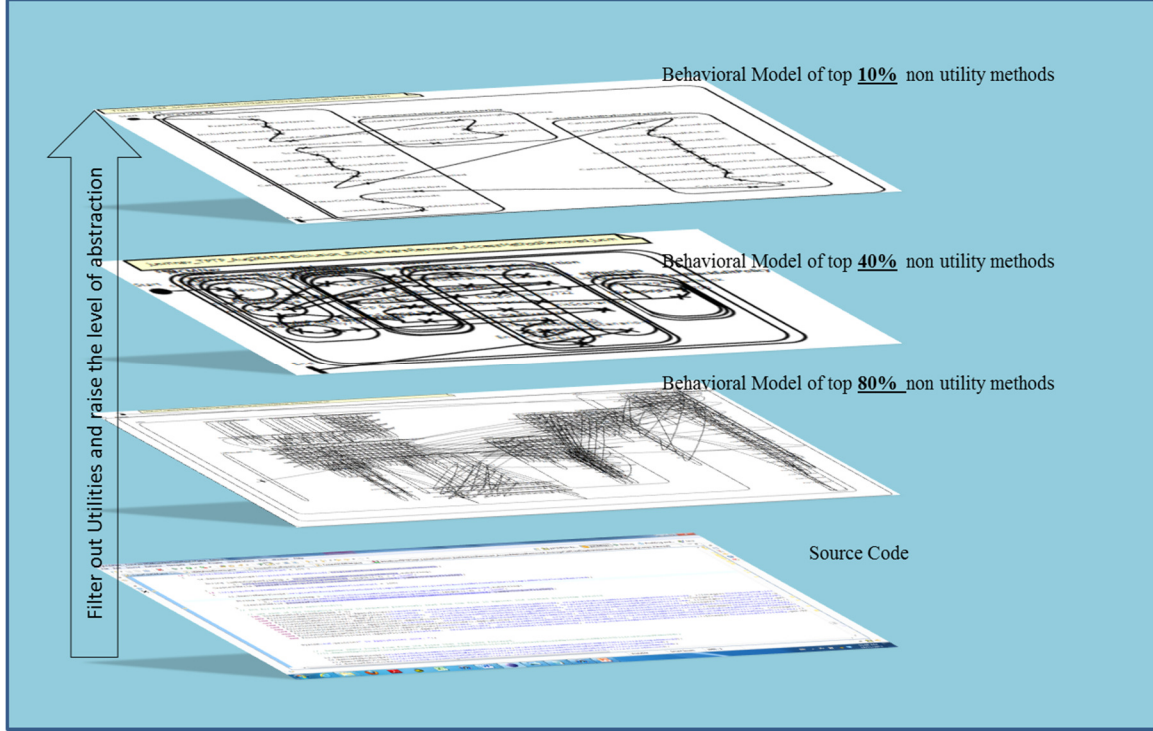


Figure 1. Filtering out utility methods.

We propose a method to automatically detect and exclude classes and methods that are *not* the main players for the feature being run, in order to simplify execution traces. To identify such *utilities*, we develop algorithms that use a combination of static data, mainly to measure method complexity, and dynamic data to measure method interactions. We use different combinations of the parameters to develop six different algorithms to calculate *utilityhood* values for each method in the execution traces. We then apply the algorithms on a set of systems and validate the results.

In an initial iteration of this work [47], we summarized lengthy execution traces from TConfig [116] and visualize the trace. TConfig is a Java application used to generate the minimum number of test configurations covering component interactions of degree n , where n is defined by the user. We filtered out low-level software components

while preserving the essence of the trace, and validated the resulting UCM (generated manually) with TConfig’s developer.

We improve on the techniques developed earlier to reduce the trace further and also increase the confidence that important methods are not filtered out by adding anti-filters. We also make the approach more user-friendly by completely automating the trace processing and generation of UCM diagrams. We develop four new algorithms using dynamic and static analysis parameters to better detect utilities. To visualize the reduced traces, we selected the Use Case Map (UCM) notation [18][59], a scenario language used to specify and explain emergent behaviour of complex and dynamic systems. The behavioural model is then displayed as UCM diagrams.

1.1. Motivation

Ultimately, the main motivation of this research is to save money for software development teams and organizations by reducing the time and effort spent on program comprehension. Software maintenance is an expensive and high demanding process [75][88]. Key activities that contribute to the high costs of maintenance include understanding the code [20]. It is estimated that the relative cost for maintaining software and managing its evolution now can represent more than 90% of its total cost [65]. We are motivated to contribute to the field of software comprehension to ease the burden software developers face on their day-to-day maintenance tasks.

It is very rare for a new programmer to be given a requirement for a software product and to be asked to implement a solution from scratch. It is much more common that a programmer has to modify existing code to add a feature, fix a bug, or enhance an existing feature. Such maintenance activities are even the topic of an international standard, namely ISO/IEC 14764 [56]. Corporate computer programmers now spend about 80% of their time just repairing software and updating it to keep it running [23]. In other words, a programmer is much more likely to be involved in software evolution and software maintenance than in new, green-field software development.

A programmer needs to understand the system to maintain it. Rajlich and Wilde state that “*software that is not comprehended cannot be changed*” [93], which evokes that program comprehension is an essential part of software evolution and software

maintenance. Most legacy systems are large and not well documented. Luckily, the new programmer rarely needs to understand the whole system. The programmer needs to understand the feature he or she will work on before starting to change the code, adopting an as-needed comprehension strategy [66] focusing only on portions of the program that are relevant to one feature at a time. He or she also needs to make sure the changed code does not have unintended consequences. Corbi [23] summarizes the role of software developers as follows: “*Programmers have become part historian, part detective, and part clairvoyant*”. To get a high-level understanding of a software feature, the programmer needs to zero in on as few key modules (e.g., classes in object-oriented systems) as possible. In order to isolate the key classes, he/she needs to eliminate low-level methods and classes in order to be left with only the main methods and classes.

1.2. Research Hypothesis and Definitions

We hypothesize that hiding program components (e.g., methods) that are used to perform low-level *utility* functions will simplify an execution trace without losing significant meaning. Our use of the term *execution trace* is the same way as feature-trace, which Greevy et al. define as “an execution trace captured as a result of triggering a feature” [43]. An execution trace, or feature-trace, consists of a sequence of method calls. As a convention, we will use the term execution trace throughout the document.

A *feature* is defined as a realized functional requirement of a system that can be triggered by a user, as explained by Eisenbarth et al. [36]. It is hence an observable unit of behavior of a system triggered by the user. In our case, a feature can also be a test case in a debugging context. We hypothesize that by filtering out utility methods from execution traces, we can keep only the top non-utility methods. Those non-utility methods can be used to describe the feature behaviour of the system and hence filtering the trace would not result in loss of meaning for that trace. The *meaning of a trace* is defined as the understanding one would get from reading a full execution trace of a feature.

We also hypothesize that those utility methods are designed to be accessed from multiple places and tend to be simple single-purposed methods, which means they should be of relatively low complexity. The methods would also be smaller (low line count), in relation to other methods in the same system.

We define a *utility method* as any element of a program designed for the convenience of the designer and implementer, and intended to be accessed from multiple places within a certain scope of the program. Utilities may or may not be designed to be reused in multiple programs.

1.3. Research Method

This thesis starts with scoping of the problem, reverse engineering of behaviour models for program comprehension. We then look through the literature to find publications that address program comprehension by using static and dynamic data analysis. We define a set of criteria that are used to evaluate the publications in the literature and our results. We make use of empirical studies that are performed iteratively. A small case study is done to decide which static and dynamic variables should be collected from the code. We collected 11 dynamic and 23 static parameters (including fan-in, fan-out, McCabe complexity index [81], size of methods, and depth of nested blocks). We then run a statistical analysis on the collected data to help us decide which of the collected data to use in order to locate utility methods. Using various combinations of the selected parameters, we develop six algorithms to calculate a utilityhood matrix for the methods in the programs.

For this first iteration, we run a few small programs written in Java. We validate the results of the first study, by manually inspecting the code of the studied programs, to check if the methods identified as utilities are indeed utilities. The main goal of the first iteration is to improve the design of the experiment. The results are used to fine-tune the experiment.

For the second iteration of the experiment we studied three open source software systems. The first application is jUCMNav [3][62], which is a graphical editor for the User Requirements Notation (URN) [2][59]. The second application is Umple [11][12][41][74], which is a modeling tool and programming language family that enables Model-Oriented Programming. The third application is UCed [111], which is an environment for use-case-based requirements engineering. The results of the second iteration of study are validated by showing the designers of each system the UCM behavioural models that are produced using different algorithms and asking them if the diagram represent their mental model of the feature run. The questionnaire asks two questions for

each UCM diagram a) whether we have successfully identified the utilities, and b) whether the behavioural diagrams derived from reduced traces resemble their mental pictures of the system.

Problem Definition	• Define the problem • Define objectives and goals of the thesis • Discuss the importance and impact of the problem
Initial Literature Review	• Create evaluation criteria for software comprehension methodologies. • Search for publications that have used static and dynamic data analysis for software comprehension. • Criticize the papers using the criteria developed.
Characterization of Utilities	• Run several small programs and collect static and dynamic data. • Study the data collected and find variables that can be used to identify utilities. • Create algorithms using different combinations of the selected variables. • Run the algorithm to filter execution trace. • Validate the results by inspecting the code.
Reverse Engineering Case Studies	• Repeat the experiment on larger software (jUCMNav, UCed and Umple) • Run one feature of the programs, collect static and dynamic data, analyze, and calculate utilityhood of each method called in the feature. • Generate UCM diagrams and validate the results with the designers.
Software Documentation Case Study (TraceToUCM)	• Evaluate the use of reduced execution traces, visualized as UCM diagrams, to document a feature. Run the tool TraceToUCM through itself and generate UCM diagrams. • Validate the result by asking designers and graduate students who are not familiar with the tool to describe the system based on the diagrams.
Analysis, Discussion and Conclusion	• Analyze the results from the case studies. • Compare findings with other works reviewed in chapter 3 using the criteria developed. • Compare the thesis' algorithms with other algorithms in the literature. • Discuss findings, threats and limitations.

Figure 2. Summary of the research method.

We did a separate experiment to evaluate the use of reduced execution trace, visualized as UCM diagram to document a feature. To test the theory, we reverse engineered TraceToUCM, a tool we developed to implement the methodology described in this thesis, and

showed the UCM diagrams to many subjects (designers and graduate students). We asked them to look at the UCM diagrams and to describe the feature depicted in the diagram.

1.4. Thesis Contributions

To be successful, software maintenance requires efficient program comprehension techniques. In this research, we have used a combination of static and dynamic analysis techniques to make program comprehension simpler. We have grouped the contributions made in this research into two groups: major and minor contributions. In this section, the contributions of this research are highlighted, together with their corresponding chapters.

1.4.1 Major Contributions

1. A framework that can be used to compare different methodologies that use trace analysis. This framework has a list of criteria that can be used to evaluate program comprehension and methodologies (Chapter 3). These can be reused by academic researchers for similar studies.
2. A methodology and tool for analyzing static and dynamic parameters of object-oriented software (Chapter 4). This includes:
 - An approach to reduce large amounts of data contained in execution traces and to acquire sufficient knowledge about the system under study from the trace;
 - An approach to organize and collate the large amount of data contained in source code, execution trace and profiling data to understand the software by identifying crucial program artefacts. The approach can be used by software designers who want to characterize how methods are being used in the system they are working on by looking at one or all of the artefacts produced by the tool (Section 6.3). The methodology can also be used by academic researchers to compare new algorithms to each other.
 - New automated tool support (Chapter 6);
3. An approach and a tool to visualize software execution traces. This includes:
 - A mapping of software execution elements to UCM elements, UCM being a standard scenario notation (Chapter 4, Section 4.5). The mapping can be

used by software designers to transform trace components to UCM diagrams.

- A tool that can be used to do the mapping and UCM diagram generation automatically (TraceToUCM). The user can use the UCM diagrams directly. Alternatively, the user can transform UCM diagrams to Message Sequence Charts (MSC) by using a pre-existing function in the UCM editor jUCMNav.

1.4.2 Minor Contributions

This thesis also includes the following minor contributions:

1. An analysis of the validity of using both static and dynamic data to better understand large software systems (Chapter 2).
2. A literature review on studies that used static and dynamic data analysis for software comprehension (Chapter 3).
 - We summarized and compared studies that adopt the strengths of the two data analysis types and minimize the effect of their weaknesses.
3. An improved understanding of the use of utility filtering for software comprehension. This is demonstrated through:
 - The development of effective algorithms for detecting utilities (Chapter 5);
 - Case studies used to illustrate and validate these algorithms for reverse engineering (Chapter 7);
 - Case studies used to illustrate and validate the use of the UCM diagrams as documentation (Chapter 8);
 - A more precise definition and characterization of utility classes and utility methods (Chapter 7, Chapter 8 and Chapter 9); and
 - A small, validated set of dynamic and static parameters that can be used to identify utilities: size of a method, complexity of a method, and to lesser extend nested block depth of a method. These can be reused by academic researchers to further characterize utilities (Chapter 7, Chapter 8 and Chapter 9).

1.5. Publications Based on this Thesis

There is one publication based on this research so far, which has been well cited (74 times as of August 2013). This paper covers essentially a first iteration of the work presented in this thesis.. I am second author but contributed equally to the first author and provided the core contributions related to UCM visualization and validation, which are further expanded in this thesis.

Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T.: Recovering Behavioral Design Models from Execution Traces. *9th European Conference on Software Maintenance and Reengineering (CSMR)*, Manchester, UK, March 2005. IEEE Computer Society, 112-121.

In this thesis we improve on the algorithm presented in this paper by automating the trace processing and generation of UCM diagrams. We also compare the algorithm with four new algorithms that we have developed and three algorithms developed by other authors.

1.6. Thesis Outline

Chapter 1 outlined the motivation, research hypothesis and research method. In the remainder of this work, Chapter 2 provides definitions of terms and a brief summary of the background knowledge used in this proposal. Chapter 3 presents a literature review of related approaches that use a combination of static and dynamic analysis. Chapter 4 gives a detailed description of the design of the experiment and the approach taken to summarize execution trace and visualize them with UCM diagrams. Chapter 5 covers the algorithms developed to detect utilities In Chapter 6, we discuss the tool developed to filter out traces and generate a behavioural map. In Chapter 7 and Chapter 8, we present the case studies. In Chapter 9, we discuss the results of the empirical studies, the methods used to validate them, and threats to validity. Finally, Chapter 10 concludes the thesis with a discussion of the results and with a section dedicated to future work.

Chapter 2. Background

In this chapter, we present definitions and a brief description of software engineering topics that are important to understand this thesis. We also discuss how the field is relevant to the problem we are studying. This thesis touches on six software engineering topics: program comprehension, reverse engineering, dynamic/static program analysis, program visualization, scenario modeling with Use Case Maps, and data compression.

2.1. Program Comprehension

Program comprehension, or program understanding, is a research area devoted to developing tools and methodologies to aid in the understanding and management of the increasing number of legacy systems. It includes the processes used by software engineers to understand software, and code analysis tools developed for the purpose of understanding programs.

Program comprehension is arguably the most important activity in software engineering. There is a large amount of software that is already written and deployed, and it takes a lot of effort to maintain and evolve this legacy software. In order to successfully carry out maintenance tasks, the target system, or at least part of it, must first be understood [38]. A significant percentage of professional software engineers' time is spent maintaining existing software [64]. Therefore, software maintenance represents a large and growing expense for organizations [65].

It takes a significant amount of time to analyse and understand software. However, software designers routinely have to make changes to solve bugs without having the time to make an in-depth analysis of how those changes will impact other features. Parnas summarizes the problem as follows:

“Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. After many such

changes, the original designers no longer understand the product. Those who made the changes never did. In other words, nobody understands the modified product” [89].

Software modification leads to a phenomenon called *software aging*, which means that program comprehension is an important and vital part of the software industry today and in the future, because not modifying a software system is not an option if the software is to continue to exist. The first of Lehman’s laws of evolution [73] states that since the environment of the real world is constantly changing, systems that are a component of real-world processes must evolve as the world changes or face becoming progressively less useful.

In this thesis, we are interested in understanding software. This understanding of a feature is facilitated by a field known as reverse engineering, defined in Section 2.2.

2.2. Reverse Engineering

Reverse engineering, also known as design recovery or architecture recovery, is a study of how a system is constructed. The term “reverse engineering” has its origin in the analysis of hardware, where the practice of reconstructing designs from finished products is common [22]. When applied to software, reverse engineering is the process of identifying and extracting higher-level abstractions from existing software systems [42]. The process of reverse engineering does not modify the software system; however, additional knowledge about the system is produced. This process can be decomposed into three parts: data gathering, knowledge management, and information exploration [108].

1. *Data gathering* is the collection of raw data required to identify the artefacts and relationships of a system and to use them to later construct higher-level abstractions.
2. *Knowledge management* is the capture, organization, understanding, and extension of past experiences, processes, and individual know-how within a corporation.

3. *Information exploration* is the navigation, analysis, and presentation of information.

The primary goal of reverse engineering is to convert the existing code into a more abstract and understandable representation to increase the overall comprehensibility of the system. Müller *et al.* summarize reverse engineering as focusing on “*understanding the code*” [85]. The need to understand the code could be for the purpose of system evolution, documentation, redesign, porting, or software comprehension as stated in the previous section. The term “software evolution” stems from a series of works, commonly referred to as Lehman’s laws, which was first proposed in 1980 [73]. It refers to the study and management of the process of updating software over time after its initial development. Software evolution comprises development, maintenance and reverse engineering activities.

Reverse engineering or architecture recovery techniques [25][28] have five typical and recognized elements: using human experts, recognizing known patterns, static and dynamic analysis, clustering and data mining, and comparing products within a family.

In this thesis, we are interested in using and improving reverse engineering techniques to facilitate program comprehension. We have chosen to focus on static and dynamic analysis, given its high potential for automation, to gather data from the software. We use human experts to validate our findings. Section 2.3 looks at dynamic and static analysis in more depth.

2.3. Program Analysis

The overall aim of program analysis is to identify and inspect the subject programs. Program analysis techniques are classified into two main parts, namely static analysis and dynamic analysis. In this section, we define both analysis techniques.

2.3.1 Dynamic Analysis

Thomas Ball defines dynamic program analysis as “the analysis of a running program” [9]. More precisely it is the post-mortem analysis of artefacts produced by a run-

ning program. For the analysis of dynamic analysis artefacts to be effective, the target program must be run with representative input to produce interesting behaviour. “It derives properties that hold for one or more executions by examination of the running program through program instrumentation” [9]. While dynamic analysis cannot prove that a program satisfies a particular property, it can be used to detect violations of properties, as well as to provide useful information about the behaviour of programs for the specific input parameters that are entered [9].

Dynamic analysis has two properties: precision of information and dependency on program inputs.

- *Precision of information:* dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its runtime state. This instrumentation can be tuned to collect precisely the information needed to address a particular problem. An example would be the analysis of the shape of data structures created by a program.
- *Dependency on program inputs:* the very same thing that makes dynamic analysis incomplete also provides a powerful mechanism for relating program inputs and outputs to program behaviour [93]. With dynamic analysis, it is possible to relate changes in program inputs to changes in internal program behaviour and program outputs; this is because all are observable and linked by the program execution [9].

The analysis of data gathered from a running program has the potential to provide an accurate picture of a software system because it exposes the system’s actual behaviour [24]. This picture can range from class-level details, up to high-level architectural views [94] [101][112]. Dynamic analysis is precise with regard to the actual behaviour of the software system. It shows exactly what software parts were executed and in what order [10].

Dynamic analysis has the disadvantages of inherent incompleteness, scalability, and observer effect. The observer effect is the phenomenon in which software can act differently when under observation (e.g., because of instrumentation), which might pose a problem particularly in multi-threaded or multi-process software [1].

Using dynamic analysis, one can limit the area of the software being studied by executing one feature with specific input variables at a time. This enables engineers to

study only one feature unencumbered by other classes of the system. It is this important characteristic of dynamic analysis that we use in this thesis to focus on one feature at a time.

We chose to use dynamic analysis because the first executed scenario acts like a program slice, and as such it limits our scope of investigation. Secondly, dynamic information is always precise with respect to the executed scenario.

2.3.2 Static Analysis

Static analysis examines a program's text to derive properties that hold for all executions [9]. The static analysis of computer software is performed without actually executing programs. In most cases, the analysis is performed on the source code. Although static analysis can be done manually, like code inspections, it is often automated by using software tools. The term static analysis is traditionally applied to the analysis performed by an automated tool. Static analysis inspects a representation (e.g., abstract syntax tree, model, etc.) derived from the analyzed program's source code.

The main objective of static analysis is to predict how the analyzed program will behave at runtime [102]. In some cases, like in object-oriented programming languages, it is difficult to ascertain in static analysis what software parts will be executed, and in what order, because of advanced language features like dynamic binding and polymorphism.

The sophistication of the analysis performed by tools varies from those that only consider the behaviour of individual statements and declarations, to those that include the complete source code of a program in their analysis. Likewise, uses of the information obtained from the analysis also vary from collecting various measures of software at the package, class, or method level (e.g., Metrics 1.3.6 [53]), to highlighting suspicious and non-portable source code that could cause bugs (e.g., lint [61]), to formal methods that analyze software through the use of rigorous mathematical methods. The mathematical techniques used include denotational semantics, axiomatic semantics, operational semantics, and abstract interpretation. Static analysis is also used in verification of properties of software found in safety-critical systems, in order to locate potentially vulnerable code.

Static analysis can be used to check the code before the program is executed. In this thesis, we take advantage of a static analysis tool, Metrics 1.3.6, to gather infor-

mation before the code is executed. We use this information to systematically reduce the large amount of data produced using dynamic analysis techniques.

Information that has been gathered in program analysis is typically displayed using program visualization techniques, as described in Section 2.4.

2.4. Program Visualization

Price defines software visualization as “... the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software” [14][90]. Ball summarizes the use of visualization to “see” programs as follows:

“Software is intangible, having no physical shape or size. After it is written, code “disappears” into files kept on disks. Software Visualization tools use graphical techniques to make software visible through the display of programs, program artifacts, and program behavior” [10].

The viewer is an active participant as he organizes and interprets visual data according to a set of cognitive principles guided by an overall principle of cognitive efficiency as a means for circumventing limitations of the human mind and brain [43][84]. Exton states that “*visualization takes advantage of the cognitive predisposition that people have to understand information visually*” [40].

Software visualization can be used at all levels of expertise. Effective software visualizations could provide measurable benefits in program comprehension [54]. Colours, shapes and text are used to provide the user with a model of the software. This can be done at various levels of abstraction. Eisenstadt *et al.* describe the use of visualization for beginners as follows “*Software Visualization allows beginners to see the innards of program execution clearly and at the same time allows experts to view high level program abstractions which help them home in quickly on buggy code*” [37].

In software development, a major difficulty comes from the intrinsic complexity of software systems. In addition, the size of production software systems can easily reach millions of lines of code.

The overall goal of program visualization is to reduce complexity; specifically, it is trying to find simplicity in a complex artefact. Software visualization tools are used to produce a selective representation of a complex abstraction.

A first attempt at using UCMs for program understanding and visualization is presented in [4]. The authors based their approach on the static generation of scenarios from manually tagged elements in the code. The manual tagging of elements is error prone and cumbersome. We try to improve on the earlier approach by eliminating the manual tagging.

In this thesis, we are interested in visualization of traces. This is because we need a way of visualizing the execution trace derived from dynamic analysis of the data superimposed on the code structure we derived from the static analysis. To visualize the reduced traces, we selected the Use Case Map (UCM) notation [59], which can be used to visualize scenarios superimposed on a structure of components. We present a short definition of UCM and its notation elements in Section 2.5.

2.5. Use Case Maps

UCM is part of the User Requirements Notation (URN), an international standard used for the elicitation, analysis, specification, and validation of requirements. URN combines two complementary views: the Goal-oriented Requirement Language (GRL) for goals, and the Use Case Map notation for scenarios [59].

UCM model scenarios can express various relationships between scenario responsibilities, including sequencing, alternatives, concurrency, and decomposition. Responsibilities can also be assigned to a structure of components. The UCM notation is designed to represent scenarios at a high level of abstraction. It is this quality of UCM that we believe makes it an interesting notation to visualize the path of a feature through software components.

The UCM notation can be used to represent execution traces in an intuitive and concise manner, so that software engineers can effectively view and understand the software with minimal effort. Figure 3 gives a summary of the main elements of the UCM notation. A more detailed explanation of how the UCM elements map to trace elements map is given in Section 4.5.

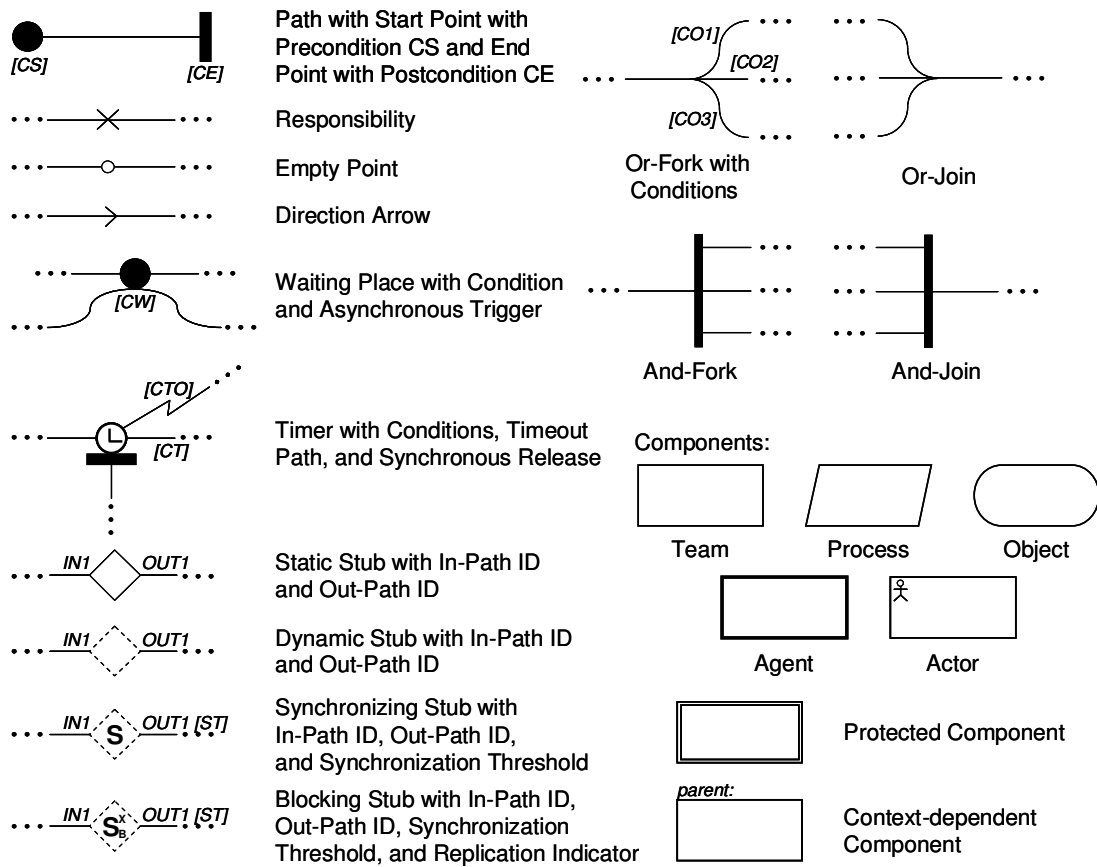


Figure 3. Summary of UCM notation (from [59]).

Figure 4 shows an example of a UCM used to represent a filtered trace of a JHotDraw [60] feature. The structure of components used in this example reflects the package structure of the application, and only the packages and classes involved in that particular scenario are displayed.

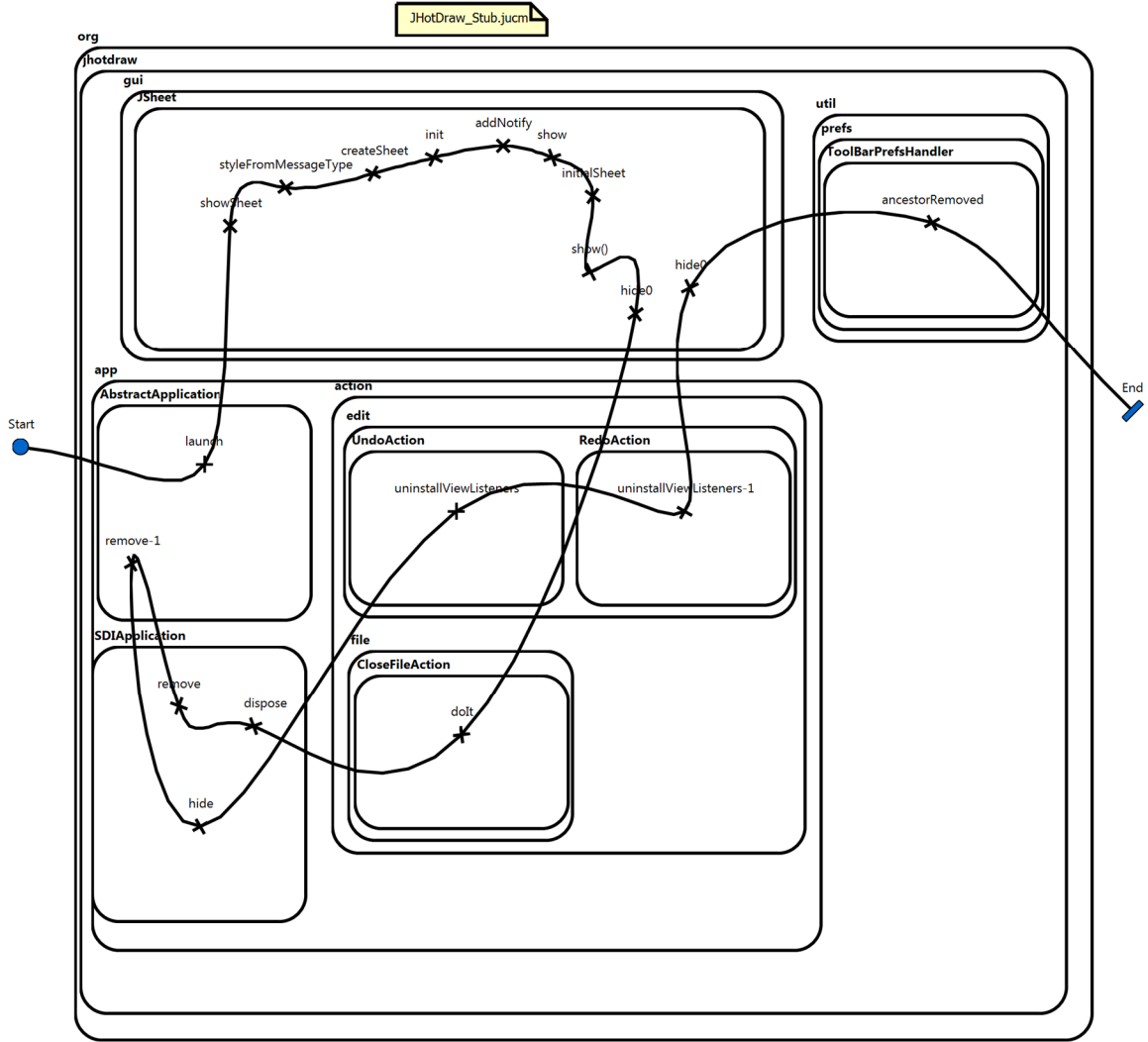


Figure 4. A UCM diagram of a filtered JHotdraw execution trace.

In this thesis, we are interested in visualizing an execution trace of the flow of one feature as it is executed, including a sequence of calls made to various methods and classes. We use UCM to create a “map” that can be used to quickly grasp the behaviour of the feature, by showing the code structures it passes through at a high level of abstraction. Only a subset of the code components will be included in the map. The challenge then is which code components to include and which ones to leave out. Amyot *et al.* [2] proposed, in 2002, the use of UCM for program visualization, based on an interactive architecture extraction exploiting static analysis tools. In this thesis, we expand substantially on this idea, by integrating dynamic analysis in the process, by providing a rigorous mapping between trace elements and UCM notation elements, and by suggesting ways to automate

the generation of visual representations (hence removing the need for manual, interactive interventions).

2.6. Data Compression

We gather dynamic data in the form of execution traces. Execution traces from interesting applications tend to be very large. For example, collecting just one byte of information per executed instruction generates on the order of a gigabyte of data per second of CPU time on a high-end microprocessor [19]. It is then necessary to compress the data in the trace files.

Compression is the process of encoding information using fewer bits than the original representation would use. Most compression algorithms are based on the principle of compressing data by removing redundancy [100]. The main reason to compress data is to consume smaller amounts of resources like hard disk space or transmission bandwidth. In our case, the main reason for compressing execution traces is usability. We need to have the representation (visualization) fit in an easily browsable visual space, such as a few pages at most.

There are two types of compression techniques: lossy and lossless compression. Lossy compression is a data encoding method that compresses data, by discarding (losing) some of it to minimize the amount of data that need to be stored. It allows an approximation of the original data to be reconstructed when the file is decompressed, in exchange for better compression rates.

Lossless data compression compresses data with no loss of information. It allows the exact original data to be reconstructed exactly from the compressed data.

In addition to being lossy or lossless, trace algorithms can have many other attributes such as a good compression rate, a fast compression speed, and a fast decompression speed.

For our purpose, we do not need to reconstruct the trace after compression, so the algorithms developed in this thesis are lossy. The most important attributes we try to maximize are retention of meaningful data and good compression rate.

2.7. Chapter Summary

This chapter has defined key terms that are used in the thesis. In addition, we have briefly described how six important software engineering fields are related to the thesis. The next chapter will be dedicated to the review and evaluation of scientific work related to our proposal. We will look at major studies that have been done, using the same techniques as this thesis, namely a combination of static and dynamic analysis.

Chapter 3. Literature Review

In this chapter, we review existing research in software engineering that used a combination of static and dynamic analysis. Several studies have been designed to maximize the complementary strength and weaknesses of static and dynamic analysis for various software fields including software vulnerabilities [1][56][65][119], software porting [5], software clustering [90], feature analysis [6][7][32][35][66][96][97][98][112], software comprehension [26][31][46][94][95][105][106][115][118] and also software architecture [70]. We will take closer look at studies that focused on software comprehension and the related field for feature analysis.

In Section 3.1, we list and explain the criteria we use to assess the papers dealing with software comprehension. In Section 3.2, we look at the papers on software comprehension and feature analysis and evaluate them against our criteria.

3.1. Evaluation Criteria for Papers on Program Comprehension

Software comprehension is a research area devoted to developing tools and methodologies to aid in the understanding of legacy systems. In this section, we list and define seven important characteristics and eight measurable criteria that we believe contribute positively to the comprehension of large software applications, with an emphasis on the comprehension of individual features. We will assign a score of 0 (does not satisfy), 1 (partially satisfies), or 2 (satisfies) for each criterion to derive an overall effectiveness score for each of the software comprehension solutions that use static and dynamic analysis. The scoring is done using information available in each paper being reviewed. In some situations, due to limited information, we have to extrapolate the answer. It is worth noting that most often there is space limitation, and authors put limited information in papers, which may lead to a lower scoring of the work.

A score of 0 indicates that either the criterion is not met or it is unclear from the paper if the criterion is met.

Scalability: Scalability is the ability of a system, network, or process, to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate the growth [15]. It refers to how well the proposed solution works when the size of the problem increases. The increase can be in the size of the software being studied or the size of the execution trace.

Criterion 1: Is the solution scalable with respect to the size of the software system?

Possible scores:

(0) Not scalable.

(1) Design appears scalable but this is not demonstrated.

(2) The authors have demonstrated the solution/tool to be scalable.

Scalability for trace size: Scalability for trace size refers to how well the solution can deal with large traces, and whether or not a trace reduction element is included in the solution.

Criterion 2: Is the solution scalable with respect to the size of the execution trace?

Possible scores:

(0) Not scalable.

(1) Design appears scalable but this is not demonstrated.

(2) The authors have demonstrated the solution/tool to be scalable.

Visualization: To be able to understand easily the abstracted model of the source code, it should be visualized. “Software visualization is the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software” [14][90]. We feel that software comprehension tools need to offer the ability to use visualization techniques to represent the available data.

Criterion 3: Is the model visualized?

Possible scores:

- (0) Not visualized.*
- (1) Design lends itself to visualization.*
- (2) The authors have presented a visualization model.*

Level of prior knowledge required: Since the goal of the study is to understand the software being studied, prior knowledge of the software should not be required to use the tool/solution. The user should ideally not have to know the source code.

Criterion 4: Is knowledge of source code required?

Possible scores:

- (0) The user has to know the source code well.*
- (1) Some prior knowledge of the source code is required.*
- (2) No prior knowledge is required.*

Existing validation: As the goal of the study is feature comprehension, the solution should be validated to see whether the model produced is easier to understand for someone who is not familiar with the system as compared to reading the source code.

Criterion 5: Has the method been validated?

Possible scores:

- (0) Not validated.*
- (1) Validated by the authors.*
- (2) Validated by users other than authors.*

Usability: Program comprehension tools and methodologies are used because the alternative, i.e. code reading, is tedious and time consuming. Therefore, any proposed tool or solution should be easy to set up and use. We believe the more automated the solution, the easier it is to use.

Criterion 6: Is the solution automated?

Possible scores:

- (0) Not automated.*
- (1) Partially automated.*
- (2) Fully automated.*

Intrusive data collection / observer effect: The observer effect is the impact of observing a process while it is running. If the process of data collection changes the behaviour of the software, then one cannot take the observed behaviour to be the true behaviour of the software while not being observed.

Criterion 7: How intrusive is the data collection?

Possible scores:

- (0) The data collection is very intrusive.*
- (1) It has some observer effect.*
- (2) It does not have any substantial observer effect.*

Target programming language: This criterion rates the extensibility of the proposed tool or solution.

Criterion 8: Is the solution extensible to other programming languages and programming paradigms?

Possible scores:

- (0) Only valid for the system demonstrated in the study.*
- (1) It can be used on other systems written in the specific programming language used in the study.*
- (2) It can be used on different programming languages and programming paradigms.*

3.2. Program Comprehension and Feature Analysis

The papers in this literature review were selected by doing several searches in Google Scholar using various combinations of Program Comprehension, Dynamic Analysis, Stat-

ic Analysis, Software Engineering, Fan-in and Fan-out. This produced several papers which were then filtered to pick out those that had done both static and dynamic data analysis for the purpose of program understanding. The following papers are then evaluated one by one using the criteria in Section 3.1.

3.2.1 Dynamic Fan-in and Fan-out Metrics for Program Comprehension

Yuying *et al.* [118] propose an approach to understand a subject system by using fan-in and fan-out metrics. They observe that methods that invoke other methods frequently or get invoked by other methods frequently play an important role in the system performance. Based on their observation, they hypothesize that methods with high fan-in or fan-out values implement the system's main functions and can be used to infer the system's functionality. They call *critical components* the components that implement the main functions or main structure of a system.

They set out to show that methods and classes with high fan-in or fan-out values play an important role in the system performance. They identified components they believed to be critical to the subject system, by collecting dynamic fan-in and dynamic fan-out values at the method and class levels. To collect the data, they first duplicated the source code and instrumented the copy to be used for data collection. They run the instrumented version, and collected execution traces of method calls and return values. From the trace, they calculated dynamic fan-in and dynamic fan-out for each of the methods in the trace, and checked to see if the critical systems are the ones with high fan-in and fan-out values. They used a highway simulation software system that simulates buses on a highway linking two cities. The results they obtained suggest that dynamic fan-in and fan-out are good indicators to identify critical methods and critical classes, and that they can be used to infer the system functionality.

The experiment Yuying *et al.* [118] developed has a few problems. We use the evaluation criteria set in Section 3.1 to evaluate the experiment.

- **Scalability: (0)** The authors had to change the source code manually to instrument it. Having to manually edit the code makes the method very difficult to use for working on large systems.

- **Scalability for trace size: (0)** The authors did not discuss or demonstrate how large traces or large programs can be handled.
- **Visualization: (0)** The results were not visualized.
- **Level of prior knowledge required: (0)** Some prior knowledge of the source code is required. The user would have to know the code or would have to learn the code in order to instrument it properly.
- **Validation: (1)** The authors validated their results themselves, by inspecting the source code. They also validated the experiment on only two systems, one of them a “simple Client/Server system”.
- **Usability: (0)** The solution is not easy to use because the user would have to duplicate and change the source code.
- **Intrusive data collection: (0)** The code had to be changed to collect data.
- **Target programming language: (1)** The method can be used on other systems written in C++. Even though the authors did not discuss or demonstrate if the solution can be extended to other object-oriented languages, it appears that it can be.

3.2.2 Scenario-Driven Dynamic Analysis for Comprehending Large Software

Salah *et al.* [99] present an approach that combines static and dynamic analysis to extract views of a software system at different levels, in order to understand large software systems. The extracted views are at use-case, module interaction, and class interaction levels. Their approach maps use-cases to portions of the source code that implement them by using dynamic analysis. This is done by exercising the feature (defined as a use-case scenario), collecting execution trace, and analyzing the trace.

Dynamic data comes in the form of an execution trace. The authors employ the concept of marked execution traces to define program features. A marked-trace is established manually during the execution of the program, by specifying the start and the end of the trace using a trace-marker feature of the profiler.

Static data comes from the “change requests” that are usually written in natural language with explicit references to software features. A developer can start from the change request, then execute the application in a profiling mode, and collect dynamic in-

formation only from the specific portion of the system. The authors use static data to guide where and how much dynamic data, (i.e., execution traces), should be collected.

After the feature is exercised, the tools analyze the traces to produce a function call graph of the program. They also generate function call statistics such as the total execution time, the number of times a method is invoked, and the number of exceptions thrown by each method. The repository subsystem defines the data and metadata models, as well as the data manipulation and query language. Finally, the analysis and visualization subsystem is used to create a visualization of the software views at various levels of detail.

The views are created, by subjecting the software systems to dynamic analysis, under various use-case scenarios. Two sets of views are constructed from the runtime data: (1) graphs that capture the parts of the software's architecture that pertain to the use-cases, and (2) metrics that measure the intricacy of the software and the similarity between use-cases. The scoring is as follows:

- **Scalability: (1)** Even though the authors point out that the performance of the approach degrades when profiling large systems, we gave the approach a 1 because by marking the beginning and end of a feature, the user is able to collect as much or as little information as needed, no matter what the size of the system is. .
- **Scalability for trace size: (1)** Even though the user has control on the size of the trace file, the authors did not discuss how trace files can be reduced.
- **Visualization: (2)** Full mark for visualization because the models can be visualized at various levels.
- **Level of prior knowledge required: (2)** No prior knowledge is required.
- **Validation: (2)** Validated by users other than authors.
- **Usability: (1)** Partially automated.
- **Intrusive data collection: (1)** The collection of traces has some observer effect.
- **Target programming language: (1)** The approach can be used on other systems written in C++.

3.2.3 Integrating Static and Dynamic Analysis to Improve Comprehension

Lucca and Penta [26] present a technique based on static and dynamic analysis of Web applications, to identify the client pages that are dynamically generated at runtime by server pages, and to group together sets of similar built client pages to be considered as a single equivalent page.

They used two existing Web application reverse engineering tools, WARE [27] (Reverse Engineering Web Application) for performing static analysis, and WANDA [5] (Web ApplicationNs Dynamic Analyzer) for dynamic analysis. Both tools have been successfully adopted in the past to perform several reverse engineering tasks over Web applications (WAs). They show how the integrated toolkit can be used to identify equivalence classes from groups of dynamically built client pages. The integration has been reached, by making compatible and consistent, the data stored in the databases produced by the two tools.

This approach requires that the WA is first statically analyzed to identify the server pages generating Built Client Pages (BCPs), and the Linear Independent Paths (LIPs) in such pages. Then, the code is instrumented to collect the execution traces corresponding to the WA navigations performed by its users, and the paths actually executed in the server pages. Finally, the execution traces are analyzed to identify the actual BCPs. Equivalent BCPs are identified by grouping together the ones due the execution of the same LIPs in the server pages.

A case study has been carried out to validate the proposed approach. The results showed that dynamic information collected via WA instrumentation allowed the set of BCPs to be correctly identified, as well as the groups of equivalent BCPs. The case study also showed that the greater the number of collected execution traces, the better the results the approach produced.

- **Scalability: (1)** Scalable.
- **Scalability for trace size: (0)** The authors did not discuss how a trace file can be reduced.
- **Visualization: (2)** The resulting model is visualized.
- **Level of prior knowledge required: (2)** No prior knowledge of the source code is required.

- **Validation: (1)** The method is validated by the authors.
- **Usability: (1)** Partially automated.
- **Intrusive data collection: (1)** The method has an observer effect from the trace collection.
- **Target programming language: (0)** The method used is specific to analyzing Web pages.

3.2.4 Understanding the Behaviour of Java Programs

Systä [105][106] demonstrates how the Shimba environment aids in understanding the behaviour of Java program, by reverse engineering the software. The dynamic trace information is generated automatically, as a result of running the target system under a customized SDK [107] debugger, and viewed as scenario diagrams using the SCED tool [65]. A scenario diagram is a variation of a UML sequence diagram. The SCED scenario diagram notation is an extension of a basic Message Sequence Chart (MSC), containing concepts semantically similar to those included in UML 2.x sequence diagrams.

The dynamic event trace information is generated automatically using the SDK. Breakpoints are set at the first line of all methods, constructors, and static initialization blocks of the selected classes. The generated event trace information is viewed as scenario diagrams using SCED. In SCED, state diagrams can be synthesized automatically from scenario diagrams. This facility is used to visualize the total behaviour of a selected object or method, disconnected from the rest of the system. The system under investigation is the FUJABA environment, consisting of almost 700 Java classes. Information about static software artifacts and their relations is viewed using the Rigi reverse engineering tool [86].

- **Scalability: (1)** The string matching algorithms are able to find behavioural patterns only if they lay entirely inside one SCED scenario diagram. Patterns that begin from the end of one scenario and continue in the next scenario are not recognized. This makes modeling the overall behaviour of a complicated object that have large state-chart diagram and that does not fit in one SCED, impossible.

- **Scalability for trace size: (1)** The authors discuss how pre-filtering can decrease the size of generated trace; however, they did not discuss how a trace file can be reduced after it is generated.
- **Visualization: (2)** Information about static software artifacts is visualized using Rigi. The dynamic trace information is viewed as scenario diagrams using the SCED tool.
- **Level of prior knowledge required: (1)** No prior knowledge is required.
- **Validation: (2)** Validated by users other than authors.
- **Usability: (1)** The user has to know how to run the SDK debugger, however, we feel that most programmers would know how to use a debugger, and the skills are for the most part transferable to any other debugger.
- **Intrusive data collection: (0)** Running code under debugger does alter the behaviour of a model which depends on timing.
- **Target programming language: (2)** The methodology described can be used on other languages and object-oriented systems. The authors also discuss that, with minor changes, it can be used for object interoperability like CORBA, RMI, DCOM, etc.

3.2.5 The CONCEPT Project

Rilling *et al.* [95] combine several visualization techniques with source code analysis to reduce the amount and complexity of the data that has to be displayed to provide clear representation of a system and its logical structure. The main motivation of this project is to address a variety of comprehension issues for large and/or network centred software systems.

The authors focus on static and dynamic program slicing, and they apply source code analysis technique on treemaps, Hyperbolic trees, and UML-based visualization techniques to support programmers in creating better mental models of the source code. They introduce a prototype called CONCEPT (Comprehension Of Net-Centered Programs and Techniques), and describe how the presented approaches can be applied to reduce the information complexity for particular source code comprehension applications. They reduce the complexity of the data to be visualized by using slicing related concepts

(contributing and influencing program artefacts). These concepts, combined with their various levels of visual abstraction, provide additional source code insights. The technique can be applied to various source code based comprehension tasks (e.g., debugging, testing, performance analysis, etc.).

- **Scalability: (1)** The authors had to change the source code manually to instrument it, and this would make the approach challenging for working on large systems.
- **Scalability for trace size: (0)** The authors did not discuss how a trace file can be reduced.
- **Visualization: (2)** Various visualization techniques were presented by the study, including UML-based diagrams, treemaps, and hyperbolic trees.
- **Level of prior knowledge required: (0)** Some prior knowledge of the source code is required. The user would have to know the code or would have to learn the code in order to instrument it properly.
- **Validation: (1)** The authors validated their results themselves, by inspecting the source code. They also validated the experiment on only two systems, one of them a “simple client/server system”.
- **Usability: (1)** The solution is not easy to use. They have to duplicate and change the source code.
- **Intrusive data collection: (1)** The method used has some observer effect.
- **Target programming language: (2)** The method can be used on other systems written in C++. Even though, the authors did not discuss or demonstrate if the solution can be extended to other object-oriented programs, it appears that it can be.

3.2.6 Mapping Program Features to Code

Wilde and Scully [115] developed a test-case based method, called *software reconnaissance*, to map features onto software components to discover where particular program features are implemented. The method uses both static and dynamic analysis.

They studied a target program containing about 15,000 lines of C code in 360 functions. They gathered the static information manually by reading the specification

document. The static information that was collected was the target program's list of functions. They then selected test cases and classified them.

For the dynamic portion of the study, they instrumented the program so that the components executed in any test case would be identified. Then two sets of test cases were developed: one set containing test cases that exhibited the feature being sought, and one set with test cases that did not exhibit this feature. Executions on the two sets were compared to identify code components that are preferentially exercised when the feature is present. The researchers demonstrated, by using only a test coverage analyzer, some simple tools, and a few test cases, that code components responsible for specific features can be located.

This information alone is not sufficient to identify the interface and the constituents of a component in the source code, but these subprograms are a starting point for a more detailed static analysis. Their approach deals with one feature at a time and gives no insight into connections between a set of related features. Another limitation of their method is that it does not find code for functionality that is always present in the program. For example, while the maintainer of a compiler might want to find the components that implement the symbol table, the reconnaissance method could not help since all non-trivial test cases would create and access the symbol table. In this thesis, we call those components *utilities*.

- **Scalability: (1)** The technique is scalable for software size because, no matter the size of the target software, the user can focus on one feature at a time. However, one could still end up with an unmanageable list of components that need to be looked at. For big features, a further simplification of the list would be required.
- **Scalability for trace size: (0)** The authors did not discuss how a trace file can be reduced.
- **Visualization: (0)** No visualization.
- **Level of prior knowledge required: (2)** No prior knowledge of the code is required to use this method to locate feature components. The user only needs to be able to select suitable test data and run the tests.
- **Validation: (2)** The authors used two subjects to validate their method. They used a real production program (XREF.C, a common shareware utility).

- **Usability: (1)** The technique is very user friendly because all the users have to do is exercise the feature of interest and collect data. We give it only 1 because it does not give information on feature interactions.
- **Intrusive data collection: (1)** The only observer effect it has is from execution trace collection.
- **Target programming language: (2)** The technique of comparing execution traces is a valid technique to use for other programming languages or paradigms.

3.2.7 Aiding Program Comprehension by Static and Dynamic Feature Analysis

Eisenbarth, Koschke and Simon [35] present a technique to map the system's externally visible behaviour to the relevant parts of the source code. The technique combines dynamic and static analysis to derive correspondences between features and code components, with a goal of reverse engineering a partial description of the software architecture. They use concept analysis – a mathematical technique to investigate binary relations – to derive correspondences between features and components.

The researchers use execution summaries generated by a profiler for different scenarios to identify the subprograms executed when any of the given features is invoked. A subprogram is defined as a function or procedure according to the programming language. One scenario represents the invocation of one feature only and yields all subprograms executed for this feature. They then use concept analysis to derive detailed relationships between features and executed subprograms. Concept analysis is used here to locate the most feature-specific subprograms among all executed subprograms.

The information gained by concept analysis (scenario-subprogram relationship) is then used to guide a subsequent static analysis along the static dependency graph, in order to narrow the executed subprograms to those that form self-contained and feature-specific components. Subprograms that are only utility subprograms used as building blocks for a component but not containing any application-specific logic are sorted out. Additional static analyses, like strongly connected component identification, dominance analysis, and program slicing, support the search for the components of interest. The combination of dynamic and static information reduces the search space drastically.

- **Scalability: (2)** Full marks for scalability because no matter the size of the target software, the user can focus on one feature at a time.
- **Scalability for trace size: (0)** The authors did not discuss how a trace file can be reduced.
- **Visualization: (1)** Although the concept lattice information can be visualized, the authors did not discuss whether the final mapping of all the features to the component can be visualised.
- **Level of prior knowledge required: (2)** No prior knowledge of the code is required to use this method to locate feature components. The user only needs to know how to invoke the features and select suitable test data.
- **Validation: (1)** The authors used two Web browsers using the same set of relevant related features. However, they did not specify how they validated the components that were identified were indeed the same ones that were used by the features.
- **Usability: (2)** The technique is very user-friendly because all that the user needs to do is exercise the feature of interest and collect data, and the process is mostly automated. We give it a high score because it can be used for multiple features and it also gives information on feature interactions.
- **Intrusive data collection: (1)** The technique has an observer effect from execution trace collection.
- **Target programming language: (2)** The technique of comparing execution traces is a valid technique to use for other programming languages or paradigms.

3.2.8 Recovering High-Level Views from Static and Dynamic Analysis

Richner and Ducasse [94] present an environment supporting the generation of tailorable views of object-oriented systems from both static and dynamic information. Their approach is based on the combination of user-defined queries that allow the user to create high-level abstractions and to produce views using these abstractions.

Their reverse engineer object-oriented applications is based on the use of a logic programming language for recovering views of the code. Static information about the program is represented by a meta-model that reifies object-oriented notions such as class,

super-class, inheritance, and attribute. This structural information is complemented with dynamic information. Behavioural information, in the form of traces from program execution, provides a program slice. The views generated work as catalysts for generating questions about the studied system, and helping to steer the process of design recovery.

- **Scalability: (0)** The tool as presented does not support very large traces, hence it is not scalable.
- **Scalability for trace size: (1)** The tool as presented does not support very large traces; however, the authors give a method that can be used to reduce the amount of trace information generated. They discuss how tracing can be more sparingly used by instrumenting the application at only some of the methods or classes.
- **Visualization: (2)** The environment includes tools for visualization, extraction and analysis of views. In addition, views can be refined to different levels of abstraction, and they are tailorable.
- **Level of prior knowledge required: (1)** No prior knowledge of the code is required to use this method to locate feature components. The user only needs to know how to use Method Wrappers [16], which allow instrumentation at the level of individual methods. However, as suggested by the authors, to be able to reduce the length of traces collected, the user would have to know the system to instrument the application at selected methods or classes.
- **Validation: (1)** The authors used the HotDraw Framework [13] to validate their method. However, they did not specify how they validated the components that were identified were the same components that were used by the features.
- **Usability: (1)** The user needs to know how to use Method Wrappers before being able to extract views using the method defined in this paper.
- **Intrusive data collection: (1)** The technique has an observer effect because the application is instrumented for data collection.
- **Target programming language: (2)** The approach is adaptable to other class-based object-oriented languages.

3.2.9 Analysis of Execution Traces for Program Comprehension

Hamou-Lhadj, in his Ph.D. dissertation [46], studied a technique to simplify the analysis of traces to facilitate understanding of large execution traces. He presented a set of metrics for measuring various properties of an execution trace, in order to assess the work required for understanding its content. He proposed that if various aspects that contribute to a trace's complexity could be measured, this information could be used by tools to analyze traces.

The metrics he collected are:

1. **Call Volume**, which is the number of calls in a trace.
2. **Component Volume**, which is the number of distinct components that are invoked during the execution of a particular scenario.
3. **Comprehension Unit Volume**, which is a measure that shows how many comprehension units exist in a trace. It is computed from the graph representation of the trace.
4. **Patterns Related**, which are concerned with measuring the number of patterns that exist in a trace, where a pattern is a comprehension unit that is repeated in a trace non-contiguously.

Metrics provide limited indication of the complexity of a trace. In subsequent work, Hamou-Lhadj and Lethbridge [51] address this issue by presenting a catalogue of metrics for assessing the various facets of traces of routine calls, with the ultimate objective being to facilitate the development of tools for the exploration of lengthy traces.

Hamou-Lhadj presented a trace summarization algorithm, which consists of taking a trace as input, successively filtering the implementation details of the trace and returning a summary of its main content as output. The author also presents an analysis of the concept of implementation details such as utilities. It is this aspect of his work, i.e., the application of trace summarization, by removing implementation details from traces that we are concerned with in this thesis. The author used *static* fan-in and fan-out to find utilities, whereas in this thesis we use *dynamic* fan-in and fan-out. We will take a closer look at this work in Chapter 5 when we compare the algorithm developed in [46] with the algorithms that are developed in this thesis.

The authors also developed a scalable exchange format called the Compact Trace Format (CTF) in order to enable sharing and reusing of traces. Hamou-Lhadj and Lethbridge [48][49] present a detailed discussion of CTF.

- **Scalability: (2)** The author has developed a scalable exchange format (CTF) in order to enable sharing and reusing of traces. The author mainly discusses scalability related to the Compact Trace Format exchange. However, we feel that the trace filtering method makes it easy to handle larger software systems.
- **Scalability for trace size: (2)** Successive filtering can reduce the size of a large trace making the method scalable.
- **Visualization: (1)** The resulting model is visualized using UML sequence diagrams.
- **Level of prior knowledge required: (1)** Even though no prior knowledge of the source code is required, the user does have to build a custom tool to use the method described in this thesis.
- **Validation: (2)** The method is validated with software developers who have knowledge of the system.
- **Usability: (0)** One needs to build a custom instrumentation tool, which would make the use of this methodology for other systems difficult.
- **Intrusive data collection: (1)** The method has an observer effect from the trace collection.
- **Target programming language: (2)** The approach is adaptable to other class-based object-oriented languages.

3.2.10 Automatic Generation of Abstract Views for Software Comprehension

Dugerdil and Repond [31] present a technique and a tool to generate abstract sequence diagrams to represent the global working of legacy programs. They use a software clustering technique based on dynamic analysis of execution trace to build an abstract view of the system. They generate the main interactions between clusters of events, which they call functional components, i.e., a set of program elements that closely work together to implement a subtask of a business function.

Dugerdil and Repond use static and dynamic data to compress the execution traces. They use dynamic analysis to get the conceptual grouping of classes as well as the analysis technique. They use static information, for example software packages, to do the comparison of the dynamic clusters with the static structure of the program. The combination of both the static and dynamic data is used to show where the responsibilities pertaining to some behaviour are actually implemented.

The main steps of their approach are as follows:

1. An execution trace is generated when executing some scenario of business value.
2. Clusters that are linked to business functions are identified. The clusters represent chunks of elementary functionality; if the elements are tightly coupled, they form what the authors call a functional component.
3. The working of the system is abstracted by displaying only the method called between the clusters.
4. The resulting event set is further compressed using event patterns repetition compression.

This work has much similarity to our work, in that the final result is a visualized behavioural model of the system. The authors include all classes in the system; unlike our work, they do not make the assumption that a class might be an implementation-only (utility) class without functional code. This means they have less chance of losing sight of functional code that is housed in a utility function. On the other hand, they collect events between application classes, as opposed to methods, so they do not start with a detailed execution trace that includes all events like we do.

- **Scalability: (2)** The technique and the tool can handle a large system.
- **Scalability for trace size: (2)** Their technique has a compression factor of 8000 to 1, and therefore is able to deal with execution traces.
- **Visualization: (2)** The authors used UML 2.x sequence diagrams.
- **Level of prior knowledge required: (2)** No prior knowledge of the code is required to use this technique.
- **Validation: (0)** The authors did not validate the resulting sequence diagrams with users. They validated the use-cases, (e.g., “we presented the use-cases to the users for validation and correction”), but not the sequence diagrams.

- **Usability: (2)** The technique is very user-friendly because the sequence diagram is automatically generated.
- **Intrusive data collection: (1)** The only observer effect is from execution trace collection.
- **Target programming language: (1)** Sequence diagrams are useful for showing the time-ordering of messages passed between object instances, but they do not work all that well for non object-oriented languages.

3.2.11 Solving the Feature Location Problem by Measuring the Component Modification Impact

Rohatgi, Hamou-Lhadj and Rilling [96] present a feature location approach that combines static and dynamic analysis techniques. They use a component dependency graph of the system and a feature trace of a selected feature to rank classes according to their relevance to the feature.

They instrument the system, exercise the feature and collect execution traces. As an execution trace is being created, they extract the distinct classes that are invoked. They call the collected classes the *execution profile* of the feature. A dependency graph is then created to rank the distinct classes according to their relevance to the feature. The ranking is based on the impact of a *component modification* (where a component maps to a class) on the remaining parts of a system.

They hypothesise that the smaller the impact of a component modification, the more likely it is that this component is specific to the particular feature. The rationale behind this is that classes that impact many other parts of the system will most likely be invoked in many other feature traces, making them non-feature-specific. A feature-specific class would tend to be self-contained (i.e., low coupling and high cohesion), and a modification to such a class should result in a very low impact on the remaining parts of the system.

The feature-specific classes are similar to what we call main classes of a feature and the non-feature-specific classes are similar to what we call *utility* classes. We use the evaluation criteria set in Section 3.1 to evaluate the experiment.

- **Scalability: (2)** Full marks for scalability because no matter the size of the target software, the user can focus on one feature at a time.
- **Scalability for trace size: (2)** The execution trace is processed “on the fly” and does not need to be saved, therefore the approach can handle large execution traces.
- **Visualization: (0)** The results were not visualized.
- **Level of prior knowledge required: (2)** No prior knowledge of the code is required to use this technique.
- **Validation: (1)** The authors used two software systems, Weka and Checkstyle, to validate their approach. However, they did not use subjects other than the authors for validation.
- **Usability: (2)** The approach is very simple and almost completely automatic, requiring almost no human intervention.
- **Intrusive data collection: (1)** The code is instrumented by inserting probes in the source code, therefore their approach has an observer effect.
- **Target programming language: (2)** The approach is adaptable to other object-oriented languages.

3.2.12 Summary of Program Comprehension and Feature Analysis Studies

Static and dynamic methods have complementary strengths and weaknesses. The research papers reviewed in this section take advantage of the strengths and minimize the weaknesses, by using the output of one as input to the other.

Using static analysis first followed by dynamic analysis can help reduce instrumentation requirements or guide test cases towards suspicious code, as seen on static diagrams. Ordering dynamic methods first followed by static ones can help to verify observed behaviour by looking at the static data (how the code is structured). Researchers can use dynamic analysis to do a feature-oriented search of the source code focusing on the components of interest. Elberzhager et al. [38] analyse and classify several existing approaches that combine different static and dynamic quality assurance techniques, including reported effects, characteristics, and constraints.

While evaluating the papers, we did not analyze how well the studies took advantage of static and dynamic analysis methods. Instead, we focused on how well the studies solved the main topic of this thesis, i.e., program understanding. We ranked the papers on the various criteria. The scores for individual criteria were 0 for not addressed, 1 for partially addressed and 2 for fully addressed. For ease of reading, we color-coded the results in the table to be red (0), yellow (1) and green (2). Table 1 shows the summary score for the papers reviewed in this section.

Criteria used	Scalability	Scalability for Trace Size	Visualization	Level of Prior Knowledge Required	Validation	Usability	Intrusive data collection	Target Programming Language
Dynamic Fan-in and Fan-out Metrics for Program Comprehension (Section 3.2.1)	0	0	0	0	1	0	0	1
Scenario-Driven Dynamic Analysis for Comprehending Large Software (Section 3.2.2)	1	0	2	2	2	1	1	1
Integrating Static and Dynamic Analysis to Improve Comprehension (Section 3.2.3)	1	0	2	2	1	1	1	0
Understanding the Behaviour of Java Programs (Section 3.2.4)	1	1	2	1	2	1	0	2
The CONCEPT Project (Section 3.2.5)	1	0	2	0	1	1	1	2
Mapping Program Features to Code (Section 3.2.6)	1	0	0	2	2	1	1	2
Aiding Program Comprehension by Static and Dynamic Feature Analysis (Section 3.2.7)	2	0	1	2	1	2	1	2
Recovering High-Level Views from Static and Dynamic Analysis (Section 3.2.8)	0	1	2	1	1	1	1	2
Analysis of Execution Traces for Program Comprehension (Section 3.2.9)	1	2	1	1	2	0	1	2
Automatic Generation of Abstract Views for Software Comprehension (Section 3.2.10)	2	2	2	2	0	2	1	1
Solving the Feature Location Problem by Measuring the Component Modification Impact (Section 3.2.11)	2	2	0	2	1	2	1	2

Table 1 Summary scores for reviewed papers.

All of the papers except Hamou-Lhadj's (Section 3.2.9) used static and dynamic data to tailor and reduce the amount of data collected but, did not go the extra step of reducing (summarizing) the dynamic data *after* it has been collected. Trace summarization, trace reduction, and abstraction are the methods that are traditionally used to handle large amounts of dynamic analysis output. We believe trace reduction/summarization to be an

important step, because one of the big problems, with using dynamic analysis, is the amount of data one has to analyse to get to a level that is understandable by humans.

With this work, we try to address this important step. We will develop several algorithms to reduce execution traces. We will also improve on the algorithm developed by Hamou-Lhadj [46].

We take advantage of the strength of both static and dynamic analysis for software comprehension, but before we use the dynamic data, like the majority of the papers reviewed in this section, we will use information from the static data to reduce it to manageable size. We will use the same criteria to assess the methodology and tool presented in this thesis in later chapters.

3.3. Chapter Summary

In this chapter, we reviewed existing research in software engineering that uses a combination of static and dynamic analysis techniques. We compared the papers related to software comprehension using a set of criteria we defined in Section 3.1. We observed that most of the papers did not address dynamic data reduction as part of the papers. We are tackling this gap in this thesis.

In the next chapter we look at the experiment we designed to reverse engineer behavioural model from execution traces.

Chapter 4. Data and Methodology Used in the Empirical Studies

In this chapter, we discuss the static and dynamic data used for our empirical studies that aim to detect utility methods among all the methods used in one execution of a feature. We describe the approach taken to collect and analyze static and dynamic data from the software systems.

We also discuss why we decided to use the particular parameters we chose to use for the studies. In the last section of the chapter, we will look at each stage of the study and discuss the methodology.

4.1. Design of the Study

4.1.1 Systems Studied

Before studying large production systems, we needed to have small programs to test our algorithms. Instead of writing the programs ourselves, we used Standard Widget Toolkit (SWT) example code [104]. SWT is an open source widget toolkit for Java designed to provide efficient and portable access to the user-interface facilities of the operating systems on which it is implemented. There are several SWT examples that range in size from 5 classes to 38 classes [104].

In all, we study 18 software systems that are all written in Java. Of these 18 programs, we extract behavioural models for four of the systems. These four programs were selected because we had access to their developers, with whom we need to validate our findings. The other fourteen programs are small programs used for proof of concept, mainly to check that our theories hold for smaller programs. The results for those are validated by stepping through the code. For example, once a utility detection algorithm is run, we simply inspect the source code to validate that the methods identified as utilities are *not* the main methods of the system, and we verify that it is possible to describe the

behaviour of the system without mentioning those methods. In other words, once a *utility* detection algorithm is run, we look at the source code to verify that the methods identified as *utilities* are in fact utility methods and central methods of the system. The complete list of software systems we looked at is presented in Table 2. The four systems studied in detail (which are highlighted in the table) are briefly described in this section.

System	Number of Classes	Number of Methods	Lines of Code
Browser example	1	9	319
hover help	2	7	248
Texteditor	2	18	359
DND	2	23	1316
Image analyzer	2	49	1906
Java viewer	3	24	442
Clipboard	3	14	476
Address book	4	52	915
Helloworld	5	5	131
Layout example	7	79	2075
Paint	29	158	1584
TraceToUCM	35	96	3441
Control example	38	481	6495
Graphics	39	290	4840
Umple	430	4457	79459
JHotDraw	674	5933	80530
UCEd	700	6033	33004
jUCMNav	851	7406	93919

Table 2 List of software systems studied.

Umple Model-Oriented Programming Technology

Umple [11][74][41] is a modeling tool and programming language family that enables Model-Oriented Programming. It was developed at the University of Ottawa. It is a set of extensions to existing object-oriented languages (currently PHP and Java) that provides a concrete syntax for UML abstractions, like associations as well as certain software patterns. Umple adds abstractions, such as associations, attributes, and state machines derived from UML to object-oriented programming languages such as Java, PHP and Ruby. It can also be used to create UML class and state diagrams textually. The Umple compil-

er [110] itself is written in Umple. Umple 1.15.0.1751 has 44 packages, 430 classes, 4457 methods and 79459 lines of code.

jUCMNav

jUCMNav is a Java-based, open source Eclipse plug-in, mainly developed at the University of Ottawa [87]. It is a graphical modeling and analysis tool for the User Requirements Notation (URN), which includes the Use Case Map (UCM) notation and the Goal-oriented Requirement Language (GRL) [62]. jUCMNav is the largest system studied. Version 4.3.0 has 851 classes, 7406 methods, and 93919 LOC.

UCEd (Use Case Editor)

UCEd [111] is an environment for use-case-based requirements engineering. It also has tools for editing use cases, such as use-case integration and use-case simulation. It is also developed at the University of Ottawa [102]. UCEd 1.6.2 has 700 classes, 3284 methods, and 33004 LOC.

TraceToUCM

TraceToUCM is an application developed in this thesis to create UCM diagrams from an execution trace file, and mainly described in Chapter 6. TraceToUCM contains 1 package, 35 classes, 96 methods and 3441 lines of code.

4.1.2 Feature Selection

One feature per program is selected and run to collect an execution trace. For this thesis, we have selected a feature of each system that is most common. For the behavioural modeling to be effective, the target program must be executed with representative inputs to produce interesting execution traces. Care is taken to use inputs that are not too simplistic. For example, one of the features selected for jUCMNav is *opening a Use Case Map*. The UCM model that is loaded is one that is not too small and has a few UCM components, responsibilities, and alternatives.

4.1.3 Tools Used to Collect Data from the Systems

We use two tools to collect the dynamic and static data from the software systems. We were not able to find one tool that can collect all the data we needed; therefore we had to use multiple tools and write code to combine the results.

TPTP

The Eclipse Test & Performance Tools Platform (TPTP) Java profiler is a free tool for analyzing the runtime behaviour of Java applications. It provides a framework and services for an open platform upon which one can build test and performance tools [34]. TPTP contains a Probekit that allows one to write code that can be invoked at the entry and exit of a method. The code can be used to output the desired information to trace the method invocations. For our purpose, we instrumented the Probekit to output full method name and call hierarchy at method entry, and method name only at method exit.

We chose TPTP because it can easily be integrated with Eclipse and it can be used to collect CPU usage and execution traces.

METRICS 1.3.6

Metrics 1.3.6 [82] is an open source plug-in for the Eclipse development environment. It is used to gather static data on software systems. It collects various measures of software at the package level, class level, and method level. A full description of the collected data is available in [53] and is further discussed in Section 4.3.

4.1.4 Validation Participants

For UCed, Umple and jUCMNav, once utilities are filtered out of a trace by the various algorithms, the resulting traces are validated by system designers who can confirm whether the essence of the feature has been captured and is sufficient for high-level comprehension. Participants for the study were recruited via e-mail. The participants targeted are designers and/or maintainers of the systems being studied. The recruitment e-mail and questionnaire were pre-approved by the Research Ethics Board of University of Ottawa. The recruitment letter and questionnaire can be found in Appendix A.

4.2. Dynamic Data Collected

In this section, we explore the data collected through dynamic analysis, which is performed by executing the software. Dynamic analysis derives properties that hold for one or more executions by examination of the running program through program instrumentation. This can provide useful information about the behaviour of programs for the specific input parameters that are entered. For each dynamic variable we chose to use, we will give a more detailed definition, describe how the data was collected, and explain why we chose to gather the specific data.

4.2.1 Gathering Execution Traces

Execution traces are collected using the Eclipse plugin TPTP. Probes are inserted into the source code to print out a line containing the method name. The lines are printed both at method entry and exit. We also keep a running total of how deep in the call tree a particular call happened.

The following is an example of execution trace from the *draganddrop* (DND) system. We can see the first level is the *main* method, at level 0. It then initializes the class *DNDEExample*, and calls the method *open* still in the *DNDEExample* class, which in turn invokes other methods (down to 3 levels of nested calls).

```
Entered: org.eclipse/swt/examples/dnd/DNDEExample.main 0
Entered: org.eclipse/swt/examples/dnd/DNDEExample.<init> 1
Exited: org.eclipse/swt/examples/dnd/DNDEExample.<init>
Entered: org.eclipse/swt/examples/dnd/DNDEExample.open 1
Entered: org.eclipse/swt/examples/dnd/DNDEExample.createDragWidget 2
Entered: org.eclipse/swt/examples/dnd/DNDEExample.createWidget 3
Exited: org.eclipse/swt/examples/dnd/DNDEExample.createWidget
Entered: org.eclipse/swt/examples/dnd/DNDEExample$10.<init> 3
Exited: org.eclipse/swt/examples/dnd/DNDEExample$10.<init>
Entered: org.eclipse/swt/examples/dnd/DNDEExample$11.<init> 3
Exited: org.eclipse/swt/examples/dnd/DNDEExample$11.<init>
Exited: org.eclipse/swt/examples/dnd/DNDEExample.createDragWidget
Entered: org.eclipse/swt/examples/dnd/DNDEExample.createDragOperations 2
Entered: org.eclipse/swt/examples/dnd/DNDEExample$1.<init> 3
Exited: org.eclipse/swt/examples/dnd/DNDEExample$1.<init>
Entered: org.eclipse/swt/examples/dnd/DNDEExample$2.<init> 3
Exited: org.eclipse/swt/examples/dnd/DNDEExample$2.<init>
Entered: org.eclipse/swt/examples/dnd/DNDEExample$3.<init> 3
...
```

The execution trace is then analyzed to derive a set of dynamic data that describes how that method was used (i.e., how it behaved) in this particular run of the selected feature. Table 3 gives the complete list of metrics and other data we collect from the execution

trace. In the rest of this section, we explain in detail the specific dynamic data metrics that are used for this thesis.

Dynamic Data Collected for Each Method in the Execution Trace	Description of the Data Collected
Trace item name	Name of the method.
Fan-in	Number of methods that called this method.
Fan-out	Number of methods this method called.
CalledBy (an array)	Unique set of methods that called this method. The length of the array is the fan-in.
NumberOfTimesCalledBy (an array)	List of all methods that this method has been called by (has duplicates).
MethodsCalled (an array)	Unique set of methods that this method. The length of the array is the fan-out.
NumberOfTimesMethodsCalled	List of all methods that this method called (has duplicates).
Depth	Depth of this method in the call hierarchy.
TotalSegmentPresence	The total number of trace segments this method was found in.
PercentageOfSegmentsMethod-PresentIn	Percentage of trace segments this method was found in.

Table 3 List of dynamic data derived from an execution trace.

4.2.2 CPU Usage

CPU usage is collected using TPTP. TPTP can be used to collect data at a package level, class level or method level. For this study, we are interested in method-level CPU data because all the trace information is also collected at the method level. The three different kinds of data we collect are *base time*, *average base time* and *cumulative CPU time*. The base time is the time taken to execute the invocation of a method, excluding the time spent in other methods that were called during the invocation. Average base time is the base time divided by the number of calls. Cumulative CPU time is the amount of CPU time spent in a method accumulated from all invocations.

The CPU usage information is used as an anti-filter. As we iteratively filter out methods that have scored high on the utilityhood measure, we want to make sure we do not throw out methods that take proportionally large amounts of CPU time. Regardless of their score on the utilityhood measure, we hypothesize that methods that take proportionally large amounts of the CPU time need to be revisited. This may have been caused by

design problem or it may be that the method is important for the feature being studied and should be mentioned in a high-level behavioural model of the feature.

4.2.3 Fan-in/Fan-Out

Fan-in and fan-out metrics are structural metrics that measure inter-module complexities. They were first introduced by Henry and Kafura [55], who define the fan-in of a module as “the number of local flows that terminate at a module, plus the number of data structures from which information is retrieved”, and the fan-out as “the number of local flows that emanate from a module, plus the number of data structures that are updated by that module”. For the purpose of this study, we will disregard information about data structures both for fan-in and fan-out because it is out of the scope of this thesis.

In this thesis, *dynamic* fan-in analysis is performed on the execution trace generated by running a feature under study. We count fan-in and fan-out at the method level. We go through the entire trace and build a list of methods that are called by a particular method, and also a list of methods that call that same method. We then count the number of entries in the lists to create the fan-in and fan-out values for the method. See Appendix F for pseudo code.

The runtime behaviour in object-oriented programs comes from their interacting objects and methods. Detecting these interactions using static analysis is not easy. Static fan-in and fan-out metrics reflect structure dependency and they cannot capture all dimensions of object-level coupling because of features of object-oriented programming such as polymorphism, dynamic binding, and inheritance. Polymorphism makes it difficult to determine which method is actually executed at runtime, and because of inheritance, each method can exhibit the behaviour in its class or any of its super classes. This difficulty of determination is further aggravated in the case of dynamically typed languages like Smalltalk, where no type definition is available at compile time, and where methods are never statically bound [94]. Therefore, we chose to use dynamic fan-in and fan-out instead of static fan-in and fan-out because the objective of the study is to understand the behaviour of one feature, and we need to get precise runtime behaviour describing the application.

By focusing only on dynamic fan-in and fan-out, we automatically disregard any method that is not called in this particular execution of the feature. When we process the execution trace file, we count all overloaded methods as the same (i.e., multiple overloaded instances of a method are all considered to be one method).

Hamou-Lhadj [46] and our work [47], has shown that fan-in and fan-out parameters can be used to filter utilities and compress traces. In order to improve on those works, we investigate what other static and/or dynamic can be used in conjunction with fan-in. To that end for each of the 18 traces that were collected, all the static and dynamic data is put in columns of a spreadsheet and the correlation coefficient of fan-in with the other static and dynamic variables is calculated using the Excel =CORREL(list1, list2) function. Table 4 shows the results of the Umple correlation data which are then reflected, together with correlations for all the other applications we studied, in Table 5.

TraceItemName	UniqueFanIn	UniqueFanOut	averageCalledByDistance	AverageCallHierarchy	DataAccessMethod	Leaf	PercentageOfSegmentsMethodPresentIn	CpuUsageAverage	MLOC	McCabe	NBD	PAR
setEnds	1	4	2.00	12.00	0	0	0.05	0.00370	31	2	2	1
minimumNumberOfEnds	1	0	0.00	12.86	0	1	0.05	0.00001	1	1	1	0
setAssociation	1	0	0.00	13.93	0	1	0.05	0.00330	10	2	2	2
setLeftAndRight	1	1	2.00	12.00	0	0	0.05	0.00044	7	2	2	0
getName	2	1	1.00	11.49	0	0	0.10	0.00109	2	3	2	0
deriveName	1	3	0.00	12.73	0	0	0.10	0.00096	14	4	2	0
getEnd	5	0	1.20	10.35	1	1	0.10	0.00018	2	1	1	1
setTokenPosition	1	0	2.00	10.29	1	1	0.05	0.00001	4	1	1	1
Correlation of each column with fan-in	1.000	-0.062	-0.01	0.19	0.209	0.030	-0.01	0.01	-0.031	-0.091	-0.041	-0.184

Table 4 Correlation of Umple fan-in score with other static and dynamic variables.

The correlation of each of the static and dynamic variable with fan-in, for all 18 systems studied, is given in Table 5. We can observe that the Umple trace correlation values (highlighted in the rectangle) come from Table 4. The closer the value to 1, the greener the cell (and the closer the value to -1, the redder).

System name	Number of methods	Dynamic Data							Static Data			
		UniqueFanOut	EncounteredinTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	Cpu Usage Average	MLOC	McCabe	NBD	PAR
HelloWorld	2	-0.033	0.253	0.157	-0.071	0.139	0.229	-0.018	-0.019	0.093	0.052	-0.011
HoverHelp	6	-0.740	0.706	0.739	0.000	0.000	0.841	-0.015	0.012	-0.189	0.000	0.000
BrowserExample	7	-0.635	-0.574	0.914	-0.083	0.557	0.851	-0.582	-0.146	-0.109	-0.069	-0.064
Clipboard	9	-0.035	0.227	-0.125	-1.000	0.125	0.395	-0.673	0.466	0.316	0.534	0.395
DND	14	-0.279	-0.136	0.684	0.248	0.366	0.802	-0.489	0.338	0.750	0.559	0.304
TextEditor	14	-0.565	-0.727	0.101	0.113	0.207	0.139	-1.000	0.253	-0.027	0.320	-0.162
Address Book	16	-0.268	-0.077	0.067	0.098	0.200	-0.228	-0.298	0.179	-0.034	0.088	-0.200
JavaViewer	20	0.579	0.451	-0.026	-0.078	-0.186	0.149	-0.453	0.276	0.205	0.461	0.157
ImageAnalyser	26	-0.020	0.341	0.519	-0.047	0.175	0.457	-0.454	0.025	0.035	-0.018	0.311
LayoutExample	35	0.054	0.949	0.643	-0.110	-0.221	0.930	-0.284	-0.022	0.039	0.130	0.260
TraceToUCM_on_jhotdraw TPT	66	-1.000	-0.953	0.124	0.015	0.334	0.073	-0.982	-0.753	-0.480	-0.297	0.011
PaintExample	104	-0.005	0.368	0.351	0.023	0.103	0.604	0.036	-0.089	-0.066	-0.035	0.017
Graphics	213	-0.033	0.253	0.157	-0.071	0.139	0.229	-0.018	-0.019	0.093	0.052	-0.011
Control	285	0.068	0.807	0.298	-0.092	-0.022	0.847	-0.235	-0.012	0.017	0.051	0.124
jucmnav_TPTP_Nov26AfterExclu	305	-0.008	0.400	0.232	0.135	0.030	0.501	0.017	0.028	0.145	0.052	-0.056
Umple_compiling_BCMS TPTP	351	-0.062	-0.026	0.187	0.209	0.030	-0.012	0.010	-0.031	-0.091	-0.041	-0.184
Used TPTP open file_Nov12Af	458	-0.019	0.330	0.207	0.022	0.062	0.344	0.033	-0.075	-0.039	-0.008	-0.029
JHotDraw	566	0.012	0.309	0.291	0.118	-0.015	0.285	-0.030	-0.078	-0.055	-0.063	-0.012
		-0.037	0.296	0.247	0.057	0.042	0.359	-0.077	-0.051	-0.013	-0.001	-0.018

Table 5 Correlation of static and dynamic variables with fan-in, for all systems.

For example, the last line shows that JHotDraw has 566 methods in the trace. The fan-in value of the 566 methods correlated with fan-out is 0.012 it is 0.309 with encountered in trace, etc.

To generalize the correlation results, an average was calculated. However, since the traces have different sizes (ranging from 2 to 566 methods), each of the correlation coefficient was calculated by the number of methods. The column was then summed up and divided by the total number of methods in all the traces.

The weighted average coefficient shows that methods with high fan-in values also tend to be called more often. In Table 5, the column titled “EncounteredinTrace” shows the correlation between a method’s fan-in value and the number of times it is encountered in the trace. “EncounteredinTrace” is counts how many times the particular method is found in the execution trace. Also, the column titled “TotalSegmentPresence” in Table 5 shows that methods with high fan-in values also have high segmentation presence. Seg-

mentation presence is calculated using a formula developed by Dugerdil *et al.* [30] for trace segmentation and clustering (see Section 4.2.5). To calculate segmentation presence, a trace is divided into several equal segments, and we count how many of those segments a method is present in. The strongest of the correlations are 0.296 and 0.359 for “encountered in trace” (the number of times the method was found in the trace) and segmentation presence, respectively. We also observe that average call depth hierarchy tends to be high at a 0.247 correlation coefficient, meaning that methods with high fan-in are called deeper in the call tree hierarchy. Conversely, calls with high-fan in tend to have a lower fan-out, lower line of count, lower complexity, and lower CPU usage. None of the negative correlations were significant.

Table 6 shows the correlation of our static and dynamic variables with fan-out

System name	Number of methods	Static										
		UniqueFanIn	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	MLOC	McCabe	NBD	PAR
HelloWorld	2	-0.033	0.062	-0.165	-0.204	-0.217	-0.038	0.051	0.261	0.177	0.383	0.043
HoverHelp	6	-0.740	-0.112	-0.701	-0.497	-0.497	-0.545	0.617	-0.088	0.615	0.091	-0.497
BrowserExample	8	-0.635	0.980	-0.278	-0.278	-0.622	-0.272	0.974	0.081	0.034	-0.144	0.022
Clipboard	9	-0.035	0.863	0.122	0.035	-0.279	-0.387	0.757	0.195	0.387	-0.161	-0.486
DND	14	-0.279	0.929	-0.310	-0.194	-0.286	-0.466	0.712	0.198	0.006	-0.276	0.000
TextEditor	14	-0.565	0.298	-0.353	-0.394	-0.719	-0.226	0.583	-0.325	-0.410	-0.609	-0.050
Address Book	16	-0.268	0.830	-0.183	-0.268	-0.550	0.461	0.641	0.198	-0.012	0.175	0.099
JavaViewer	20	0.579	0.346	-0.374	-0.271	-0.645	-0.043	0.375	0.262	0.239	0.283	0.227
ImageAnalyser	26	-0.020	0.057	-0.317	-0.166	-0.531	-0.095	0.521	0.137	0.088	0.024	-0.046
LayoutExample	35	0.054	0.167	0.546	-0.371	-0.742	0.203	-0.401	0.209	-0.103	0.395	0.000
TraceToUCM_on_jhotdraw_TPT	66	-1.000	0.954	-0.133	-0.017	-0.359	-0.075	0.984	0.765	0.496	0.316	-0.007
PaintExample	104	-0.005	0.115	-0.074	-0.266	-0.377	0.062	0.053	0.306	0.180	0.166	0.035
Graphics	213	-0.033	0.062	-0.165	-0.204	-0.217	-0.038	0.051	0.261	0.177	0.383	0.043
Control	286	0.068	0.296	0.055	-0.216	-0.253	0.215	-0.135	0.165	0.077	0.220	0.100
jucmnav_TPTP_Nov26AfterExclu	305	-0.008	0.550	-0.045	-0.125	-0.263	-0.027	0.024	0.107	-0.006	0.004	-0.037
Umple_compiling_BCMS_TPTP	351	-0.062	0.286	-0.164	-0.218	-0.331	-0.072	-0.013	0.695	0.525	0.337	0.186
Uced_TPTP_open file_Nov12Af	458	-0.019	0.061	-0.080	-0.106	-0.583	0.008	-0.075	0.151	0.088	0.076	0.126
JHotDraw	566	0.012	0.041	-0.143	-0.260	-0.493	-0.082	-0.017	0.350	0.151	0.170	0.280
		-0.037	0.221	-0.096	-0.194	-0.404	-0.012	0.022	0.300	0.170	0.177	0.123

Table 6 Correlation of static and dynamic variables with fan-out, for all systems.

The weighted average coefficient shows that methods with high fan-out values also tend to be relatively larger (high MLOC values) at 0.3. Conversely, methods with high fan-out tend to not be leaf calls (-0.405). None of the other relationships are noteworthy.

4.2.4 Depth of Call

One of the variables we want to use in the utility detection algorithms is depth of call in the call hierarchy. If utilities have less fan-out than fan-in as stated by Hamou-Lhadj [46] they must also appear deeper in the call hierarchy. To test this theory and also to see if we can use this variable in utility detection algorithm, we calculate call hierarchies from an execution trace. We do this by looking at each call that has been made to this method, and taking note of its depth in the call tree. We increment the depth for each call and later divide it by the number of times the method has been called (its dynamic fan-in) to get the average depth of calls. We also make note of whether the method is a leaf on the call tree or not. See Appendix F for pseudo code of the algorithm.

		Dynamic									Static			
System name	Number of methods	UniqueFanIn	UniqueFanOut	EncounteredInTrace	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	0.157	-0.165	0.011	0.422	-0.124	0.081	-0.039	-0.039	0.005	-0.299	-0.126	-0.332	-0.045
HoverHelp	6	0.739	-0.701	0.526	0.346	0.346	0.927	-0.414	-0.414	-0.385	-0.378	-0.658	-0.549	0.346
Clipboard	9	-0.125	0.122	0.227	0.125	0.125	0.395	0.192	0.133	1.000	0.308	0.316	-0.205	0.395
BrowserExample	14	0.914	-0.278	-0.192	-0.200	0.447	0.878	-0.201	-0.201	1.000	-0.231	-0.205	-0.166	-0.155
DND	14	0.684	-0.310	-0.190	0.563	0.829	0.445	-0.213	-0.147	0.956	-0.350	0.226	-0.133	0.477
TextEditor	14	0.101	-0.353	0.218	0.613	0.491	0.316	-0.105	-0.104	0.494	-0.055	0.177	0.204	0.651
Address Book	16	0.067	-0.183	-0.158	-0.098	0.333	0.228	-0.228	-0.228	-0.228	-0.101	0.308	0.194	-0.333
JavaViewer	20	-0.026	-0.374	0.488	0.296	0.621	0.809	-0.222	-0.622	-0.621	-0.015	0.241	0.052	0.160
ImageAnalyser	26	0.519	-0.317	0.556	0.010	0.455	0.581	-0.186	-0.185	0.664	-0.268	0.008	-0.222	0.478
LayoutExample	35	0.643	0.546	0.652	-0.222	-0.368	0.705	-0.612	-0.612	-0.599	-0.003	0.040	0.176	0.055
TraceToUCM	66	0.124	-0.133	-0.078	-0.124	0.371	0.168	-0.173	-0.259	0.739	-0.422	-0.472	-0.576	0.311
PaintExample	104	0.351	-0.074	0.521	-0.261	0.119	0.537	0.111	0.112	0.164	-0.240	-0.239	-0.269	0.293
Graphics	213	0.157	-0.165	0.011	0.422	-0.124	0.081	-0.039	-0.039	0.005	-0.299	-0.126	-0.332	-0.045
Control	292	0.298	0.055	0.328	0.156	0.186	0.439	-0.157	-0.157	-0.120	-0.049	-0.007	0.094	0.070
jucmnav	305	0.232	-0.045	0.177	0.099	0.060	0.302	-0.002	-0.002	0.116	0.047	0.220	0.124	-0.292
Umple	351	0.187	-0.164	0.019	0.129	0.072	0.623	-0.003	0.000	0.130	-0.043	0.041	-0.002	0.016
Uced	458	0.207	-0.080	0.284	0.064	0.054	0.704	-0.015	-0.015	0.192	-0.080	0.087	0.102	0.015
JHotDraw	566	0.291	-0.143	0.346	0.230	0.139	0.366	-0.018	-0.018	0.113	-0.132	-0.017	-0.051	-0.109
		0.249	-0.096	0.232	0.139	0.102	0.456	-0.046	-0.050	0.114	-0.104	0.016	-0.023	-0.018

Table 7 Correlation of static and dynamic variables with average call hierarchy, for all systems.

In Table 7, the average depth of call is correlated with fan-in (0.249), and TotalSegmentationPresence (0.456). All of the other correlations are very weak.

4.2.5 Segmentation and Clustering

Another variable we use is segmentation data. Dugerdil and Jossi [30] noticed that some classes were present in most of the segments of an execution trace. These classes contain methods that perform some utility work, not specific to any functional component.

Dugerdil and Jossi define the clustering algorithm as follows: Let us define the number of segments in the trace as N_s and the binary occurrence vector VC for a given class C as a vector whose size is N_s and whose i^{th} component indicates the presence (1) or absence (0) of the class in the i^{th} segment. Then, the correlation between any two classes $C1$ and $C2$ is given by:

$$C(V1, V2) = V1.V2 / (V1[i] \text{ OR } V2[i])$$

where $V1.V2$ is the usual dot product for vectors and $V1[i] \text{ OR } V2[i]$ is the Boolean OR operator used between the corresponding components of both vectors.

In this case, we are not interested in the correlation value but rather in the binary occurrence of a method C in all the segments. In other words, we are only interested in the vectors for each of the methods. A method that is present in all segments will have the sum of its vectors be N_s . A method that is only present in one segment will have a value of 1.

We use this opportunity to test Dugerdil and Jossi’s observation that the more frequent the occurrence of a class C is in all the segments, the more likely a method is a *utility*. “Classes that are present in most of the segments of the execution trace... were classes that perform some utility work, not specific to any functional component” [30]. Their paper focused on trace summarization and did not explore the idea of utilities too far.

4.3. Static Data Collected

In this section, we look at the data collected by static analysis. Static analysis examines a program’s code to derive properties that hold for all executions [9]. The static analysis is performed without actually executing programs. Table 8 shows the list of relevant static

data collected using Metrics 1.3.6 [82]. The following sub-sections give a more detailed definition of these static variables and explain why we chose to use them.

Static Data Collected	Description of Static Data Collected
Method Lines of Code (MLOC)	Total number of lines of code inside method bodies, excluding blank lines and comments.
Nested Block Depth (NBD)	The depth of nested blocks of code.
McCabe Cyclomatic Complexity (McCabe)	Number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and conditional logic operators in expressions), this metric is incremented by one. Calculated for methods only. For a full treatment of this metric, see Section 4.3.1.

Table 8 Static data collected by using Metrics 1.3.6.

Even though we had collected a lot of static variables, only the static variables that were found to describe utility methods are used in this thesis.

4.3.1 McCabe Cyclomatic Complexity

Cyclomatic complexity is a measure of software complexity, and is at times referred to as conditional complexity or McCabe complexity index. It was developed by Thomas J. McCabe in 1976 [64][69]. It measures the number of linearly independent paths through a method. Cyclomatic complexity is computed using the control flow graph of the program (or more likely a method in OO programs). In his seminal paper [81], McCabe explains that the nodes of the graph correspond to the commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Using the flow graph, the cyclomatic number $V(G)$ of a graph G with n vertices, e edges, and p connected components is:

$$V(G) = e - n + p$$

McCabe showed that $V(G)$ is also equal to the number of binary decision nodes in G plus one. There are four basic rules that are used to calculate the McCabe complexity index [52]:

1. Increment one for every IF, CASE or other alternate execution construct;
2. Increment one for every iterative DO, DO-WHILE or other repetitive construct;

3. Add two less than the number of logical alternatives in a CASE;
4. Add one for each logical operator (AND, OR) in an IF statement.

Based on our definition of *utilities* as elements of a program designed for the convenience of the designer, utilities will tend to be reused and accessed from multiple places within a certain scope of the program. We hypothesize that methods that are designed to be accessed from multiple places tend to be simple. It then follows those methods that have high *fan-in value* would be low on the McCabe complexity index.

To test our hypothesis, we calculated correlation coefficients to test what other characteristics correlate with the complexity index of methods. Table 9 shows the correlation of several static and dynamic variables with McCabe complexity indexes for 18 different systems. Since the programs vary in size, we calculated the weighted average coefficient.

System name	Number of methods	Dynamic Data								Static		
		UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	MLOC	NBD	PAR
HelloWorld	2	0.093	0.177	0.042	-0.126	-0.411	0.146	0.000	-0.294	0.483	0.496	0.296
HoverHelp	6	-0.189	0.615	0.092	-0.658	-0.586	-0.586	-0.348	0.926	0.022	0.612	-0.586
Clipboard	9	0.316	0.387	0.419	0.316	-0.316	-0.316	0.350	0.050	0.663	0.637	0.350
BrowserExample	10	-0.109	0.034	-0.112	-0.205	-0.321	-0.370	0.271	-0.144	0.991	0.558	0.890
DND	14	0.750	0.006	0.060	0.226	-0.127	-0.187	0.569	-0.094	0.733	0.709	0.249
TextEditor	14	-0.027	-0.410	0.373	0.177	-0.166	0.670	0.594	0.020	0.165	0.723	0.375
Address Book	16	-0.034	-0.012	-0.141	0.308	-0.150	0.011	-0.206	0.162	0.254	0.625	-0.331
JavaViewer	20	0.205	0.239	0.773	0.241	-0.195	-0.177	0.528	-0.057	0.907	0.759	0.072
ImageAnalyser	26	0.035	0.088	-0.090	0.008	0.093	0.116	-0.145	-0.105	0.291	0.647	0.073
LayoutExample	35	0.039	-0.103	0.107	0.040	-0.143	0.025	0.056	0.013	0.422	0.399	0.627
TraceToUCM on jhotdraw TPTP	66	-0.480	0.496	0.466	-0.472	-0.045	-0.737	-0.183	0.544	0.888	0.844	-0.077
PaintExample	104	-0.066	0.180	-0.024	-0.239	-0.355	-0.222	-0.122	0.006	0.574	0.736	0.059
Graphics	213	0.093	0.177	0.042	-0.126	-0.411	0.146	0.000	-0.294	0.483	0.496	0.296
Control	288	0.017	0.077	0.085	-0.007	-0.289	-0.241	0.077	0.001	0.523	0.578	0.044
jucmnav TPTP Nov26AfterExclu	305	0.145	-0.006	0.191	0.220	-0.197	0.046	0.328	0.027	0.643	0.487	0.011
Umple_compiling BCMS TPTP	351	-0.091	0.525	-0.025	0.041	-0.271	-0.293	-0.067	0.012	0.605	0.627	0.267
Uced TPTP open file Nov12Af	458	-0.039	0.088	0.040	0.087	-0.209	-0.112	0.026	-0.025	0.701	0.701	0.281
JHotDraw	566	-0.055	0.151	0.008	-0.017	-0.256	-0.200	0.076	-0.036	0.699	0.766	0.090
		-0.013	0.170	0.063	0.016	-0.252	-0.146	0.062	-0.017	0.628	0.647	0.156

Table 9 Correlation of static and dynamic variables with McCabe, for all systems.

The weighted average coefficients show that methods with a high complexity index also tend to have high MLOC values and higher nested block depth. At values of 0.628 and 0.648, both of the weighted averages are the strongest. In addition to the weighted average coefficient, MLOC and NBD are stronger for almost all of the systems individually. On the negative side, we see high negative correlation with the data access column, where the coefficient is -0.250. Methods that are data access methods (getters and setters) tend to have low complexity. Therefore, we made the decision to create an algorithm that uses MLOC, McCabe and NBD and the inverse of fan-out to detect utilities.

4.3.2 Method Lines of Code (MLOC)

Method lines of code (MLOC) are a count of non-blank and non-comment lines in the body of a method. We hypothesize that utility methods will tend to be relatively smaller than other methods in the same application. It then follows that methods with a high *fan-in value* would be smaller.

To verify this idea, we calculated the correlation between the MLOC values of methods in the execution trace with several of the dynamic and static parameters that we collected. The results of the correlation are presented in Table 10.

The weighted average coefficient shows that methods with high complexity indexes also tend to have a high fan-out value (0.3), high complexity index (0.627), and high nested depth (0.530). It is also worth noting that as well as the weighted average coefficient, both McCabe and NBD correlation coefficient are strong for almost all of the systems individually.

On the negative side, we see a negative correlation with the data access column (where the coefficient is -0.259). Methods that are data access methods (getters and setters) tend to have low MLOC. Therefore, we made the decision to create an algorithm that uses MLOC, McCabe and NBD and the inverse of fan-out to detect utilities.

System name	Number of methods	Dynamic Data								Static Data		
		UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	McCabe	NBD	PAR
HelloWorld	2	-0.019	0.261	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	0.483	0.570	0.473
HoverHelp	6	0.012	-0.088	-0.140	-0.378	-0.442	-0.442	-0.381	-0.127	0.022	0.595	-0.442
BrowserExample	9	-0.146	0.081	-0.075	-0.231	-0.272	-0.445	0.256	-0.108	0.991	0.446	0.932
ClipBoard	9	0.466	0.195	0.575	0.308	-0.466	-0.493	0.678	-0.201	0.663	0.593	0.066
DND	14	0.338	0.198	0.300	-0.350	-0.341	-0.502	0.225	-0.110	0.733	0.733	-0.225
TextEditor	14	0.253	-0.325	-0.013	-0.055	-0.308	-0.076	0.228	-0.260	0.165	0.437	-0.213
Address Book	16	0.179	0.198	0.525	-0.101	-0.341	-0.582	0.346	0.695	0.254	0.720	-0.077
JavaViewer	20	0.276	0.262	0.647	-0.015	-0.262	-0.235	0.318	-0.099	0.907	0.879	0.009
ImageAnalyser	26	0.025	0.137	-0.128	-0.268	-0.108	-0.280	-0.143	-0.102	0.291	0.504	-0.250
LayoutExample	35	-0.022	0.209	0.015	-0.003	-0.298	-0.438	-0.039	-0.155	0.422	0.702	0.046
TraceToUCM_on_jhotdraw_TPTP	66	-0.753	0.765	0.716	-0.422	-0.039	-0.691	-0.185	0.787	0.888	0.743	-0.106
PaintExample	104	-0.089	0.306	-0.059	-0.240	-0.261	-0.199	-0.176	-0.057	0.574	0.711	0.016
Graphics	213	-0.019	0.261	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	0.483	0.570	0.473
Control	287	-0.012	0.165	0.154	-0.049	-0.422	-0.415	0.095	-0.018	0.523	0.553	0.056
jucmnav_TPTP_Nov26AfterExclu	305	0.028	0.107	0.252	0.047	-0.183	-0.066	0.134	0.028	0.643	0.335	0.146
Umple_compiling_BCMS_TPTP	351	-0.031	0.695	-0.001	-0.043	-0.059	-0.078	-0.029	0.005	0.605	0.270	0.113
Uced_TPTP_open_file_Nov12Af	458	-0.075	0.151	-0.019	-0.080	-0.251	-0.099	-0.083	-0.039	0.701	0.640	0.232
JHotDraw	566	-0.078	0.350	0.013	-0.132	-0.268	-0.252	0.005	-0.008	0.699	0.591	0.103
		-0.051	0.299	0.074	-0.104	-0.261	-0.205	-0.002	0.005	0.627	0.530	0.142

Table 10 Correlation of static and dynamic variables with MLOC, for all systems.

4.4. Summarization Process

Using the static and dynamic data described in Sections 4.2 and 4.3, we filter a trace to remove its utilities and then visualize it as a UCM diagram. The following is a brief summary of the main steps taken by the trace summarization tool. A more in-depth coverage is given in Chapter 8.

1. Run the program and collect the execution trace.
2. Collapse loops
3. Remove access methods (labelled .get or .set and have less than 2 lines of code and complexity index less than 2.
4. Remove small methods (with less than 5 lines of code) and simple methods (that score 2 on the complexity index).
5. Obtain the static data (from Metrics 1.3.6) and CPU data (from TPTP) for the methods.
6. Process the trace and extract the dynamic data.

7. Filter the traces using different combination of the static and dynamic data presented in this chapter.
8. Generate UCMs from the resulting traces. In the next section, we discuss how trace elements are mapped to UCM elements.
9. Validate the results with the original developers of the systems.

Figure 5 gives a visual summary of the execution trace reduction process.

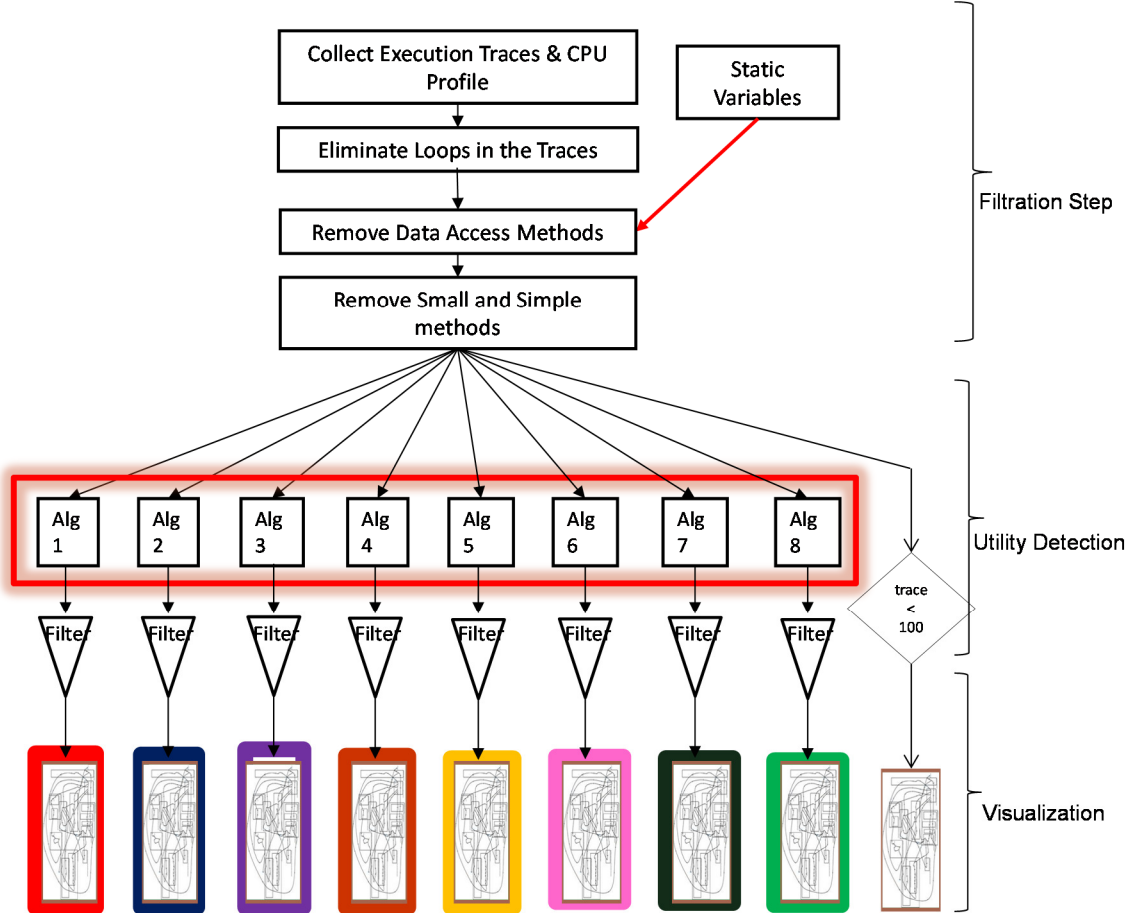


Figure 5. Summary of the algorithm comparison process.

The validation step might lead to further filtering of the trace if the software engineer judges that the trace still contains too much detail.

4.5. Visualization of Reduced Traces using UCM

We chose Use Case Maps (UCM) to visualize the reduced execution traces because they are a rich requirements-level notation for showing at a glance the various control-flow

possibilities in a system. Unlike UML message sequence diagrams, which are often used for trace visualization, UCMs abstract from the inter-component communication to focus on the business/feature logic. Using jUCMNav, we are also able to generate MSC diagrams and compare the understandability of the same scenario represented in the two different formats.

Like UML activity diagrams, UCMs can integrate many scenarios with operators for looping, forking and joining alternative or concurrent paths. Complex maps can also be decomposed into sub-maps (with dynamic or static stubs). Stubs may also contain multiple sub-maps, allowing for flexible integration and exploration of scenarios that have overlapping parts.

Each item in the filtered execution trace is visually communicated using UCM. To translate execution traces to UCM maps, we provide a mapping from trace elements to UCM notation elements in Table 11.

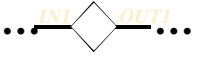
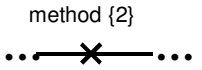
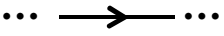
Trace element	UCM element	Symbol
Package	Shown as a rectangle with thick border.	
Class	Shown as a rectangle in another rectangle (Package).	
Beginning / End of trace	Start point (circle) / End point (bar) (also used as connectors for linking sub-scenarios to the parent stub)	
Method	Method name (shown as a X on a path)	
Block of x or more instructions in the same class/object	Stub (diamond) with the name of the first instruction that is not a constructor. This stub contains a plug-in (another sub-map) showing the sub-sequence with one responsibility per instruction. The number of instructions that are placed in a block is set in the TraceToUCM tool.	
Repeated method	Method name (repetition count between curly brackets)	
Order of trace items	Direction of scenario flow (shown as an arrow head). We use the order of trace items in execution traces to dictate the direction of the scenario flow.	

Table 11 Mapping of trace elements to UCM notation elements.

We developed a tool, TraceToUCM, that uses the mapping to generate UCM diagram automatically. We use the same layout program to automatically draw all the UCM diagrams, which helps keep the layout variable consistent between the UCM diagrams ob-

tained from different utility detection and filtering algorithms. We chose to do direction arrow manually because most of the time they are not needed.

4.6. Chapter Summary

In this chapter, we discussed the static and dynamic data used for the studies that aim to detect utility methods among all the methods used in one execution of a feature. We looked at the approach taken to collect and analyze static and dynamic data from the software systems. We also discuss why we decided to use the particular parameters we chose to use for the experiment. In the next chapter we will see the algorithms developed to detect *utility* components.

Chapter 5. Utility Detection Algorithms

In this chapter, we describe the algorithms developed to detect *utility* components. We develop several algorithms using a combination of static data and dynamic parameters as presented in Chapter 4. Some of the parameters give us information specific to the method like its size or complexity, and others measure the interaction of the method with other methods. Table 12 gives a summary of the parameters collected. The parameters fall into several different groups which include static, dynamic, method localized (internal measures of the method), and the method interaction (how they relate to other methods).

	Static	Dynamic	Method-specific	Interaction of a method with other methods
Fan-in		X		X
Fan-out		X		X
MLOC	X		X	
McCabe	X		X	
Segmentation Presence		X		X
Nested Block Depth	X		X	
Depth of Call		X		X
CPU		X	X	

Table 12 Types of parameters collected and used for the algorithms.

We use different combinations of the parameters in Table 12 to develop the algorithms to calculate *utilityhood* values ($U(r)$) for each method r in the execution traces. Table 13 gives a summary of the eight algorithms. In the rest of this chapter, we look at these algorithms one by one.

Algorithm	Utilityhood Function
Algorithm 1	$U(r) = \frac{UniqueFanin(r)}{N - 1}$
Algorithm 2	$U(r) = \frac{UniqueFanin(r)}{N} \times \frac{\log(\frac{N}{UniqueFanout(r) + 1})}{\log(N)}$
Algorithm 3	$U(r) = (\frac{1}{Nbd(r)}) * (\frac{1}{McCabe(r)}) * (\frac{1}{RelativeMethodSize(r)})$
Algorithm 4	$U(r) = fanout(r) * (\frac{1}{Nbd(r)}) * (\frac{1}{McCabe(r)}) * (\frac{1}{RelativeMethodSize(r)})$
Algorithm 5	$U(r) = \frac{SegmentPresence(r)}{TotalNumberOfSegments}$
Algorithm 6	$U(r) = (\frac{1}{((UniqueFanin(r) * UniqueFanout(r))^2)})$
Algorithm 7	$U(r) = \frac{UniqueFanin(r)}{N - 1} * (\frac{1}{AverageCallHierarchy}) * (\frac{1}{RelativeMethodSize(r)})$
Algorithm 8	$U(r) = \frac{1}{CpuUsageAverage(r)} * (\frac{1}{McCabe(r)}) * (\frac{1}{RelativeMethodSize(r)})$

Table 13 Summary of the eight algorithms.

5.1. Algorithm 1: Fan-in

This algorithm is adapted from preliminary work [47], where we used static fan-in of methods and classes to calculate *utilityhood* with the goal of understanding the behaviour of the whole software system. We analyzed an execution trace generated from an object-oriented system called TConfig (version 2.1) [116]. We used our own instrumentation tool based on the BIT framework [71] to insert probes at the entry and exit points of each system's non-private methods and branch of every condition statement. We improve on

the preliminary work by automating the trace processing and generation of UCM diagrams. For this thesis, our goal is to understand the behaviour of one feature of a software system. We chose to use dynamic fan-in because we only want information on methods that are invoked by the execution of that one feature.

Given a method r and the following:

- N = Number of unique methods in the execution trace. We calculate N by going through the execution trace and counting methods called. We count overloaded methods as one method.
- $UniqueFanin(r)$ = Number of unique methods in the execution trace, other than r (recursive method calls are ignored), that call r . A method that makes multiple calls to r is only counted once.

The *utilityhood* metric U of the method r is defined as:

$$U(r) = \frac{UniqueFanin(r)}{N - 1}$$

$U(r)$ ranges from 0 to 1. A value of 0 indicates that r is not called by any method and is therefore not a *utility*. A good example would be the method *main()*. A value of 1 would mean that the method is called by all N methods. $N-1$ is the scope considered in the computation of utilities. We use $N-1$ as the denominator so that self-dependency is ignored since in recursive calls the caller and the callee are the same.

5.2. Algorithm 2: Fan-in and Fan-out

For the second algorithm, we use an algorithm developed by Hamou-Lhadj [46] based on fan-in analysis on a call graph generated by static analysis of the system. We adapt the algorithm to use dynamic fan-in and fan-out because we are interested in the methods that are invoked for one execution. Instead of deriving the fan-in and fan-out variables from a static call tree, as were done by Hamou-Lhadj [46], we derive the information from the execution trace. A method, say method1 for example, that did not make a call to another method, say method2, in this particular run of the software will have 0 fan-out, even if

method1 includes a call to method2 in the portion of the code that did not get executed in this run.

Given a method r and the following:

- N = Number of unique methods in the execution trace. We calculate N by going through the execution trace and counting methods called. We count overloaded methods as one method.
- $UniqueFanin(r)$ = Number of unique methods in the execution trace that called r . A method that made multiple calls to r is only counted once. Recursive method calls are ignored.
- $UniqueFanout(r)$ = Number of unique methods in the execution trace that are called by r . A method that is called multiple times is only counted once. Recursive method calls are ignored.

The *utilityhood* metric U of the method r is defined as:

$$U(r) = \frac{UniqueFanin(r)}{N} \times \frac{\log(\frac{N}{UniqueFanout(r) + 1})}{\log(N)}$$

The first part of the equation, $UniqueFanin(r)/N$, simply reflects that routines with large fan-in are the ones that are most likely to be utilities. The second part of the equation looks at the number of routines that are called by r . The log of $N/(UniqueFanout(r) + 1)$ is used to deemphasize fan-out. Therefore, the first part is multiplied by a coefficient that takes into account fan-out, although with lower emphasis.

5.3. Algorithm 3: McCabe

For the third algorithm, we use dynamic fan-in from Section 5.1 and divide it by the McCabe value discussed in Section 4.3.1. The McCabe value, also known as cyclomatic complexity, measures the number of linearly independent paths through a method. One of the hypotheses for this work is that methods that are simple are more likely to be utilities than methods that are more complex.

We calculated correlation coefficients to test what other characteristics correlate with the complexity index of methods. Table 9 shows the correlation results. The

weighted average coefficient shows that methods with a high complexity index also tend to have a high nested block depth (NBD) value (0.647), a high complexity index (0.628), and a high nested depth (0.530). It is also worth noting that just as it is for the weighted average coefficient, both MLOC and NBD are significant for almost all of the systems separately.

On the negative side, methods that are data access methods (getters and setters) tend to have low complexity, and since their coefficient is -0.252, we have not included the variable in the algorithm because all data access methods are labelled utility and filtered out.

We developed the following algorithm to filter out utility methods that are small and simple.

Given a method r and the following:

- **$McCabe(r)$** = A count of linearly independent paths through the method r .
- **$Nbd(r)$** = The depth of nested blocks of code in method r .
- **$MLOC(r)$** = A count of non-blank and non-comment lines in the body of a method r .
- **$MaxMLOC$** = The size of the largest method in the trace, according to the same rules as **$MLOC$** .
- **$RelativeMethodSize(r)$** = The size of a method r in relation to other methods in the system.

We use the inverse of McCabe because the simpler the method is, the more likely that it is a *utility*.

$$Mc(r) = 1 / McCabe(r)$$

We use the inverse of relative size with the belief that the smaller a method, the more likely it is to be a utility.

$$RelativeMethodSize(r) = MLOC(r) / MaxMLOC$$

$$MLocInv(r) = 1 / RelativeMethodSize(r)$$

We use the inverse of nested depth with the belief that the smaller the depth of code, the more likely it is to be a utility.

$$Nd(r) = 1/Nbd(r)$$

Then we take the product of these three parts:

$$U(r) = Nd(r) * Mc(r) * MLocInv(r)$$

5.4. Algorithm 4: MLOC

For the fourth algorithm, we once again use dynamic fan-in from Section 5.1. We hypothesize that smaller methods are more likely to be *utilities* than larger ones, and therefore methods that are designed to be accessed from multiple places would tend to be small.

We use the relative size of a method within the whole system as a measure, instead of an absolute size, to account for differences in programming style, and the size of the whole system. A method considered large in one software system might be the smallest in another. We calculated correlation coefficients to test what other characteristics correlate with the size of methods. Table 10 shows the correlation results. The weighted average coefficient shows that methods that are relatively large also tend to have a high fan-out value (correlation coefficient = 0.299), high complexity index (correlation coefficient = 0.627), and high nested depth (correlation coefficient = 0.530).

The only difference between the parameters that correlate with McCabe and MLOC is the fan-out variable. Therefore we simply take the McCabe algorithm from 5.3 and multiply it by fan-out.

We developed the following algorithm to filter out utility methods that are small, simple, and do not make a lot of method calls.

Given a method r and the following:

- N = Number of unique methods in the execution trace. We calculate N by going through the execution trace, and counting methods called. We count overloaded methods as one method.
- $UniqueFanout(r)$ = Number of unique methods in the execution trace that are called by r . A method that is called multiple times is only counted once.

We use the inverse of fan-out because the higher the number of methods that are called by r , the less likely it is a *utility*. We use $N-1$ as the denominator so that self-dependency is ignored since in recursive calls the caller and the callee are the same.

$$fanout(r) = \frac{N - 1}{UniqueFanout(r)}$$

Then multiply algorithm 4 by Fan-out:

$$U(r) = fanout(r) * (Nd(r) * Mc(r) * MLocInv(r))$$

5.5. Algorithm 5: Segment Presence

The fifth algorithm is based on Dugerdil’s work [30]. Dugerdil *et al.* developed trace segmentation and clustering technique to reverse-engineer software systems. While segmenting the execution trace, the authors remove classes that are present in most of the segments of the execution trace. They observe those classes “..perform some utility work, not specific to any functional component” [30]. We take the trace segmentation part of their methodology to find out if their technique of detecting utilities works for methods. We test if methods that appear in many segments of a trace are in fact *utility methods*. The assumption is that traces that are called in only 1 segment of the trace are less likely to be *utilities* than methods that are called in N segments (where N is the total number of segments the trace is broken into). Therefore the algorithm is more about density and cluster than number of times the method is called.

To calculate segmentation presence, the trace is first split into contiguous segments of equal sizes. Dugerdil *et al.* [30] chose the number of segments to be twice the number of classes present in the execution trace. We changed the formula so that the number of segments is dependent on the size of the trace file to be segmented. Table 14 gives details of how we calculate the number of segments in a trace; the number of segments is a percentage of the number of lines in the trace whose value is a function of the size of that trace. Then we check how many of the segments contain a given method. The more frequent a method is found in an execution trace, the more likely it is a *utility*. If a method r is present in all the segments, then its *utilityhood* value will be 1.

Given a method r and the following:

- *SegmentPresence(r)* = Number of segments that method *r* is found in.
- *TotalNumberOfSegments*: The total number of segments the execution trace is broken into. The way we calculate *TotalNumberOfSegments* depends on the size of the execution trace. For example, a trace file that has 10 or less lines is not divided and will have *TotalNumberOfSegments* set to 1 which means the lines per segment will be equal to the lines in the execution trace. A trace file that has 80,000 lines will be segmented into 80 parts of 1,000 lines each.

Number of lines in the execution trace	Percentage of number of lines	Number of segments	Lines per segment
1 – 10	-	1	~1 - 10
11 – 100	20%	2 – 20	~5
101 – 1,000	10%	10 – 100	~10
1,001 – 10,000	5%	50 – 500	~20
10,001 – 50,000	1%	100 – 500	~100
50,001 – 100,000	0.1%	10 – 100	~1,000
100,001 – 1,000,000	0.01%	10 – 100	~10,000
1,000,001 and up	0.001%	10 – 100	100,000+

Table 14 Trace segmentation table.

The *utilityhood* metric *U* of the method *r* is defined as:

$$U(r) = \frac{SegmentPresence(r)}{TotalNumberOfSegments}$$

Note: The higher the number of segments, the more unlikely methods will be found in high number of segments and hence the more unlikely we would have high *U(r)* values. Also the transitions are not smooth. For example, a trace of 999,999 lines will be divided in segments of 10,000 lines each, whereas one of 1,000,002 lines (only 3 more) will result in segments of 100,000 each.

5.6. Algorithm 6: High Fan-in and High Fan-out (Yuying)

For the sixth algorithm, we take an algorithm used by Yuying *et al.* [118], overviewed in Section 3.2.1. Yuying *et al.* believe that the static (syntactical) situation of a software program reflects only inaccurately the situation of the dynamic behaviour of the system, using factors like actual number and type of procedure calls, as well as size of the actual

transferred information, etc. These authors hypothesize that methods with high fan-in or fan-out values implement the system's main functions, and can be used to infer the subject system's functionality. They call components that implement the main functions or main structure of a system's *critical* components. Their formula is used to calculate *IF*(information flow).

- ***UniqueFanin(r)*** = Number of unique methods in the execution trace that called *r*. A method that made multiple calls to *r* is only counted once. Recursive method calls are ignored.
- ***UniqueFanout(r)*** = Number of unique methods in the execution trace that are called by *r*. A method that is called multiple times is only counted once. Recursive method calls are ignored.

IF of the method *r* is defined as follows:

$$IF(r) = (UniqueFanin(r) * UniqueFanout(r))^2$$

We invert the formula Yuying *et al.* [118] use to detect the “main functions” and to find and filter out the functions (methods that are not “main”). By inverting the equation the methods that have fan-in and fan-out will be labeled as utilities and are to be filtered out.

$$U(r) = \left(\frac{1}{((UniqueFanin(r) * UniqueFanout(r))^2)} \right)$$

We would like to compare if it is better to measure interface complexity than measuring the complexity of the modules, therefore we have taken the formula directly from [118].

5.7. Algorithm 7: Average Call Tree Depth

For this algorithm, we look at each method in the execution trace, and determine and store a value that represents the average call hierarchy, also called average call tree depth. The average call hierarchy is calculated by taking the depth at which each method is called during the run, and then dividing it by the number of different depths from which it is called. For example, if method A is called by Z()->Y()->A(), which is depth 3, and also

called by $Z() \rightarrow X() \rightarrow W() \rightarrow A()$, which is depth 4, then the *AverageCallHierarchy* would be $(3+4)/2 = 3.5$.

We calculated the correlation of average depth of a call in relation to other static and dynamic data we collected. In Table 7, we see that average depth of call is correlated with fan-in, with a 0.246 correlation coefficient. In Table 10 we see that depth of a call is slightly negatively correlated with the size of a method, with -0.102 as correlation coefficient. The deeper a method is in the call tree, the less likely it is to be a big method. Therefore we develop an algorithm using fan-in and MLOC, in conjunction with average call tree, to filter out utilities.

Given a method r and the following:

- ***AverageCallHierarchy(r)*** = The average depth that r is called at in the execution trace.
- ***MLOC(r)*** = A count of non-blank and non-comment lines in the body of a method r .
- ***MaxMLOC*** = The size of the largest method in the trace, according to the same rules as *MLOC*.
- ***RelativeMethodSize(r)*** = The size of a method r in relation to other methods in the system.
- N = Number of unique methods in the execution trace. We calculate N by going through the execution trace and counting methods called. We count overloaded methods as one method.
- ***UniqueFanin(r)*** = Number of unique methods in the execution trace that called r . A method that made multiple calls to r is only counted once.

$$RelativeMethodSize(r) = MLOC(r)/MaxMLOC$$

We use the inverse of relative size with the belief that the smaller a method is, the more likely it is to be a utility. We also use the inverse of average call hierarchy because the smaller the depth, the less likely the method is a utility.

The *utilityhood* metric U of the method r is defined as:

$$U(r) = \frac{UniqueFanin(r)}{N - 1} * (\frac{1}{AverageCallHierarchy}) * (\frac{1}{RelativeMethodSize(r)})$$

The smaller the average depth, the fewer the methods that call it and the larger the method, the less likely it is a utility.

5.8. Algorithm 8: Calculate Utilityhood CPU

This algorithm is used to pick the methods that take the least amount of CPU time, along with being the smallest and the least complicated, and to label them as *utility* methods. The most complex methods with the biggest average time, and with the most lines of code, are selected and used to draw the UCM.

Given a method r and the following sets:

- **McCabe(r)** = A count of linearly independent paths through the method r .
- **MLOC(r)** = A count of non-blank and non-comment lines in the body of a method r .
- **MaxMLOC** = The size of the largest method in the trace, according to the same rules as **MLOC**.
- **RelativeMethodSize(r)** = The size of a method r in relation to other methods in the system.
- **BaseCpuUsage** = For any invocation, the base CPU usage is the time taken to execute the invocation, excluding the time spent in other methods that were called during the invocation (this is provided by TPTP).
- **AverageCpuUsage(r)** = the base time divided by the number of calls (this is provided by TPTP).

We use the inverse of McCabe because the simpler the method is, the more likely that it is a *utility*.

$$Mc = 1 / McCabe(r)$$

We use the inverse of relative size with the belief that the smaller a method is, the more likely it is to be a utility.

$$RelativeMethodSize(r) = MLOC(r)/MaxMLOC$$

$$MLOC(r) = 1/RelativeMethodSize(r)$$

We use the inverse of average CPU usage with the belief that the higher the CPU usage, the less likely that the method is a utility.

$$cpu(r) = 1/CpuUsageAverage(r)$$

$$U(r) = cpu(r) * Mc(r) * MLoc(r)$$

5.9. Chapter Summary

In this chapter, we describe the different algorithms that are used to calculate the *utility* values for the methods in the execution trace. In the next chapter, we describe the tool we created to automate the methodology we developed.

Chapter 6. Tool Support

6.1. Overview

In this chapter, we describe the suite of tools that are used to transform execution traces to UCM diagrams. Figure 6 gives an overview of the tools and their interactions. The main tool is TraceToUCM. It is written in Java and configured to run in Eclipse [32].

6.2. Inputs

The TraceToUCM tool takes up to six text files as input. The three required files are an execution trace generated via TPTP, its associated static data generated via Metrics 1.3.6 in XML and converted to a tab-delimited file, and a CPU usage file also generated via TPTP in XML and converted to tab delimited text. Related percentage distribution, item exclusion, and item inclusion files are optional. As an example, the six input files used to generate the behaviour model of ‘Umple while Compiling BCMS’ are described in greater detail in the following sub-sections.

6.2.1 Execution Trace Input File

The first argument of the tool is a file containing an execution trace generated using TPTP. The execution trace input file is labeled l1 in Figure 6. The execution trace contains the method name and call hierarchy of each method that has been called in the execution. Each entry and exit from a method results in an entry into the trace file labelled “Entered” or “Exited”. The trace line designated “Entered” also includes the depth of the call in the call hierarchy. Figure 7 shows a few lines of a sample trace file. The execution trace is a mandatory input parameter for the tool.

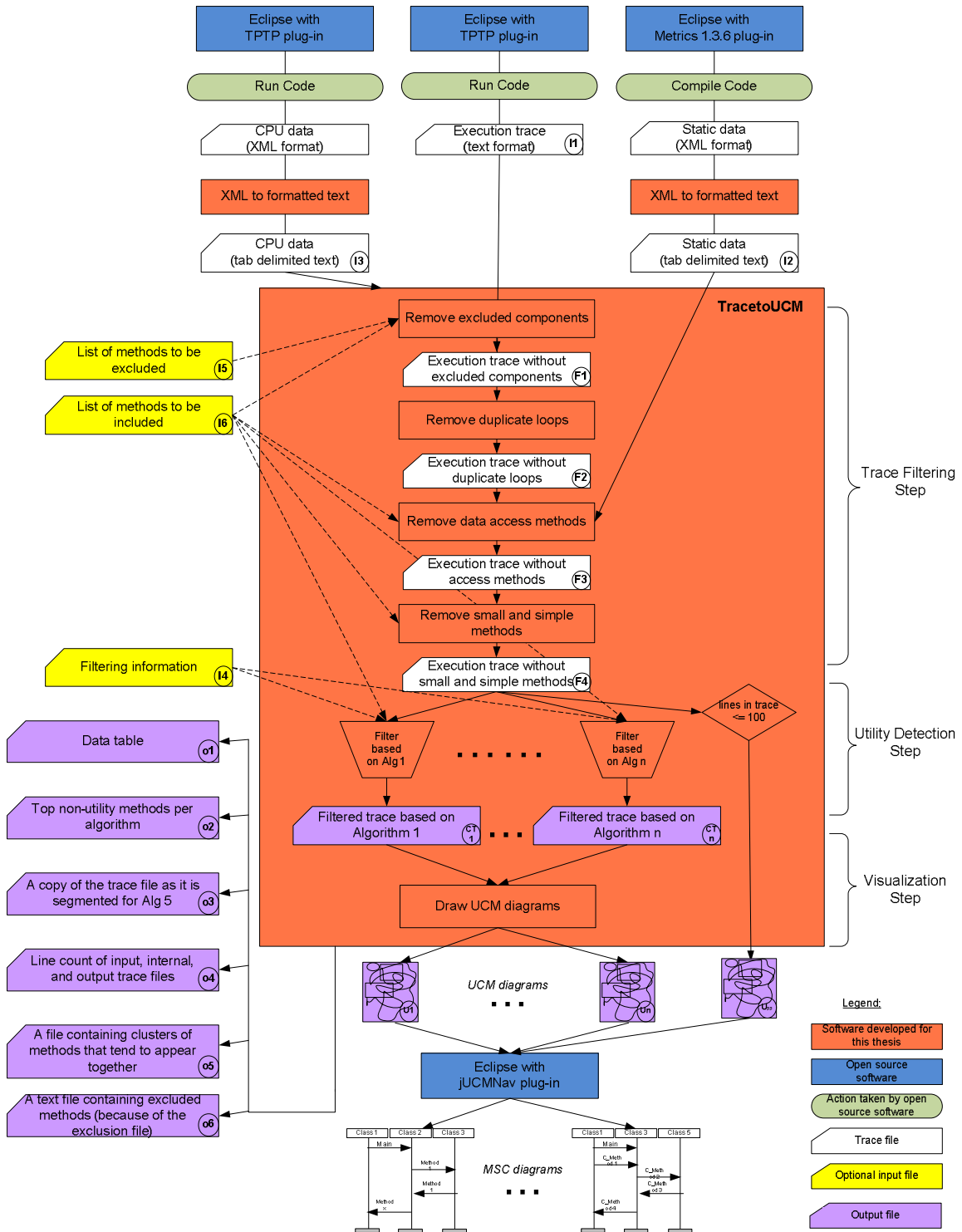


Figure 6. Suite of tools and their interactions.

```

main 0
Entered: cruise/umple/UmplesConsoleMain.optParse 1
Entered: joptsimple/OptionParser.<init> 2
Entered: joptsimple/BuiltinHelpFormatter.<init> 3
Exited: joptsimple/BuiltinHelpFormatter.<init>
Entered: joptsimple/internal/AbbreviationMap.<init> 3
Exited: joptsimple/internal/AbbreviationMap.<init>
Entered: joptsimple/OptionParserState.moreOptions 3
Entered: joptsimple/OptionParserState$2.<init> 4
Entered: joptsimple/OptionParserState.<init> 5
Exited: joptsimple/OptionParserState.<init>
Exited: joptsimple/OptionParserState$2.<init>
Exited: joptsimple/OptionParserState.moreOptions
Exited: joptsimple/OptionParser.<init>
Entered: joptsimple/OptionParser.acceptsAll 2
Entered: joptsimple/ParserRules.<clinit> 3
Exited: joptsimple/ParserRules.<clinit>
Entered: joptsimple/ParserRules.ensureLegalOptions 3
Entered: joptsimple/ParserRules.ensureLegalOption 4

```

Figure 7. Input parameter 1 – Execution trace.

6.2.2 Static Data Input File

The second file (labeled l2 in Figure 6) is a space delimited text file containing the static information about the program being studied. The data is extracted using Metrics 1.3.6 and saved as XML. Figure 8 shows a sample static data output for the Umples trace. We wrote a simple program to automatically transform the XML file into a delimited text file in order to simplify processing.

```

- <Metrics scope="cruise.umple" type="Project" date="2012-09-05">
+ <Cycle name="cruise.umple.compiler.sql et al" nodes="7" diameter="2"></Cycle>
+ <Metric id="NORM" description="Number of Overridden Methods"></Metric>
+ <Metric id="NOF" description="Number of Attributes"></Metric>
+ <Metric id="NSC" description="Number of Children"></Metric>
+ <Metric id="NOC" description="Number of Classes"></Metric>
+ <Metric id="MLOC" description="Method Lines of Code"></Metric>
+ <Metric id="NOM" description="Number of Methods"></Metric>
+ <Metric id="NBD" description="Nested Block Depth" max="5" hint="use Extract-method to split
+ <Metric id="DIT" description="Depth of Inheritance Tree"></Metric>
+ <Metric id="NOP" description="Number of Packages"></Metric>
+ <Metric id="CA" description="Afferent Coupling"></Metric>
+ <Metric id="NOI" description="Number of Interfaces"></Metric>
+ <Metric id="VG" description="McCabe Cyclomatic Complexity" max="10" hint="use Extract-me
+ <Metric id="TLOC" description="Total Lines of Code"></Metric>
+ <Metric id="RMI" description="Instability"></Metric>
+ <Metric id="PAR" description="Number of Parameters" max="5" hint="Move invoked method or
+ <Metric id="LCOM" description="Lack of Cohesion of Methods"></Metric>
+ <Metric id="CE" description="Efferent Coupling"></Metric>
+ <Metric id="NSM" description="Number of Static Methods"></Metric>
+ <Metric id="RMD" description="Normalized Distance"></Metric>
+ <Metric id="RMA" description="Abstractness"></Metric>
+ <Metric id="SIX" description="Specialization Index"></Metric>
+ <Metric id="WMC" description="Weighted methods per Class"></Metric>
+ <Metric id="NSF" description="Number of Static Attributes"></Metric>
</Metrics>

```

Figure 8. Static data extracted using Metrics 1.3.6.

We collect all of the data available using Metrics 1.3.6 (shown in Table 15), but in the end only four of the static data collected (McCabe/VG, MLOC, LOC and NBD, highlighted in the table) are used in this thesis.

We picked parameters which showed a strong positive or negative correlation with fan-in, which is the sole parameter used in the first iteration of this work. The tool continues to gather and process all of the parameters in Table 15 because we want to keep our option of parameters open for future work.

Static Data Collected	Description of Static Data Collected
Method Name	Name of the method
Class Name	Name of the class containing the method
Lines of Code (LOC)	Total lines of code in the selected scope. Only counts non-blank and non-comment lines inside method bodies.
Number of Classes (NOC)	Total number of classes in the selected scope
Number of Static Methods (NSM)	Total number of static methods in the selected scope.
Afferent Coupling (CA)	The number of classes outside a package that depend on classes inside the package.
Efferent Coupling (CE)	The number of classes inside a package that depend on classes inside the package.
Instability (RMI)	$CE / (CA + CE)$
Abstractness (RMA)	The number of abstract classes (and interfaces) divided by the total number of types in a package.
Normalized Distance (RMD)	$ RMA + RMI - 1 $, this number should be small, close to zero for good packaging design.
Depth of Inheritance Tree (DIT)	Distance from class Object in the inheritance hierarchy.
Number of Overridden Methods (NORM)	Total number of methods in the selected scope that are overridden from an ancestor class.
Number of Methods (NOM)	Total number of methods defined in the selected scope.
Specialization Index (SIX)	Average of the specialization index, defined as $NORM * DIT / NOM$. This is a class level metric
Number of Attributes (NOF)	Total number of attributes in the selected scope.
Number of Packages (NOP)	Total number of packages in the selected scope.

Static Data Collected	Description of Static Data Collected
Method Lines of Code (MLOC)	Total number of lines of code inside method bodies, excluding blank lines and comments.
Weighted Methods per Class (WMC)	Sum of the McCabe Cyclomatic Complexity for all methods in a class.
Number of Static Attributes (NSF)	Total number of static attributes in the selected scope.
Nested Block Depth (NBD)	The depth of nested blocks of code.
Lack of Cohesion of Methods (LCOM)	A measure for the Cohesiveness of a class. Calculated with the Henderson-Sellers method: If $m(A)$ is the number of methods accessing an attribute A, calculate the average of $m(A)$ for all attributes, subtract the number of methods m and divide the result by $(1-m)$. A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub)classes.
McCabe Cyclomatic Complexity (VG)	Counts the number of flows through a piece of code. Each time a branch occurs (if, for, while, do, case, catch and the ?: ternary operator, as well as the && and conditional logic operators in expressions) this metric is incremented by one. Calculated for methods only. For a full treatment of this metric see McCabe.
Number of Parameters (PAR)	Total number of parameters in the selected scope.
Number of Interfaces (NOI)	Total number of interfaces in the selected scope.

Table 15 Input parameter 2 – List of static data collected using Metrics 1.3.6.

6.2.3 CPU Input File

A third argument is a text file containing the CPU statistics. The file (labeled l3 in Figure 6) contains all the methods that are called during program execution and three types of CPU usage for each method, namely:

- Base Time, which is the time taken to execute the invocation of a method, excluding the time spent in other methods that were called during the invocation.
- Average Base Time, which is the base time divided by the number of calls.
- Cumulative CPU Time, which is the amount of CPU time spent in a method accumulated from all invocations.

```

- <View name="Execution Statistics - cruise.umple.UmpleConsoleMain at CORE-5F1KJN16EN [ PID: 6668 ]">
  <data Method="main(java.lang.String[]) void" Class="UmpleConsoleMain" Package="cruise.umple"
    Base_Time_seconds="0.129596" Average_Base_Time_seconds="0.129596"
    Cumulative_Time_seconds="28.789010" Calls="1"/>
  <data Method="optParse(java.lang.String[]) joptsimple.OptionSet" Class="UmpleConsoleMain"
    Package="cruise.umple" Base_Time_seconds="0.042201" Average_Base_Time_seconds="0.042201"
    Cumulative_Time_seconds="0.230578" Calls="1"/>
  <data Method="preModelOptionProcess(joptsimple.OptionSet) boolean" Class="UmpleConsoleMain"
    Package="cruise.umple" Base_Time_seconds="0.000084" Average_Base_Time_seconds="0.000084"
    Cumulative_Time_seconds="0.000128" Calls="1"/>
  <data Method="println(java.lang.String) void" Class="UmpleConsoleMain" Package="cruise.umple"
    Base_Time_seconds="0.002071" Average_Base_Time_seconds="0.001036"
    Cumulative_Time_seconds="0.002071" Calls="2"/>
  <data Method="postModelOptionProcess(joptsimple.OptionSet, cruise.umple.compiler.UmpleModel) boolean"
    Class="UmpleConsoleMain" Package="cruise.umple" Base_Time_seconds="0.000022"
    Average_Base_Time_seconds="0.000022" Cumulative_Time_seconds="0.000041" Calls="1"/>
  <data Method="OptionParser()" Class="OptionParser" Package="joptsimple" Base_Time_seconds="0.025250"
    Average_Base_Time_seconds="0.025250" Cumulative_Time_seconds="0.046592" Calls="1"/>
  <data Method="acceptsAll(java.util.Collection, java.lang.String) joptsimple.OptionSpecBuilder"
    Class="OptionParser" Package="joptsimple" Base_Time_seconds="0.027436"
    Average_Base_Time_seconds="0.004573" Cumulative_Time_seconds="0.050942" Calls="6"/>
  <data Method="recognize(joptsimple.AbstractOptionSpec) void" Class="OptionParser" Package="joptsimple"

```

Figure 9. CPU data extracted using TPTP.

The CPU data is collected using TPTP and saved as XML. Figure 9 shows a sample output from for the sample Umple trace. We wrote a program to automatically transform the XML file into a simpler delimited text file that is used as input into the TraceToUCM tool.

6.2.4 Percent Distribution Input File

The fourth argument (labeled l4 in Figure 6) to the tool is optional. This is a file containing filtering information on how the utility scores should be assigned and how much information should be included in the UCM diagram. If the file is not provided, internal defaults will be used. Figure 10 shows a sample filtering information file with a percentage distribution.

The first 10 lines of the file are used to set how the methods should be ranked (on a 5-rank scale) for each of the 10 algorithms. The first line of the file (“20 40 60 80 100 // Algorithm 1”) means that once the utilityhood is calculated using algorithm 1 and sorted, the first 20% of the methods will be ranked as 1, meaning least likely to be utilities; the next 20% ranks as 2, and so on. The most likely utility methods would then have the rank number 5. This enables the user to set the scale for each algorithm inde-

pendently as needed, and the scales do not have to be linear. In this way, a distribution can be fine-tuned, for example by changing it to use the first 15% instead of 20% for rank #1. The default value is (20 40 60 80 100).

```
20 40 60 80 100 // Algorithm 1
20 40 60 80 100 // Algorithm 2
20 40 60 80 100 // Algorithm 3
20 40 60 80 100 // Algorithm 4
15 40 60 80 100 // Algorithm 5
20 40 60 80 100 // Algorithm 6
20 40 60 80 100 // Algorithm 7
20 40 60 80 100 // Algorithm 8
20 40 60 80 100 // Algorithm 9 //currently not used.
20 40 60 80 100 // Algorithm 10 //currently not used.
3 // Algorithm 1
3 // Algorithm 2
3 // Algorithm 3
3 // Algorithm 4
4 // Algorithm 5
3 // Algorithm 6
3 // Algorithm 7
3 // Algorithm 8
3 // Algorithm 9 //currently not used.
4 // Algorithm 10 //currently not used.
30 -precise
```

Figure 10. Input parameter 4 – Filtering information.

The next 10 lines of the file have a single number between 1 and 5 and a comment to tell the user which algorithm the line belongs to. These lines specify the lowest ranked methods to include for each algorithm. Currently, the tool supports eight algorithms but can easily be extended to support more. In the example above, the ‘cut-off’ rank is the number 3 for Algorithm 1. That number indicates that the UCM map will only include methods that are ranked 1 to 3 (whereas likely utilities ranked 4 and 5 would be pruned out).

The last line of the file has a single number and an optional switch (`-precise`). It is used to specify the number of unique methods n (or responsibilities in the resulting filtered trace) that should be included in each UCM diagram. If the `-precise` switch is used, then exactly the top n methods that are least likely utilities will be used even if some of the entries after the n^{th} have the same value as the n^{th} . If `-precise` is not used then there could be more than n unique responsibilities in the filtered trace, since it will use the top n plus any others that have the same value at the n^{th} . The use of this switch often offers a less cluttered (at most n responsibilities) UCM diagram.

6.2.5 Exclusion Input File

The fifth argument of the tool (labeled l5 in Figure 6) is optional. It is a text file containing a list of items (packages, classes and/or methods, and any keywords) to be *excluded* from the filtered trace. Figure 11 shows an example of the optional exclusion input file.

```
cruise/umple/compiler/StateMachine.addNestedStateMachinesTo
cruise/umple/compiler/RuleInstance.updateNextPart
cruise/umple/util/StringFormatter.toUnderscore
cruise/umple/compiler/Parser.addRule
cruise/umple/compiler/RuleInstance.updateNextIdentifier
cruise/umple/compiler/RuleInstance/analyzeRule
cruise/umple/compiler/RuleInstance/configureDefinition
```

Figure 11. Optional input parameter 5 – List of components that must be excluded.

The information for the exclusion file can come from a designer that knows the system or someone trying to reverse-engineer a system in order to understand it. Provided with a suitable GUI, users could look at a large and messy diagram, click on an item to exclude, and repeat until they are happy with the resulting visualization. If the unwanted method occurs in multiple places, users can add the method to the exclusion list so that it is removed from subsequent runs. By the same token, users can also include items in the inclusion file (next section) to make sure they are never excluded.

6.2.6 Inclusion Input File

The sixth argument of the tool (labeled l6 in Figure 6) is also optional. It is a text file containing a list of items (packages, classes and/or methods, and any keywords) that have to be *included* in the UCM diagrams. These have priority over the items of the exclusion list. Figure 12 provides a simple example.

```
cruise/umple/compiler/UmpleInternalParser
```

Figure 12. Optional input parameter 6 – List of components that must be included.

The information for the inclusion file can come from a designer that knows the system or, just like in the case of the exclusion file, a user trying to reverse-engineer a system in order to understand it. An item in the inclusion file is never excluded.

6.3. Output Files

The tool generates several output files. The following are the output files and their descriptions.

1. One condensed execution trace file for each of the algorithms used. The condensed files are labeled CT1 to CTn in Figure 6.
2. A text file containing all the lines that have been excluded from the trace (because of the exclusion file). This is used to verify no other lines are pruned out. In Figure 6, the file is labeled o6.
3. An internal text file (labeled F1 in Figure 6) that contains trace items that are included after the items in the exclusion files are pruned out. This internal file is used as an input to other filtering steps.
4. A copy of the execution trace file excluding components in the exclude file and repeated loops (labeled F2 in Figure 6). This internal file is used as an input to other filtering steps.

5. A copy of the execution trace file excluding components in the exclude file, repeated loops and access methods (labeled F3 in Figure 6). It is used as an input to other filtering steps.
6. A copy of the execution trace file excluding components in the exclude file, repeated loops, access methods and small and simple methods (labeled F4 in Figure 6). If this file is less than 100 lines, a UCM diagram is generated directly from it. It is also used for subsequent filtering using the algorithms.
7. One UCM map for each of the algorithms in Chapter 5. In Figure 6, the UCM files are labeled U1 to Un.
8. A data table, a list of all methods in the execution trace along with the corresponding static information, CPU data, and their *utilityhood* score assigned by each algorithm. In Figure 6, the data table is labeled o1, and one example can be found in Section 9.1.4 (Table 40).
9. A list containing a column of method per algorithm, each containing the topmost non-utility methods according to each of the algorithms in Chapter 5. The first column is initially empty and is filled with a list provided from the human expert. This list is considered the canonical (i.e., presumed correct) list. We compare each algorithm (the data in each column) according to how good it is at picking out the methods in the list. For this, we use precision and recall metrics. In Figure 6, the data table is labeled o2.
10. A file containing a copy of the trace file as it is segmented for the segmentation presence algorithm. In Figure 6 the data table is labeled o3.
11. A file that is used to record the length of the trace file at each step of the filtering process. From this we can calculate the percentage of reduction of the traces at each step. In Figure 6 the data table is labeled o4.
12. One interesting artifacts that is not used at this time is a correlation report calculated using Dugerdil's clustering algorithm [30]. In Figure 6 the file containing clustering information is labeled o5. Methods that tend to appear together in cluster are listed together. Figure 13 shows an example of Correlation report for jUCMNav. We can see the method `seg/jUCMNav/JUCMNavPlugin/start` is corre-

lated to two methods below and the strongest correlation is with

seg/jUCMNav/JUCMNavPlugin/getDefault

- a. seg/jUCMNav/model/ModelCreationFactory/getNewURNspec
- b. seg/jUCMNav/JUCMNavPlugin/getDefault

```
For method: seg/jUCMNav/JUCMNavPlugin/start
Related to:
-->      seg/jUCMNav/model/ModelCreationFactory/getNewURNspec
-->      seg/jUCMNav/JUCMNavPlugin/getDefault

Strongest correlation to:
-->      seg/jUCMNav/JUCMNavPlugin/getDefault

*****
For method: seg/jUCMNav/model/ModelCreationFactory/getNewURNspec
Related to:
-->      seg/jUCMNav/JUCMNavPlugin/start
-->      seg/jUCMNav/JUCMNavPlugin/getDefault
-->      seg/jUCMNav/Messages/getString
-->      seg/jUCMNav/model/ModelCreationFactory/getNewObject
-->      seg/jUCMNav/editors/UCMNavMultiPageEditor/setModel
-->      seg/jUCMNav/editors/UCMNavMultiPageEditor/setTarget
-->
seg/jUCMNav/editors/actionContributors/UCMActionBarContributor/buildActions
-->
seg/jUCMNav/editors/actionContributors/UCMActionBarContributor/declareGlobal
ActionKeys
-->
seg/jUCMNav/editors/actionContributors/UCMActionBarContributor/contributeToT
oolBar
-->      seg/jUCMNav/actions/AutoLayoutActionDelegate/setActiveEditor
-->      seg/jUCMNav/actions/AutoLayoutActionDelegate/selectionChanged
-->      seg/jUCMNav/actions/SwitchModelLanguageDelegate/setActiveEditor
-->      seg/jUCMNav/actions/SwitchModelLanguageDelegate/selectionChanged
-->      seg/jUCMNav/actions/AdvancedModeActionDelegate/setActiveEditor
-->      seg/jUCMNav/actions/AdvancedModeActionDelegate/selectionChanged
-->      seg/jUCMNav/actions/UCMActionDelegate/setActiveEditor
-->      seg/jUCMNav/actions/UCMActionDelegate/selectionChanged
-->
seg/jUCMNav/actions/scenarios/VerifySyntaxActionDelegate/setActiveEditor
-->
seg/jUCMNav/actions/scenarios/VerifySyntaxActionDelegate/selectionChanged
-->      seg/jUCMNav/editors/UCMNavMultiPageEditor/selectionChanged
-->
seg/jUCMNav/editors/actionContributors/ActionRegistryManager/updateEditPartA
ctions
-->
seg/jUCMNav/editors/actionContributors/ActionRegistryManager/updateActions
-->      seg/jUCMNav/editors/MultiPageTabManager/selectionChanged

Strongest correlation to:
-->      seg/jUCMNav/editors/MultiPageTabManager/selectionChanged
```

Figure 13. Clustering report – List of methods that tend to occur together.

This report is not currently used as clustering is outside of the scope of this thesis. However, it provides interesting insight into relationships of methods in the trace and users might like to review it.

6.4. Final Product of the Tool

6.4.1 Sample Condensed Trace File

At the end of the TraceToUCM transformation, the user gets several condensed versions of the input execution trace file described in Section 6.2.1. The condensed files are labeled CT1 to CTn Figure 6. Figure 14 shows a sample condensed trace file described in Section 6.3, item number 1.

```
ttu/TraceToUCM/main 0
ttu/TraceToUCM/PrepareOutputFileNames 1
ttu/TraceToUCM/IncludeStaticdataToMethodsInTrace 1
ttu/TraceToUCM/CalculateFaninFanoutAndCallHierarchy 1
ttu/TraceSegmentationAndClustering/calculateNumberOfSegmentsUsingInputFileSize 1
ttu/TraceSegmentationAndClustering/FindMethodsInSegmentsofFile 1
ttu/TraceSegmentationAndClustering/CalculateCorrelation 1
ttu/TraceSegmentationAndClustering/printCorrelationReport 2
ttu/TraceToUCM/CountMarkAndRemoveLoops 1
ttu/TraceToUCM/ScanForLoops 2
ttu/TraceToUCM/RemoveExitMarkersFromTraceFile 1
ttu/TraceToUCM/MarkAndFilterOutAccessMethods 1
ttu/TraceToUCM/CalculateAverageDistance 1
ttu/TraceToUCM/CalculateAverageDistanceBetweenMethodsCalled 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodCSMR2005 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodStaticFaninFanout 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodMcCabe 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodMLOC 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodSegmentationPresence 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodYuying 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodAverageCallTreeDepth 1
ttu/CalculateUtilityhoodVariants/CalculateUtilityhoodCPU 1
ttu/TraceToUCM/IncludeCPUInfo 1
ttu/TraceToUCM/FilterOutSmallSimpleMethods 1
ttu/TraceToUCM/writeListofNonUtilityMethodsToFile
```

Figure 14. Output artefact 1 – A condensed trace file.

6.4.2 Sample Use Case Map

The behavioural model of the condensed trace file in Figure 14 is visualized in Figure 15. The UCM components represent the structure of the packages and the UCM responsibilities the sequence of methods invoked. Loops can also be visualized.

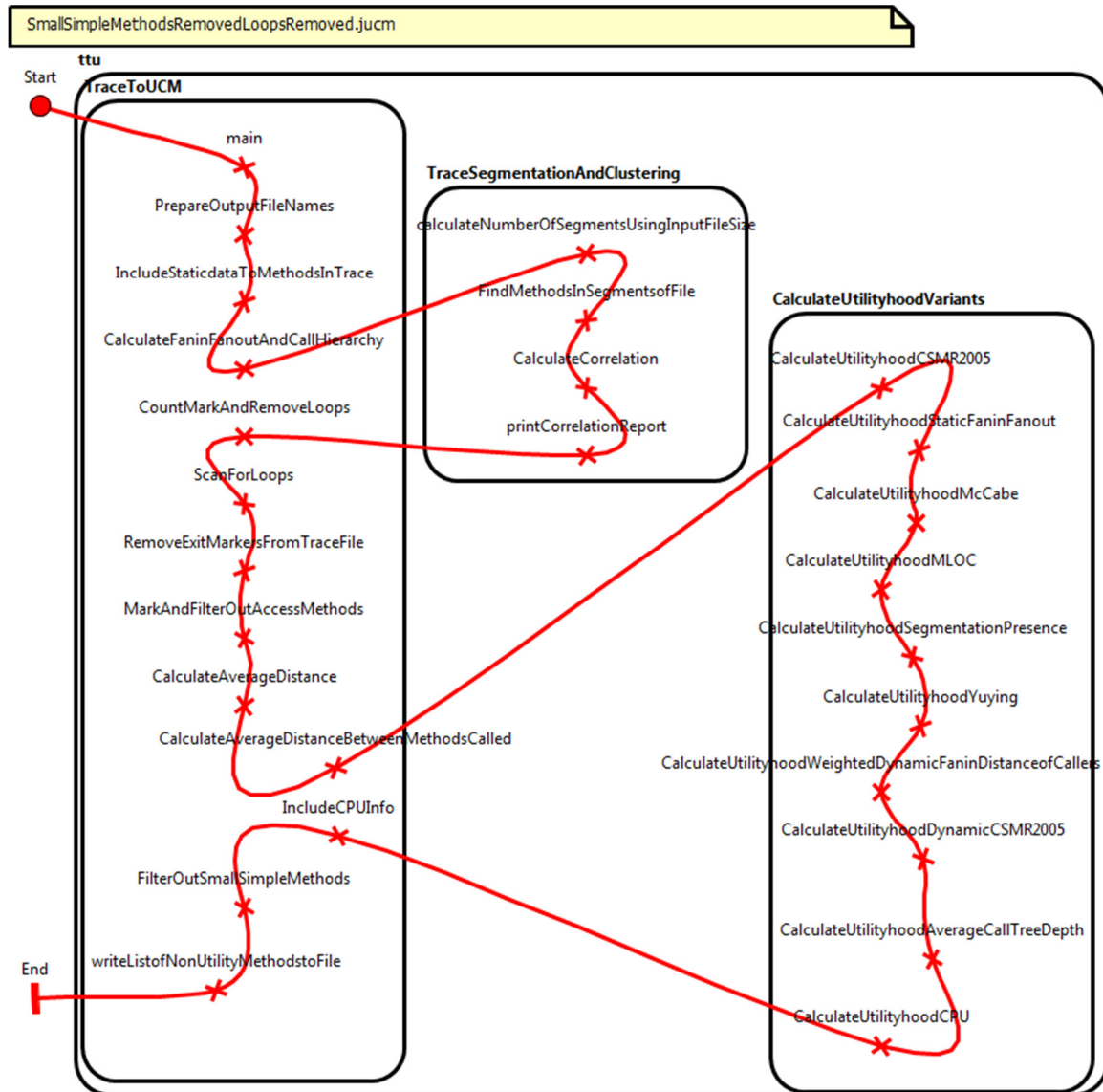


Figure 15. Use case map of the TraceToUCM feature.

When Use Case Maps become too large and complex, we make use of UCM stubs to manage this complexity through hierarchical decomposition. For example, the tool transforms the UCM shown in Figure 16 to the UCM shown in Figure 17. The tool has a

switch to enable the user to ask two copies of the UCM, one with stubs and one without. The tool generates a UCM diagram automatically. This feature is the key feature that makes our methodology scalable and addresses the complexity of visualizing large traces..

Table 11 shows the mapping of trace components to UCM notations. If there are x instructions in the same class, we create a stub and draw the x responsibilities in a plug-in (a sub-map). Currently, the stub's plug-ins do not include components. There are many improvements that can be done when creating stubs in order to increase abstraction of larger and complex traces than we have handled in this work. Future work (Section 10.3) discusses some of the ideas related to how this can be expanded.

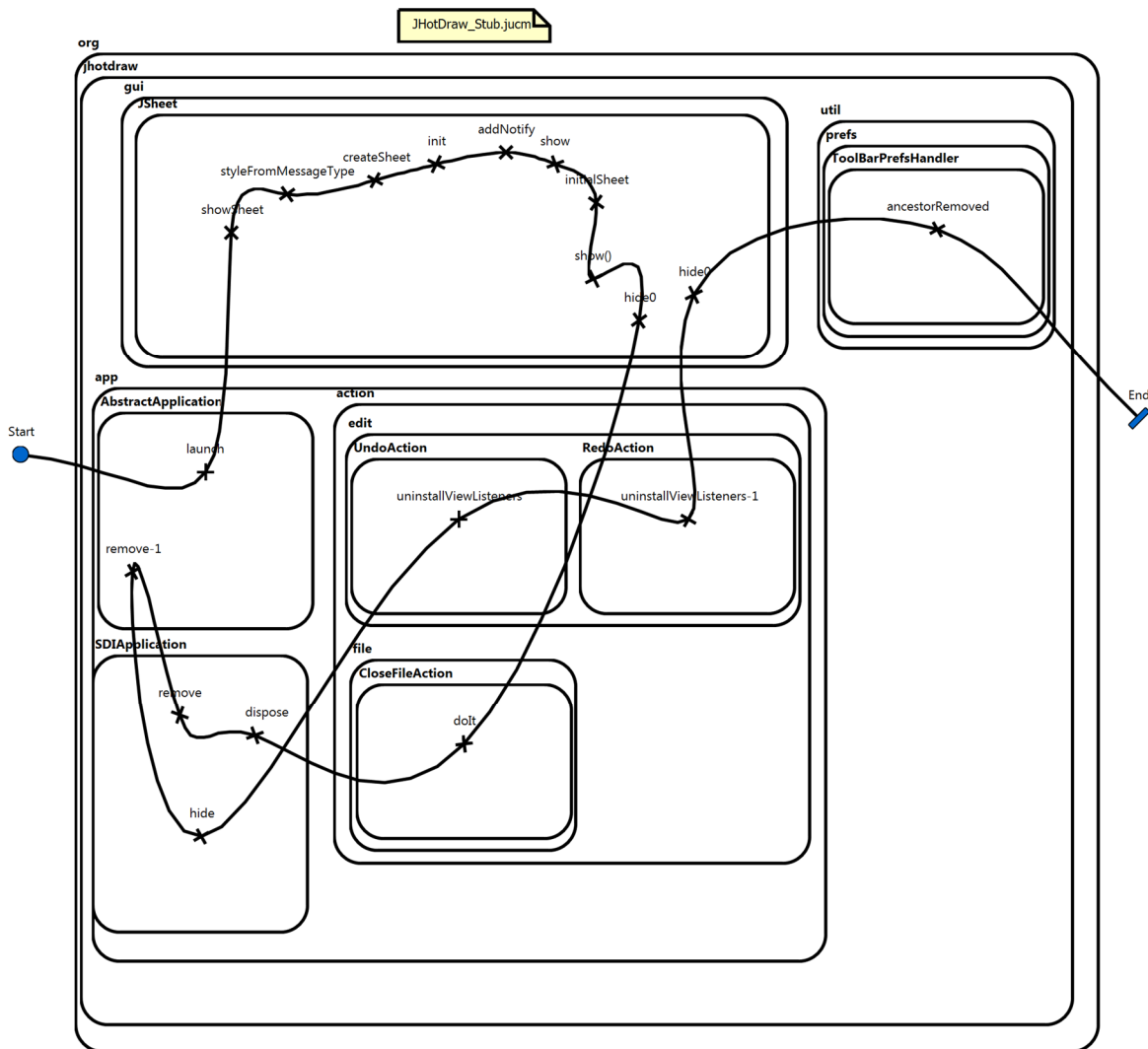


Figure 16. A UCM diagram without stubs.

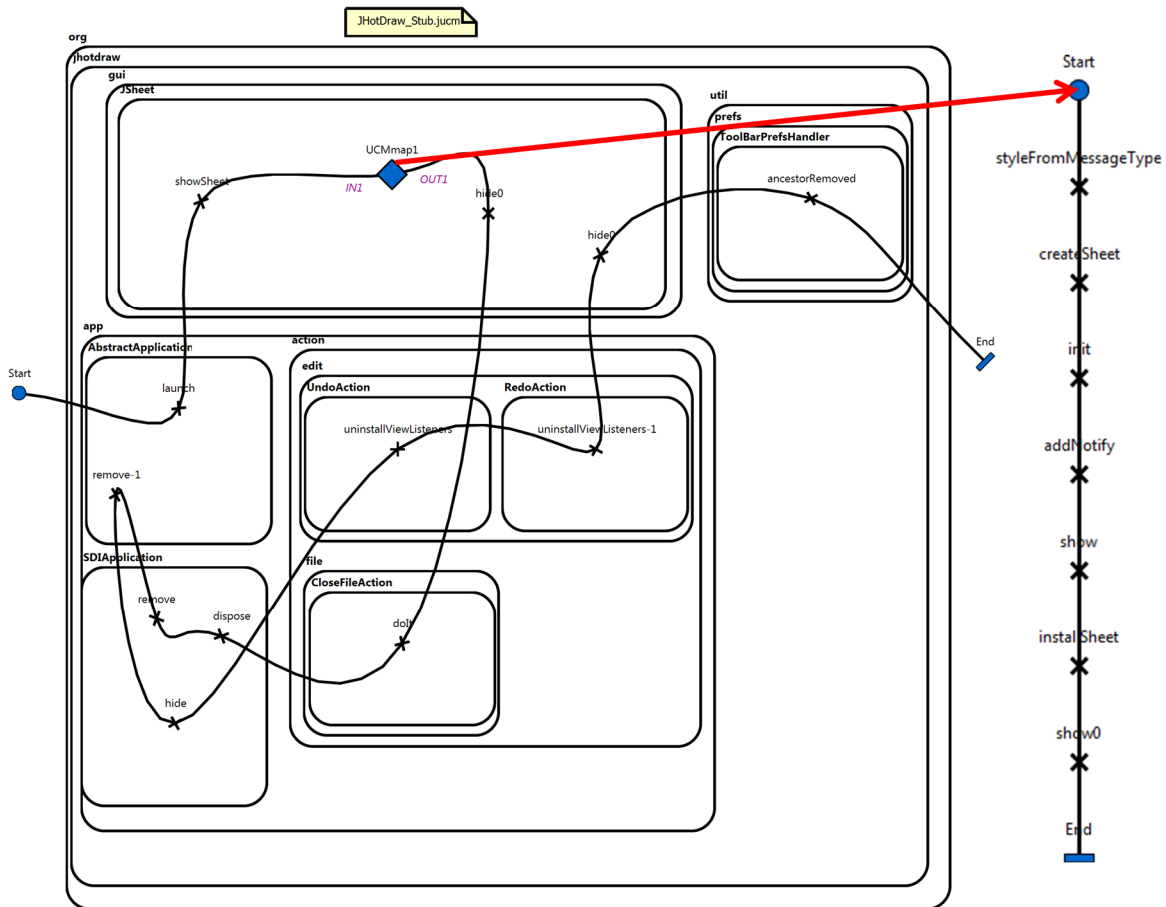


Figure 17. A UCM diagram with stub (left), with its plugin UCM (right).

6.4.3 Message Sequence Chart

Message Sequence Chart (MSC) is an ITU language [58] now incorporated in UML 2's sequence diagrams. It is used to describe a sequence of interactions between a set of component, as shown at the top of Figure 18. Each vertical line represents a component (class in this context) and is called an instance. Each horizontal arrow represents a synchronous message, capturing method invocations. A user who is more comfortable reading MSCs or UML sequence diagrams can generate a MSC using a pre-existing jUCMNav feature that enables a user to transform UCMs to MSCs. To enable this, TraceToUCM also generated, for each UCM file, a scenario definition that can be executed for scenario highlighting (as for the red path in Figure 15) and for MSC generation.

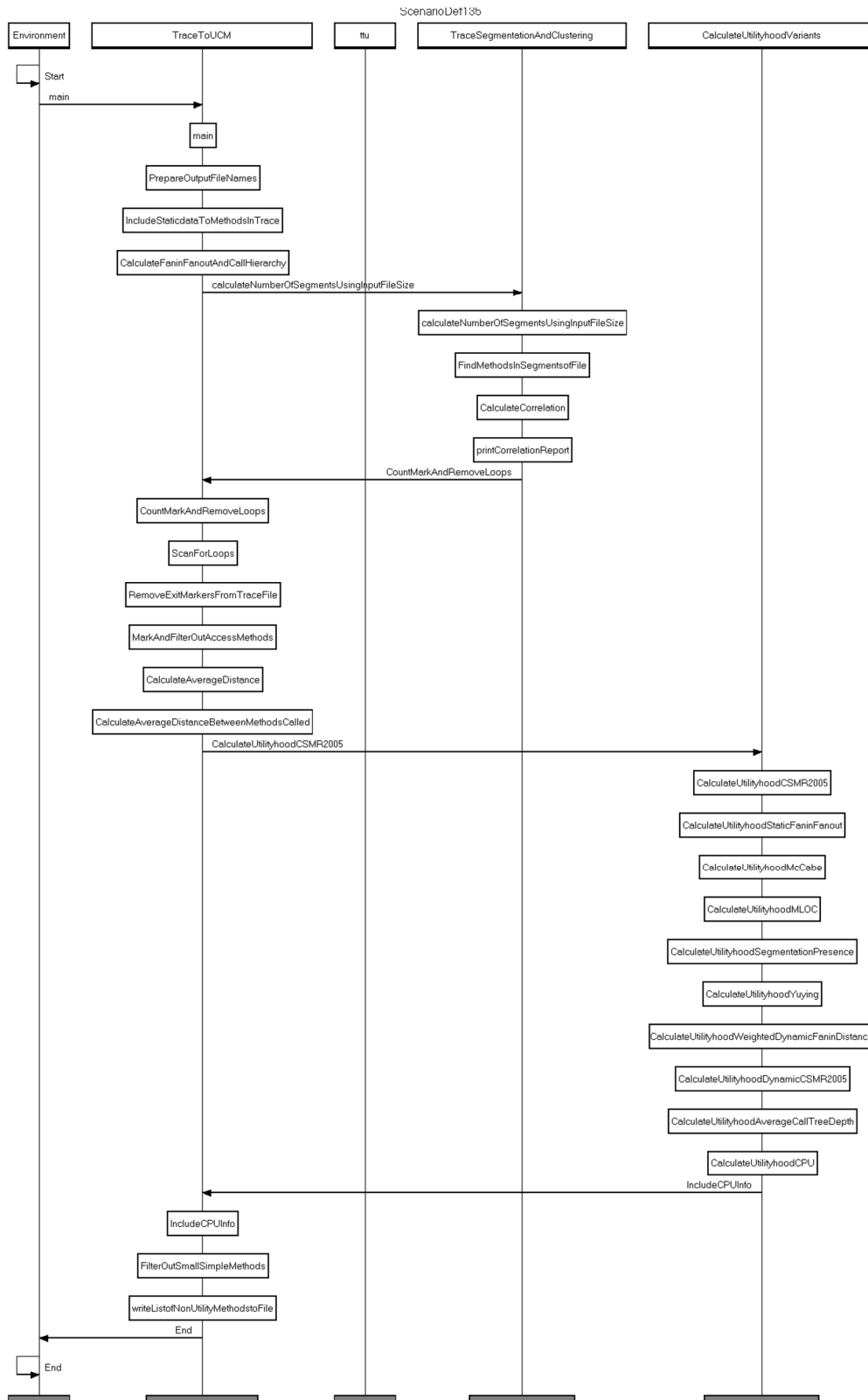


Figure 18. Message Sequence Diagram of the TraceToUCM feature.

For example, the condensed trace in Figure 14 is first transformed to Figure 15 using TraceToUCM. In order to generate the MSC diagram in Figure 18, the user selects the generated scenario definition and then exports that scenario to the MSC format, which can also be visualized with jUCMNav (and further exported to various graphical formats). Miga *et al.* [83] give a detailed description of the procedure used to generate Message Sequence Charts from Use Case Maps, which was further refined by Kealey [63].

6.5. Summary

This chapter gives a description of new suite of tools, with the new TraceToUCM as the main tool, used to implement the methodology described in this thesis. The tools are used to analyze static and dynamic parameters of object-oriented software, filter execution traces using algorithms that are described in Chapter 5, and finally visualizing the traces using UCM diagrams (and even MSC diagrams). In the process of transforming execution trace to UCM diagrams, several artefacts are produced. We reviewed the main input, internal, and output files. Finally we looked at an example of a trace file used to produce a UCM diagram and then an MSC. In the next two chapters, we look at case studies that were done to evaluate the use of reduced execution traces, visualized as UCM diagrams.

Chapter 7. Reverse Engineering Case Studies

In order to validate the effectiveness of the methodology we have developed in this thesis, and to evaluate the various filtering algorithms in practice, we used our tool-supported approach on three software systems, namely jUCMNav, UCed and Umple. We reverse engineered these systems using our approach and asked their developers to evaluate the results. These software systems were selected because they are all mature open-source production systems and also because of the ease of access to their creators and maintainers. In this chapter, we present the details of these case studies and analyse their empirical results.

7.1. Participants

The participants for the overall study are the creators, designers and maintainers of the three software programs studied. The recruitment letter and questionnaire that were used for the study were sent to potential participants via email, and they are included in Appendix A.

The recruitment e-mail was sent to fifteen people. Eight people volunteered to participate. We had three participants for jUCMNav, four participants for Umple and two participants for UCed. One of the participants is counted twice because he worked on two of these systems. All participants are familiar with the UCM notation, with jUCMNav participants being experts.

7.2. Empirical Study

For each of the three programs studied, we first select one feature to reverse engineer. We compile the system under study in order to collect static data. We exercise the feature, collect execution traces, and apply the algorithms discussed in Chapter 5 to those execution traces in order to detect and filter out utilities. The systems are run again so that CPU

profiles get collected. Using TraceToUCM, we then iteratively filter out the utility methods from the trace and thereby reduce the total size of the execution trace. We use the resulting, significantly-smaller execution traces from each of the algorithms to draw behavioural models. UCM diagrams are generated automatically for each such trace (one per algorithm), using the mapping shown in Table 11. We then show the resulting UCM diagrams to the software developers to validate whether the algorithms have identified the correct methods as the principal methods, and also to determine whether the resulting diagrams capture the behaviour of the feature.

For each of the three software systems, the questionnaires were modified to include eight system-specific UCM diagrams, one for each of the eight algorithms discussed in Chapter 5 (and summarized Table 13). The three systems studied are introduced in Section 4.1.1. Results of our evaluation of each system are explained in the following three sections.

7.3. Case Study 1 – jUCMNav

jUCMNav is a Java-based, open source Eclipse plug-in primarily developed at the University of Ottawa [87]. It is a graphical modeling and analysis tool for the User Requirements Notation (URN), which includes the Use Case Map (UCM) notation and the Goal-oriented Requirement Language (GRL) [62]. jUCMNav is the largest software system studied in this thesis.

In this case study, we sent questionnaires to four jUCMNav designers. Three of the designers filled out the questionnaire. Prior to filling the questionnaire, we asked one of the main designers to give us a list of the principal methods that are used for the feature under test to be used for algorithm verification.

7.3.1 Description of Feature Selected

We selected and ran the jUCMNav application feature “Open a UCM diagram”, using a simple UCM model for a pizzeria business process shown in Figure 19. The feature uses a basic pattern of functionality in jUCMNav: draw components, tasks, paths, then create a UCM map and terminate the UCM map. We believe it gives us a good behavioural model of the selected feature because the diagram contains the major UCM modeling elements.

Figure 20 shows one of the behavioural models of jUCMNav generated. The rest of the diagrams are listed in Appendix B.

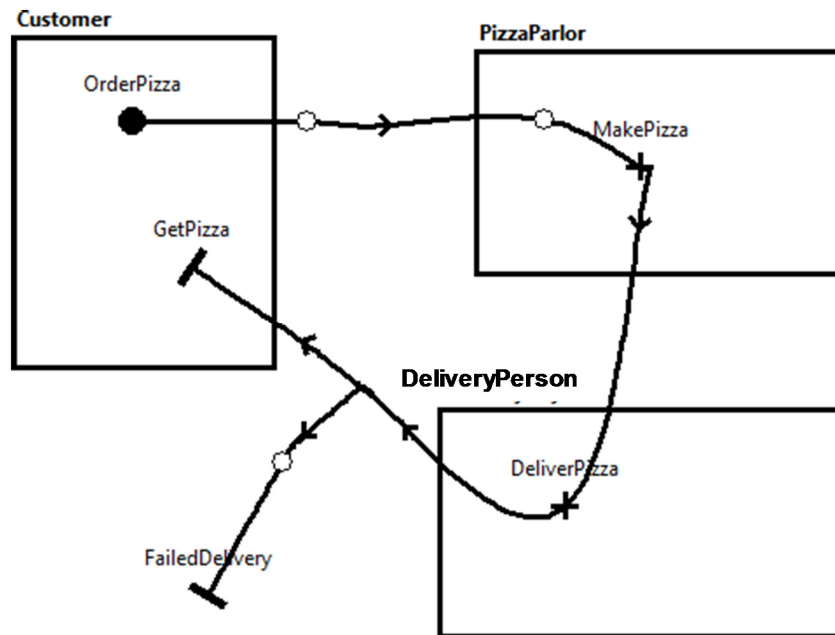


Figure 19. Use case map created to exercise the “Open a UCM diagram” feature.

7.3.2 jUCMNav Traces

In this section, we look at the traces that are produced by the TraceToUCM tool. Table 16 gives a summary of the jUCMNav trace files. We can see that even before the algorithms are applied to the trace, just by removing repeated loops, exit markers, access methods and methods that are small and simple, we have reduced the trace of method calls from a length of 31,459 lines to 2391 lines, which represents a 92.4% reduction in size.

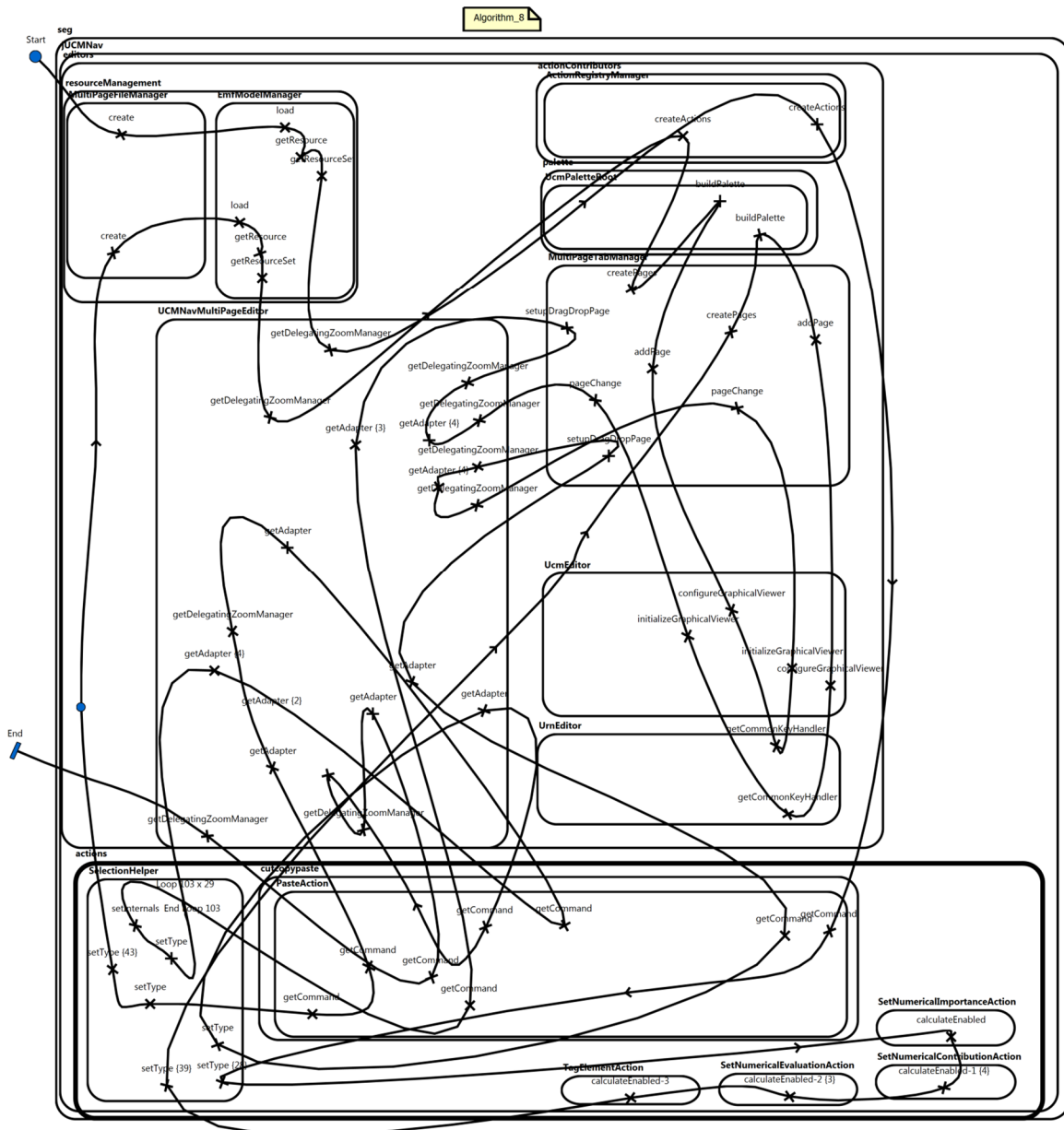


Figure 20. Behavioural map of jUCMNav using Algorithm 8.

File name	File size	Cumulative reduction
jucmnav_Original	31459	0.00%
jucmnav_Original_LoopsRemoved	12411	60.55%
jucmnav_ExitMarkerRemoved	6623	78.95%
jucmnav_AccessMethodsRemoved	2743	91.28%
jucmnav_Original_SmallSimpleMethodsRemoved	2391	92.40%
jucmnav_Algorithm_1	354	98.87%
jucmnav_Algorithm_2	88	99.72%
jucmnav_Algorithm_3	251	99.20%
jucmnav_Algorithm_4	426	98.65%
jucmnav_Algorithm_5	49	99.84%
jucmnav_Algorithm_6	107	99.66%
jucmnav_Algorithm_7	52	99.83%
jucmnav_Algorithm_8	130	99.59%

Table 16 jUCMNav execution trace line counts.

To assess which of the algorithms best depicts the behavioural model of the tool, we gathered the answers given by the three jUCMNav designers (labelled jS1 to jS3) to the question “Does Algorithm X’s diagram contain the main methods used in jUCMNav’s code for the ‘Open and load a UCM file’ feature?”. Table 17 shows the answers to that question.

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
jS1	-2	-2	-2	-2	-2	-2	-2	1
jS2	-1	1	-1	1	0	-1	0	-1
jS3	1	1	1	0	-2	1	-1	1
Summary	-2	0	-2	-1	-4	-2	-3	1

Legend	
Strongly disagree	-2
disagree	-1
neutral	0
agree	1
strongly agree	2

Table 17 Summary of responses from jUCMNav designers (algorithms’ selection of methods).

There is no clear consensus among the three designers; however, we can see that Algorithm 8 is the most favoured by the group (relative to the other) whereas Algorithm 5 is

the least favoured. The legend in Table 17 also applies to Table 18. Given that statistical significance cannot be achieved with so few answers, a “summary” line is provided that sums up the scores/answers provided by the designers.

Another question posed to the designers was: “Does the UCM diagram match the mental image you have of the jUCMNav ‘Open a UCM file’ feature?”. The answers, summarized in Table 18, suggest that most diagrams do not match their mental models, with Algorithm 5 again being the worst, and Algorithm 2 being the best this time.

Note that, at times, even when a designer thinks the algorithm has selected the correct methods, he/she still says the map does not match his/her mental image. For example, jS3’s answer for Algorithm 1 (Table 17) gives the algorithm a score of 1 for picking the methods, but -1 for the diagram (Table 18). This might be caused by the layout of the diagram.

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
jS1	-2	-2	-2	-2	-2	-2	-2	-2
jS2	0	1	0	1	0	0	0	0
jS3	-1	1	0	0	-2	1	0	1
Summary	-3	0	-2	-1	-4	-1	-2	-1

Table 18 Summary of responses from jUCMNav designers (UCM diagram comparison).

7.4. Case Study 2 – Umple

Umple is a modeling tool and programming language family that enables Model-Oriented Programming. The Umple compiler itself is written in Umple [11][12][41][74]. We used Umple 1.15.0.1751, which has 44 packages, 430 classes, 4457 methods and 79,459 lines of code.

Figure 21 shows a behavioural model of Umple (for a selected program) generated using Algorithm 3. The rest of the diagrams can be found in Appendix C.


```

/* BcmsCore.ump - Core Umple model file for the Crisis management system
 * Source document: http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf
 * Author: Timothy C. Lethbridge
 * To use this, a master file must be created with use statements that link in RoutesAndLocations.ump and
 * CoordinationStateMachine.ump Plus provide * multiplicities as described. */
/* The singleton that will link to the various system elements
 * Note that variants with different multiplicities will associate this with crises and coordinators */
class CrisisManagementSystem
{ // There is a single crisis management system
  singleton; }
/* A crisis is what is being managed by the Crisis Management System */
class Crisis
{
  identifier;
  Location gpsLocation;
  Time initialReportedTime;
  status {active {} closed {}}
  description;
  0..1 -> * Vehicle assignedFireVehicles;
  0..1 -> * Vehicle assignedPoliceVehicles;
  0..1 -> 0..1 Route proposedFireRoute;
  0..1 -> 0..1 Route proposedPoliceRoute;
  0..1 -> * Route disagreedRoutes;
  0..1 -> * RoadSegment
    blockedOrCongestedSegments;
}
// Police or fire vehicle from Requirements section 8 Data Dictionary
class Vehicle
{
  vehicleID;
  Location currentLocation;
  Time eta;
  Location intendedLocation;
  locationStatus {
    station {}
    enRouteToCrisis {}
    atCrisis {}
    enrouteReturn {}
  }
  0..1 -- 0..1 Route routePlan;
  Boolean brokenDown;
  Boolean inTrafficCongestion;
}
/*
 * The station coordinator is the automated agent of the Human coordinator in either the fire service (FSC)
 * or police service (PSC). Any running version of this program will just have one of these active as the current
 * coordinator and one or more active as remote coordinators to communicate with*/
class StationCoordinator
{
  identification;
  Boolean police; // false for fire
}
// Classes defined elsewhere
class RoadSegment {}
class Route {} // ${End_of_model}$?

```

Figure 22. Program used to run Umple.

7.4.2 Umple Traces

In this section, we study the Umple execution trace collected while compiling the program in Figure 22 with the Umple compiler. We processed the traces using the methodology described in this thesis to extract the behavioural model of the selected feature. Table 19 gives a summary of the trace files generated using our tool suite. We can see that even before the algorithms are applied to the trace, just by removing repeated loops, exit markers, access methods, and methods that are small and simple, we have reduced the execution trace from 2,023,349 lines to 14,183 lines, which represents a 99.3% reduction in size.

File name	File size	Cumulative reduction
Umple_Original	2023349	0.00%
Umple_Original_LoopsRemoved	701019	65.35%
Umple_ExitMarkerRemoved	363687	82.03%
Umple_AccessMethodsRemoved	80731	96.01%
Umple_Original_SmallSimpleMethodsRemoved	14183	99.30%
Umple_Algorithm_1	4331	99.79%
Umple_Algorithm_2	123	99.994%
Umple_Algorithm_3	245	99.990%
Umple_Algorithm_4	1765	99.91%
Umple_Algorithm_5	607	99.97%
Umple_Algorithm_6	172	99.99%
Umple_Algorithm_7	75	99.996%
Umple_Algorithm_8	577	99.97%

Table 19 Umple execution trace line counts.

To assess which of the algorithms best show the “compile” feature of the Umple program, we gathered the answers given by four of the Umple designers (labelled uS1 to uS4) to the question “Does the Algorithm X’s diagram contain the main methods used for the Umple compile feature?” Table 20 shows the answers to that question.

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
uS1	1	-1	0	0	0	0	-1	-1
uS2	1	1	1	1	1	1	1	0
uS3	-1	-1	1	0	-1	-1	0	0
uS4	-2	1	2	-1	-2	0	-2	-2

Summary	-1	0	2	1	-2	0	-2	-3
---------	----	---	---	---	----	---	----	----

Legend	
Strongly disagree	-2
disagree	-1
neutral	0
agree	1
strongly agree	2

Table 20 Summary of responses from Umple designers (algorithms' selection of methods)

Similarly, Table 21 shows the answers to the question regarding which diagram matches their mental image. The legend in Table 20 applies also to Table 21. There is no clear consensus among the four designers; however, we can see that Algorithm 3 is the most favoured by the group (without anyone disliking it), and Algorithm 8 is the least favoured. Algorithm 3 was favored both for choosing the correct “main methods” and for being closer to the designers' mental image of the feature.

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
uS1	1	-1	0	0	0	0	-1	-1
uS2	1	1	2	2	1	2	1	1
uS3	-1	-1	2	-1	-1	-1	-1	0
uS4	-2	1	2	-1	-2	0	-2	-2

Summary	-1	0	4	1	-2	1	-3	-2
---------	----	---	---	---	----	---	----	----

Table 21 Summary of responses for Umple designers (UCM diagram comparison).

7.5. Case Study 3 – UCed (Use Case Editor)

UCed [111] is an environment for use-case-based requirements engineering. It has tools for editing use cases, such as use-case integration and use-case simulation. It is also developed at the University of Ottawa [102]. We used UCed 1.6.2, which has 700 classes, 3284 methods, and 33,004 lines of code.

7.5.1 Description of Feature Selected

We selected and run the UCed application's "Open a project" feature with the UCed project shown in Figure 23. We chose a project used on the UCed webpage [111] to illustrate the use of the editor.

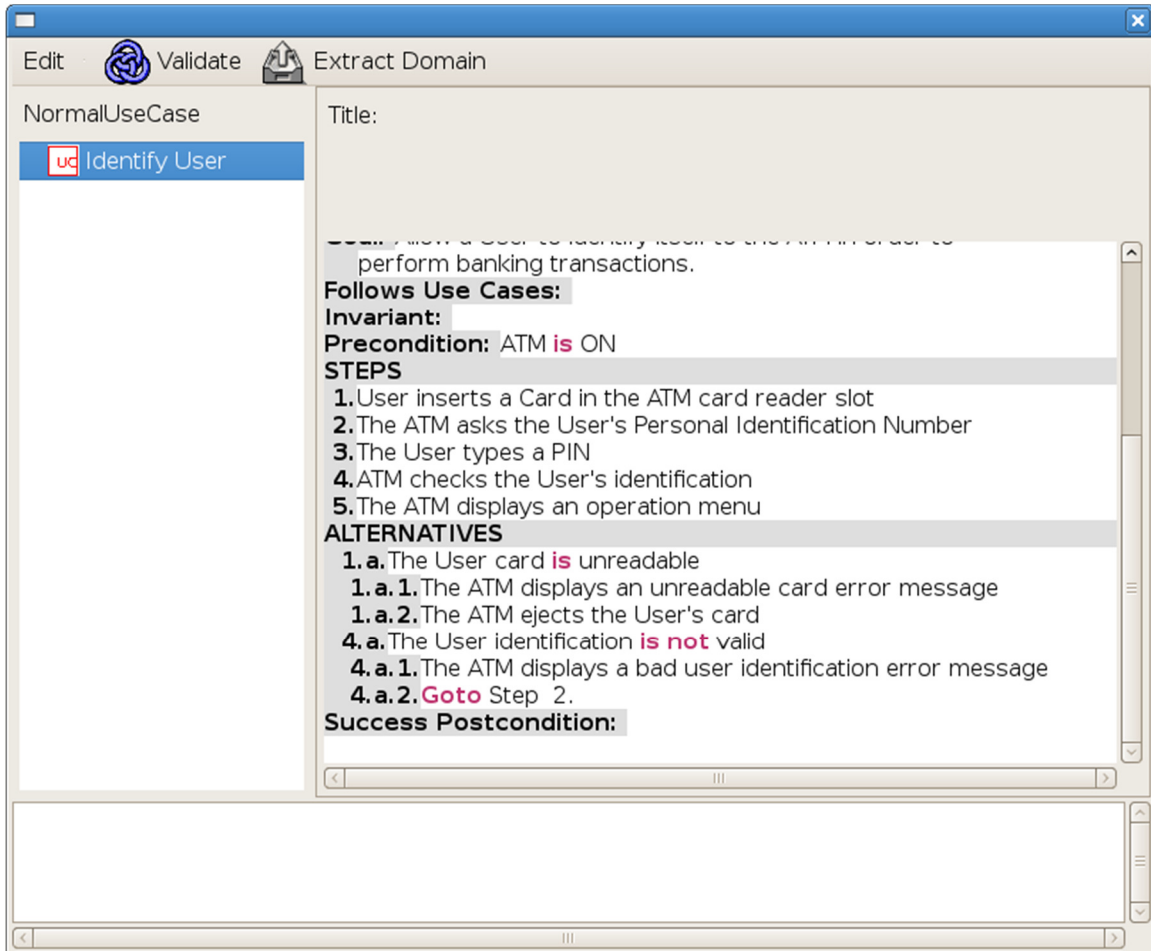


Figure 23. Use case example opened using UCed feature [111].

7.5.2 UCed Traces

In this section, we look at the UCed execution traces that are processed using our methodology. Table 22 gives a summary of the trace files. We can see that even before the algorithms are applied to the trace, just by removing repeated loops, exit markers, access methods and small and simple methods, we have reduced the execution trace from 19,543 to 1987 lines, which represents an 89.9% reduction in size.

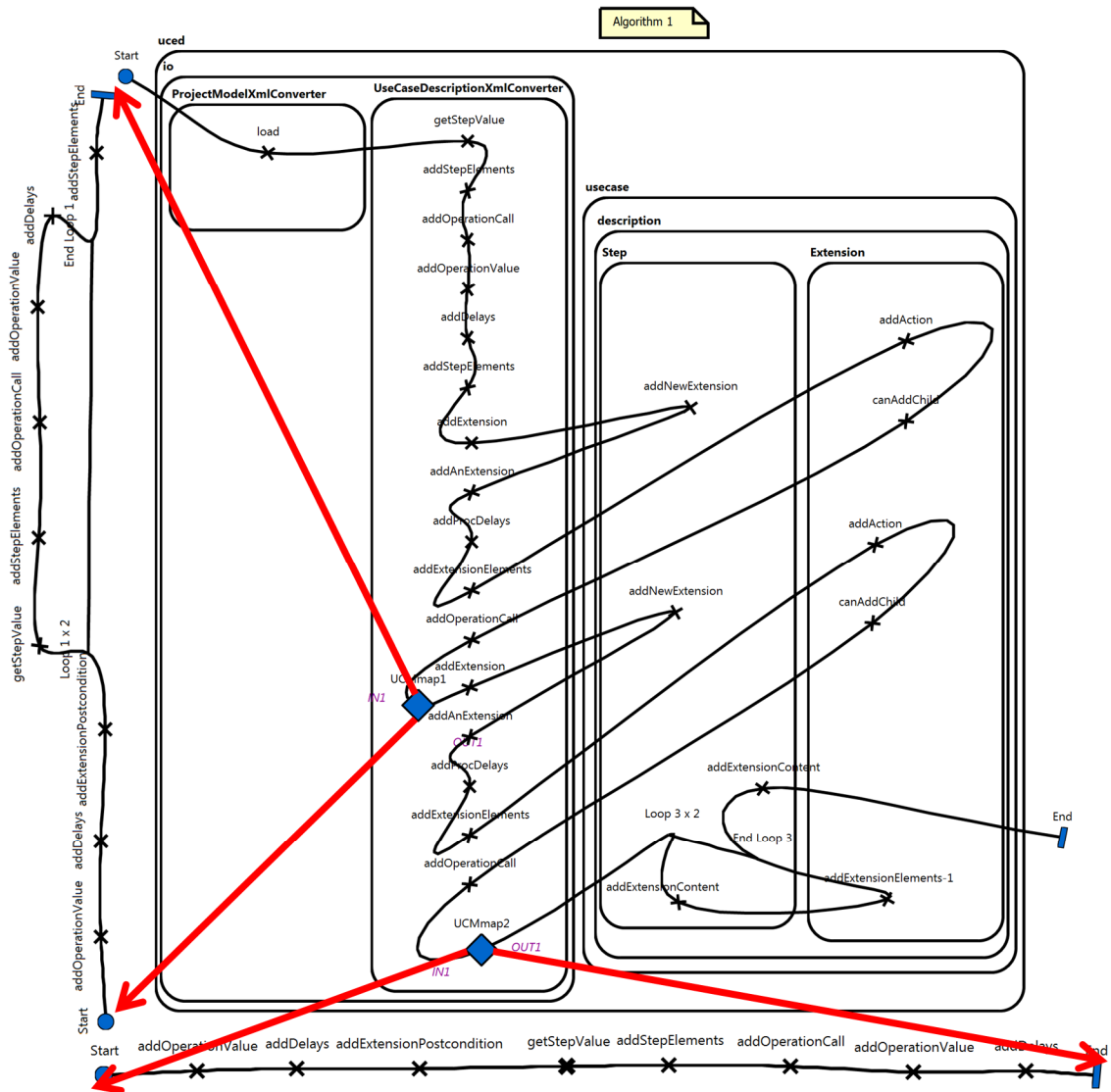


Figure 24. Behavioural map of UCed using Algorithm 1.

Figure 24 shows the behavioural UCM model of UCed generated using Algorithm 1. The rest of the UCed diagrams can be found in Appendix D.

File name	File size	Cumulative reduction
Uced_Original	19543	0.00%
Uced_Original_LoopsRemoved	7698	60.61%
Uced_ExitMarkerRemoved	4164	78.69%
Uced_AccessMethodsRemoved	3088	84.20%
Uced_Original_SmallSimpleMethodsRemoved	1987	89.83%
Uced_Algorithm_1	72	99.63%
Uced_Algorithm_2	73	99.63%
Uced_Algorithm_3	53	99.73%
Uced_Algorithm_4	164	99.16%
Uced_Algorithm_5	29	99.85%
Uced_Algorithm_6	20	99.90%
Uced_Algorithm_7	61	99.69%
Uced_Algorithm_8	99	99.49%

Table 22 UCEd execution trace line count.

To assess which of the algorithms best shows the main methods for the “Open a project” feature, we recruited two of the main UCEd designers (labelled uCS1 and uCS5) to fill out a questionnaire. Table 23 shows the answer to the question “Does the Algorithm X’s diagram contain the main methods used for the “open a project” feature in UCEd code?” The legend in Table 23 applies also to Table 24.

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
uCS1	2	1	-1	2	-2	1	0	-2
uCS5	1	1	0	1	-1	-2	1	-1
Summary	3	2	-1	3	-3	-1	1	-3

Legend	
Strongly disagree	-2
disagree	-1
neutral	0
agree	1
strongly agree	2

Table 23 Summary of responses from UCEd designers (for each algorithm’s selection of methods).

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
uCS1	2	1	-1	2	-2	1	0	-2
uCS5	1	-1	-1	0	-2	-2	-1	-1
Summary	3	0	-2	2	-4	-1	-1	-3

Table 24 Summary of responses from UCed designers (UCM diagram comparison).

There is some consensus among the two designers. Algorithm 1 and 4 are the most favoured and Algorithm 5 & 8 are the least favoured.

7.6. Validity of UCM Diagrams for Behavioral Model

To assess the validity of using a UCM diagram to show behavioural maps, all participants were asked the following question: “Assuming the correct methods had been selected, would you use UCM diagrams to document the behaviour of a feature in your design documents?”

In this case, seven out of the eight participants answered that they agreed or strongly agreed, and none disagreed. Even though we had nine participants we only counted eight because one of the participants worked both on Umple and UCed.

Participants	Part II
uS1	Agree
uS2	Agree
uS3	Agree
uS4	Neutral
jS1	Agree
jS2	Agree
jS3	Agree
uCS1	Neutral
uCS5	Strongly agree

Table 25 Participants’ answers to part II of the questionnaire.

Table 25 shows the participants' response to part II of the questionnaire. The aim of the question in part II is to validate the approach of using UCM to visualize traces, independent of the method selection algorithms. The participants in general agree that UCM diagrams can in fact be used to visualize traces. We discuss a few ideas to improve trace visualisation in Section 10.3.

7.7. Algorithm Comparison Process

7.7.1 Precision and Recall

In order to compare the performance of the eight algorithms independent of the visualization technique used, we asked one designer from each of the three systems to give us a list of the main methods for the selected features, prior to the study of the traces. This list is considered the canonical (i.e., presumed correct) list, and we compare each algorithm according to how good it is at picking out the methods in the list. For this, we use precision and recall metrics.

Precision is defined as the fraction of retrieved instances that are relevant or correct, while recall is the fraction of relevant instances that are retrieved. We need to look at both precision and recall to evaluate which algorithm is better at selecting the main methods. We do this by comparing the list picked by the algorithm with the canonical list.

We received a list for jUCMNav and Umple, which are in Appendix G and H respectively. We did not receive a list of main methods for UCEd. There are 70 methods in the list for Umple and 33 methods in the jUCMNav list. The list for Umple is bigger because two entire classes were included: UmpleInternal-Parser and UmpleModel. The algorithms were then run, and the top methods picked by each of the algorithms which were then compared against the lists produced by the designers. One of the artefacts of the TraceToUCM tool is a file containing a column of methods per algorithm, each containing the topmost non-utility methods according to each of the algorithms. The first column is initially empty and is filled with a list provided from the human expert. The output file containing the top non-utility methods is labeled o2 in Figure 6. The tool was set to pick the top 70 methods for Umple and top 33 for jUCMNav. Therefore there were 70 entries in the Umple file and 33 entries for the jUCMNav file. The formula used to

calculate the scores is the number of correct picks, versus the number of methods picked by the creator of the system.

The precision and recall formulas used for algorithms are as follows:

$$Recall(i) = \frac{A_i \cap C}{C}$$

$$Precision(i) = \frac{A_i \cap C}{A_i}$$

Where:

- A_i = the set of answers (methods) selected by algorithm i
- C = the correct set of answers provided by the human expert (e.g., 33 methods for jUCMNav and 70 methods for Umple)

In order to have a single performance measure, we calculate the weighted harmonic mean for Precision and Recall using the *F-Measure*, which is at its best when it reaches the value 1 (and at its worse when it gets to 0). The F-Measure formula used for algorithms is as follows:

$$F-Measure(i) = \frac{2 * Precision(i) * Recall(i)}{Precision(i) + Recall(i)}$$

Table 26 shows the recall scores for each of the eight algorithms for the jUCMNav trace. For jUCMNav, where there were 33 methods picked to be the top methods by the main creator of the system, algorithm 1 only selected 3 out of 33 which is only 9%. Algorithm 5 performed the best by selecting 8 out of 33, hence with a recall of 24%, yet it was scored to be the worst by the jUCMNav designers. This is because the UCM diagrams were generated using TraceToUCM the threshold was set to pick the top 17 distinct non-utility methods for each algorithm.

Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4	Algorithm 5	Algorithm 6	Algorithm 7	Algorithm 8
3	1	0	2	8	1	3	5
9.09%	3.03%	0.00%	6.06%	24.24%	3.03%	9.09%	15.15%

Total methods = > 33

Table 26 Recall values of the eight algorithms at selecting jUCMNav's main methods.

The threshold was set by trial-and-error to have the diagram fit in one page with a possibility of one level of stubs. In addition to the low threshold, the layout of the UCM diagrams may have played a factor.

Figure 25 highlights the performances of the eight algorithms for jUCMNav, which are listed in Table 27. Algorithm 5 scored high in recall and also precision. Algorithm 2 and 6 almost completely overlap in the chart because they have exactly the same recall values and almost the same precision value. Algorithm 3 has 0 precision and recall value therefore it is not charted on Figure 25.

	Algorithm 1		Algorithm 2		Algorithm 3		Algorithm 4		Algorithm 5		Algorithm 6		Algorithm 7		Algorithm 8	
Samples	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0.03	0.10	0	0	0	0	0.03	0.10	0.06	0.20	0.00	0	0.03	0.10	0.03	0.10
15	0.06	0.13	0.03	0.07	0	0	0.06	0.13	0.06	0.13	0.03	0.07	0.03	0.07	0.03	0.07
20	0.09	0.15	0.03	0.05	0	0	0.06	0.10	0.06	0.10	0.03	0.05	0.03	0.05	0.06	0.10
25	0.09	0.12	0.03	0.04	0	0	0.06	0.08	0.09	0.12	0.03	0.04	0.03	0.04	0.06	0.08
30	0.09	0.10	0.03	0.03	0	0	0.06	0.07	0.18	0.20	0.03	0.03	0.03	0.03	0.15	0.17
33	0.09	0.09	0.03	0.03	0	0	0.06	0.06	0.24	0.24	0.03	0.03	0.09	0.09	0.15	0.15

Table 27 Precision and Recall values of the eight algorithms at selecting jUCMNav's main methods.

As we can see from Table 27, at sample sizes 15 Algorithm 4, 5 and 1 do best at 15% Algorithm 2, 6, 7 and 8 are second at 7%. If the threshold was set at 15 instead of 33, and the UCM diagram had better layout than the other algorithms, it is possible that it would convey more meaning to the designers than other algorithms that have picked out the same or more methods but have poor layout.

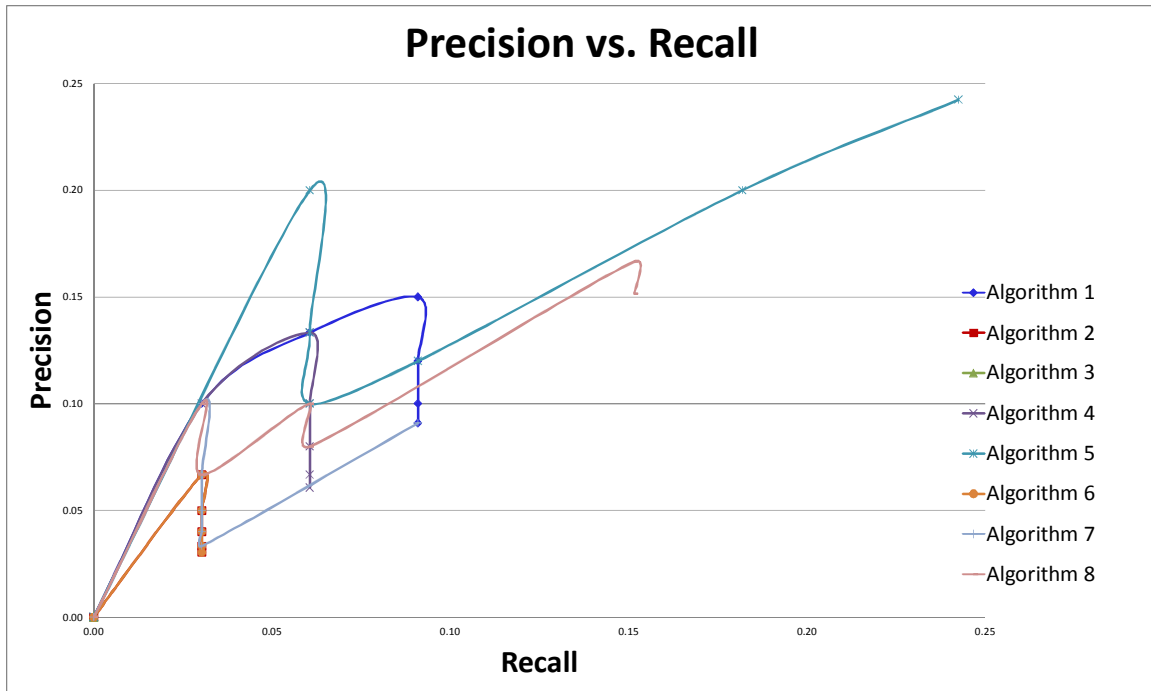


Figure 25. Precision versus recall graph for algorithms used on jUCMNav methods.

Figure 26 shows the harmonic means (F-Measures) based on the Precision and Recall values shown in Figure 25. It highlights similar results, with Algorithm 5 having the best result followed by Algorithm 8.

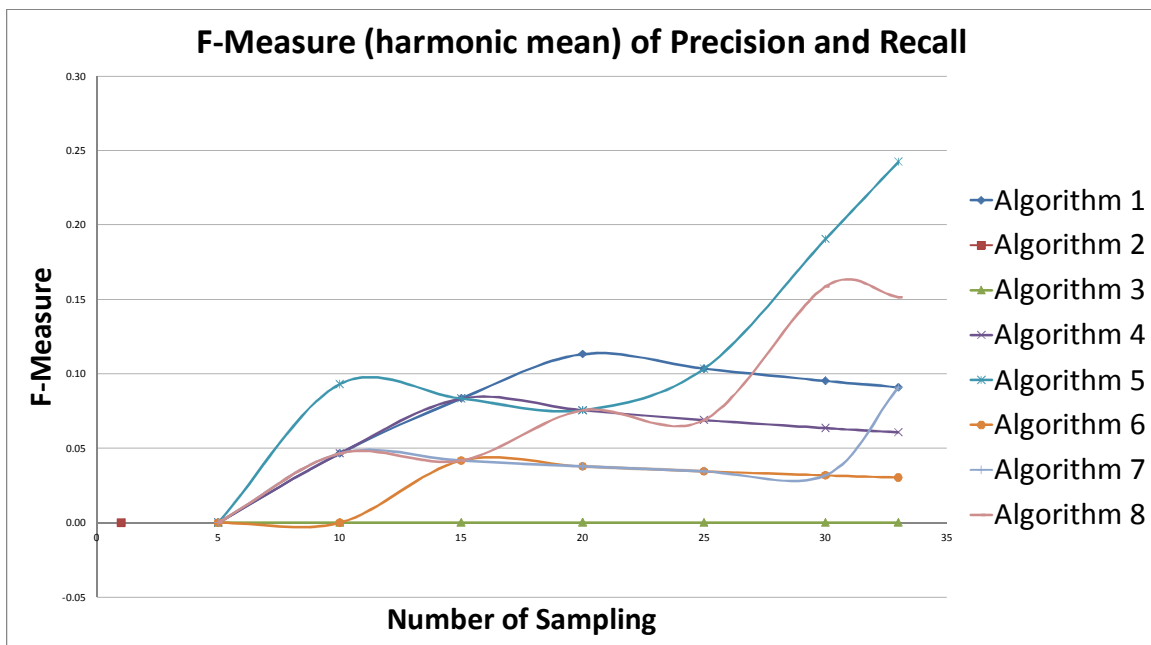


Figure 26. F-Measures of the eight algorithms for jUCMNav.

The human expert for Umple identified an entire class as being important for the feature, hence the large number 70. Table 28 shows the list of recall values (the number of answers divided by the total number of correct answers) for the eight algorithms for Umple in the tool.

Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4	Algorithm 5	Algorithm 6	Algorithm 7	Algorithm 8
16	34	27	31	3	30	8	28
22.86%	48.57%	38.57%	44.29%	4.29%	42.86%	11.43%	40.00%

Total methods =>

70

Table 28 Recall values of the eight algorithms at selecting Umple's main methods.

Figure 27 shows the performances of the eight algorithms when used to select the main methods of Umple. Algorithm 2 did better in general followed by Algorithms 4 and 6.

	Algorithm 1		Algorithm 2		Algorithm 3		Algorithm 4		Algorithm 5		Algorithm 6		Algorithm 7		Algorithm 8	
Sample	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision
5	0	0	0.03	0.40	0	0	0.03	0.40	0	0	0	0	0	0	0.01	0.20
10	0.06	0.40	0.06	0.40	0.04	0.30	0.06	0.40	0	0	0.03	0.20	0.00	0.00	0.06	0.40
15	0.06	0.27	0.11	0.53	0.11	0.53	0.13	0.60	0	0	0.10	0.47	0.01	0.07	0.11	0.53
20	0.06	0.20	0.16	0.55	0.13	0.45	0.14	0.50	0	0	0.11	0.40	0.03	0.10	0.14	0.50
25	0.06	0.16	0.23	0.64	0.19	0.52	0.20	0.56	0	0	0.17	0.48	0.04	0.12	0.19	0.52
30	0.10	0.23	0.30	0.70	0.21	0.50	0.27	0.63	0	0	0.24	0.57	0.07	0.17	0.24	0.57
35	0.11	0.23	0.36	0.71	0.27	0.54	0.31	0.63	0	0	0.29	0.57	0.07	0.14	0.30	0.60
40	0.13	0.23	0.39	0.68	0.30	0.53	0.34	0.60	0	0	0.31	0.55	0.07	0.13	0.33	0.58
45	0.14	0.22	0.41	0.64	0.31	0.49	0.36	0.56	0	0	0.33	0.51	0.10	0.16	0.34	0.53
50	0.16	0.22	0.43	0.60	0.33	0.46	0.39	0.54	0.01	0.02	0.37	0.52	0.11	0.16	0.36	0.50
55	0.17	0.22	0.44	0.56	0.33	0.42	0.39	0.49	0.01	0.02	0.37	0.47	0.11	0.15	0.36	0.45
60	0.19	0.22	0.46	0.53	0.33	0.38	0.41	0.48	0.03	0.03	0.40	0.47	0.11	0.13	0.36	0.42
65	0.21	0.23	0.46	0.49	0.33	0.35	0.41	0.45	0.04	0.05	0.40	0.43	0.11	0.12	0.36	0.38
70	0.23	0.23	0.49	0.49	0.39	0.39	0.44	0.44	0.04	0.04	0.43	0.43	0.11	0.11	0.40	0.40

Table 29 Precision and Recall values of the eight algorithms at selecting Umple's main methods.

The Umple expert picked 70 methods out of 395 distinct methods in the execution trace, this accounts for the better results in the precision and recall scores for Umple than for jUCMNav. Table 29 gives the detailed performance of the eight algorithms both for precision and recall.

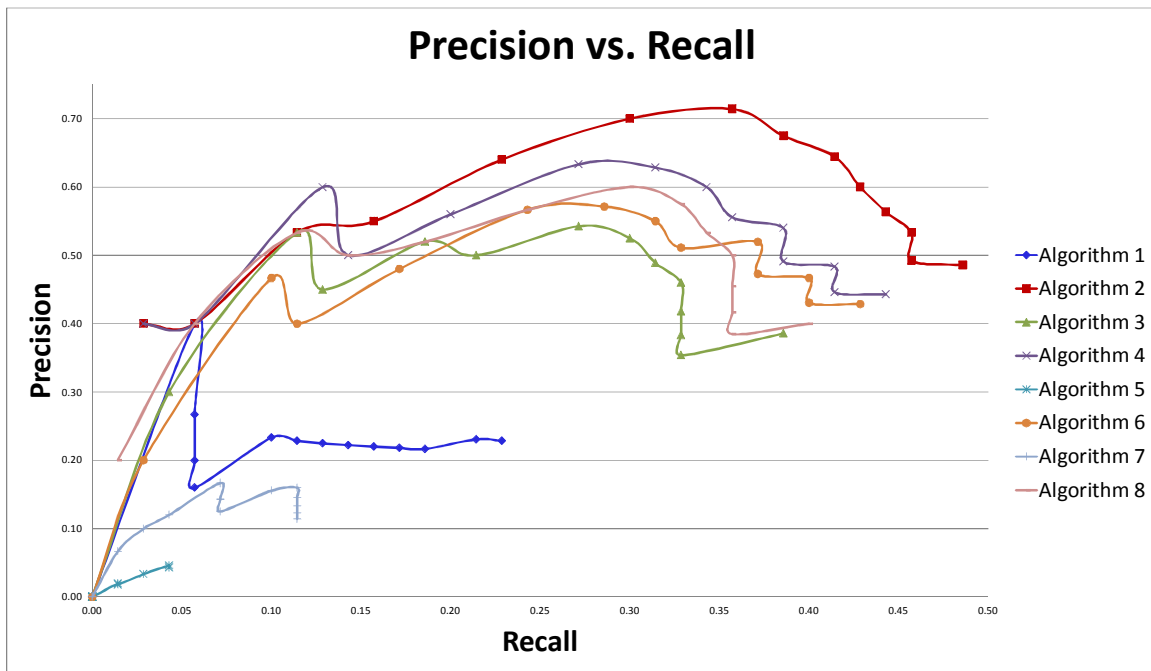


Figure 27. Precision versus recall graph for algorithms used on Umple methods.

Figure 28 shows the F-Measures of the algorithms for Umple, again with similar results: Algorithms 2, 4 and 6 outperform the rest of the algorithms.

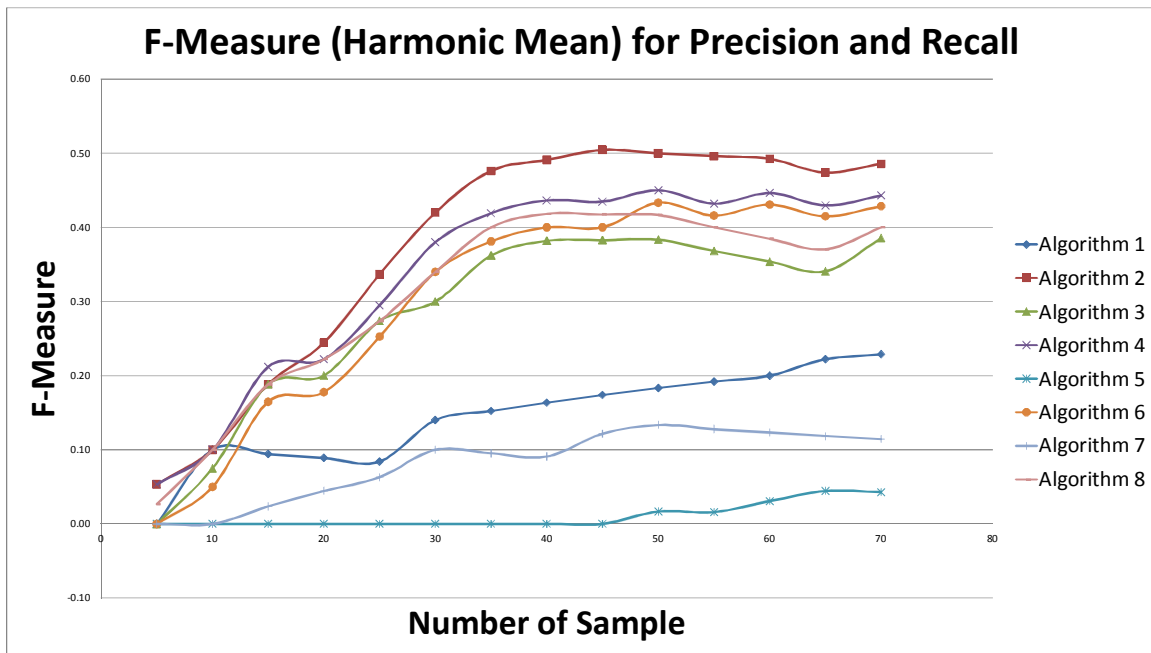


Figure 28. F-Measures of the eight algorithms for Umple.

Doing precision and recall helped us answer the question, “are the methods selected by this algorithm correct” independent of the visualization technique used.

The human experts were not limited as to how many “main” methods they identified for the selected features. However, the UCM drawing algorithms were limited by the number of methods that will fit in one page of UCM diagram. This caused all algorithms to be scored better using the precision and recall calculation as opposed to the UCM diagram based scores given by the designers. For example, for Algorithm 5, the first 33 methods were taken into account for the precision and recall calculations but only the first 17 distinct methods were used to draw the UCM diagram.

The experiment can be improved by improving the UCM drawing code to generate multiple layers of stubs, then asking the creator to pick the “main” methods and then running the tool to generate UCM diagrams setting the threshold to be the same as the number of methods picked by the creator of the system.

7.7.2 Comparison of Algorithms

In order to compare the answers for the question “Are the methods selected by this algorithm correct?” we received from all the designers, we compiled their answers into one table (Table 30). From the responses, we can see the three groups of designers do not agree on any single algorithm. In general, the algorithms worked better for UCED, followed by Umple and last for jUCMNav, which can explain the almost complete opposite marks given by jS1 and uS2. Algorithm 2 and Algorithm 4 do better than others with a total score of 2 out of a possible 18 (9 participants with ratings of -2..2 each). Table 31 shows the ranking of all the algorithms (last column), together with the parameters they have used. There is no clear winner; however, there is a trend: the parameters McCabe, MLOC and NBD are used in the algorithms ranked 1 to 3.

	Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8
jUCMNav	jS1	-2	-2	-2	-2	-2	-2	-2	1
	jS2	-1	1	-1	1	0	-1	0	-1
	jS3	1	1	1	0	-2	1	-1	1
UMPLE	uS1	1	-1	0	0	0	0	-1	-1
	uS2	1	1	1	1	1	1	1	0
	uS3	-1	-1	1	0	-1	-1	0	0
	uS4	-2	1	2	-1	-2	0	-2	-2
UCED	uCS1	2	1	-1	2	-2	1	0	-2
	uCS5	1	1	0	1	-1	-2	1	-1
Summary		0	2	1	2	-9	-3	-4	-5

Legend

Strongly disagree	-2
disagree	-1
neutral	0
agree	1
strongly agree	2

Table 30 Summary of responses from all participants (algorithm's selection of methods).

On the negative side, the summary of all participants puts Algorithm 5 to be the worst for the job of selecting main methods for a feature. Algorithm 5 also gets the most -2 (strongly disagree). The best mark algorithm 5 gets is a 1 by one of the Umples designers.

Algorithms	Fan-in	Fan-out	Relative Method Size	Complexity of a Method	Nested Block Depth	Average CPU usage by Method	Segmentation Presence	Depth of Call hierarchy	Rank
Algorithm 1	X								3
Algorithm 2	X	X							1
Algorithm 3			X	X	X				2
Algorithm 4		X	X	X	X				1
Algorithm 5							X		7
Algorithm 6	X	X							4
Algorithm 7	X		X					X	5
Algorithm 8			X	X		X			6

Table 31 Summary of parameters used for each of the eight algorithms.

Table 31 also shows that Depth of Call Hierarchy, CPU and Segmentation Presence are in the lowest scoring algorithms. It was surprising to see Algorithm 8 at the bottom, because it has been ranked as the best algorithm by jUCMNav designers and it shares 2 parameters with the winning Algorithm 4.

7.7.3 Algorithm Parameter Comparison

We looked closer at which group of designers chose which algorithms. Table 32 shows which algorithms scored first in each of the three systems. Even though the three sets of designers did not agree on which algorithm performed the best, there is a common trend among the algorithms that were selected by each group. The jUCMNav designers favoured Algorithm 8 with a total score of 1 (very weak agreement), the Umple designers chose Algorithm 3 with a score of 2 (also very weak agreement), and the UCED designers chose Algorithms 1 and 4, both with a score of 3 out of possible 4 (strong agreement). See Table 17, Table 20 and Table 23 for the detailed responses for jUCMNav, Umple and UCED respectively. Method size and method complexity were in 3 out of the 4 best algorithms. Nested block depth was used in 2 of the 4 algorithms. Fan-in, Fan-out and CPU were only used in one of the algorithms.

Software Systems	1st choice
jUCMNav	Algorithm 8 ¹
Umple	Algorithm 3 ²
UCED	Algorithms 1 and 4 ³

Table 32 First choice of algorithms by the three sets of designers.

If we expand the algorithm selection from first only to the first three choices of each group, we can see that Algorithm 4 shows up in all three groups. In addition, Algorithm 4 scores 3 out of 4 for UCED, which is the highest any algorithm scored.

Software System	1st choice	2nd choice	3rd choice
jUCMNav	Algorithm 8 ¹	Algorithm 2 ^o	Algorithm 4 ¹
Umple	Algorithm 3 ²	Algorithm 4 ¹	Algorithm 2, Algorithm 6 ^o
UCED	Algorithms 1 and 4 ³	Algorithm 2 ²	Algorithm 7 ¹

Table 33 Top 3 choices of algorithms by the 3 sets of designers.

7.7.4 Possible Further Improvement of the Algorithms

There are two improvements to our approach that would improve the tool for all algorithms. They are:

- 1) Support for multiple layers of stubs. In order to effectively visualize reduced traces, the TraceToUCM tool needs to be improved to produce UCM diagrams with multiple nested levels of stubs.
- 2) Higher thresholds for the number of methods to be displayed in UCM diagrams. The algorithms performed much better when the threshold was set to 70 to match the list given by the Umple expert.

Algorithm 7, based on the depth of call hierarchy and CPU usage, did not show much promise. Better investment of time would be to focus on Algorithms 1 to 6.

Even though Algorithm 5, based on Dugerdil's segmentation presence [30], scored low for jUCMNav, we still believe it could be improved and used to detect utilities in conjunction with other static or dynamic parameters. One obvious improvement for the segmentation algorithm is to use a logarithmic function to segment the traces, rather than the discrete function in Table 14. Another improvement would be to use the number of times a method is found in a trace (EncounteredinTrace) in addition to segmentation presence, which would help ensure that the segmentation algorithm does not skew the result.

7.8. Summary

This chapter presents the reverse-engineering case studies used to validate which algorithmic strategies enable good detection and filtering of utilities. We reverse-engineered 3 software systems, namely jUCMNav, UCed and Umple, and asked their developers to evaluate the results. We had 3 participants for jUCMNav, 4 participants for Umple and 2 participants for UCed.

The 3 jUCMNav designers favoured Algorithm 8 for having picked the most correct methods. They also picked the UCM diagram generated by Algorithm 2 for being closer to their mental image. When the threshold for the utility filtering is set low, Algo-

rithm 5 was the least favoured; however, when the threshold is raised from 17 to 33, it scores the best, suggesting that users prefer to cope with a trace containing more than 30 elements than with one showing fewer than 20. The precision and recall calculations show Algorithm 5 was the best at matching the canonical list provided by one of the main designers, with 24% of the methods. The second place went to Algorithm 8 with 15%. However, the algorithm most favored by the designers, Algorithm 2, only scored 3%.

The 4 Umple designers favored Algorithm 3, and they also picked this algorithm for being closer to their mental image. However, when the threshold is set at 70, the precision and recall calculations showed that Algorithm 2 performed the best at matching the canonical list. Algorithm 2 picked 49 % of the methods, with the second place going to Algorithm 4 at 44%. The algorithm most favored by the designers, Algorithm 3, scores fourth at 39%. The discrepancy between the preference of the designers and the precision and recall calculations is not as stark as it was for jUCMNav. The layout of the diagram might be an issue. A UCM diagram using Algorithm 3 with 27 of the 70 “main” methods could be chosen over a UCM diagram using Algorithm 2 with 34 of the 70 “main” methods, if it had better layout.

The 2 UCED designers favoured Algorithms 1 and 4 for picking the correct methods, and they picked Algorithm 1 for being closer to their mental image. Unfortunately we were not able to get a list of main methods from a human expert for UCED, and therefore we were not able to do precision and recall analysis.

Looking at the scores from the 3 groups of designers, Algorithms 2 and 4 come ahead of the other algorithms. The algorithms that did better than the others used *method size*, *method complexity* and *nested block depth*. More work needs to be done to fine-tune the combinations of those parameters in the algorithms.

In general, there was agreement that UCM diagrams were suitable to represent execution traces. 7 out of the 8 designers agreed (including one who strongly agreed) and one was neutral.

In the following chapter we will present another case study that investigates the use of reduced execution traces, visualized as UCM diagrams, to document the behaviour of a feature. We will validate this case study with designers who are not familiar with the reverse-engineered system.

Chapter 8. Case Study on Reverse-Engineered Behavioural Diagrams as Code Documentation

In this chapter, we present a case study that was done to evaluate the use of reduced execution traces, visualized as UCM diagrams, to document the behaviour of features. In addition to the original goal of using UCM visualized traces to understand a software system written by other designers, we want to test whether this methodology can be used by designers to document their own code. We envision the UCM behavioural model being used as a dynamic model of the software to complement a static model presented by class diagrams.

We test this idea on our own TraceToUCM software by creating UCM diagrams based on the tool itself. We show the UCM diagrams to several subjects and get them to rate the UCM diagrams. The main difference between this case study and those presented in Chapter 7 is that the participants in this case study are not the designers of the reverse engineered software, but users of the UCM diagrams.

8.1. Description of the Study

The objective of this study is to test if reverse engineered behavioural maps of a feature can be used as an aid to software description or software documentation and if so, which of the algorithms produce the best diagram.

We selected TraceToUCM as a test system. TraceToUCM is a one-feature program; therefore, we run the only feature it has, which is to take dynamic and static information about a system, filter out details, and produce several UCM diagrams that attempt to describe the feature that was run.

TraceToUCM was run on a JHotDraw execution trace and an execution trace of the run was collected. The original trace came in at 162,483 lines. Table 34 gives a summary of the TraceToUCM execution trace files. We can see from that table that before we

apply the utility detection algorithms the trace has been reduced by 99%. To handle small trace files, we have a condition in the tool that if the number of lines has been reduced to less than 100, it will produce a UCM diagram at the end of the filtering step (without any algorithms applied) *as well as* after utility detection steps. Therefore, even though we only have 8 algorithms, a total of 9 UCM maps were produced. The ninth UCM diagram is labelled Uss in Figure 6. The reason for generating a UCM diagram for traces that are small is to test if filtering is necessary for very small traces.

File name	File size	% of original size
Trace_to_UCM_Original	162483	100.00%
Trace_to_UCM_OriginalLoopsRemoved	7704	4.74%
Trace_to_UCMAcR_ExitMarkerRemoved	3449	2.12%
Trace_to_UCMAcRAccessMethodsRemoved	2399	1.48%
Trace_to_UCM_Original_SmallSimpleMethodsRemoved	27	0.02%
Trace_to_UCMAlgorithm_1	4	0.00%
Trace_to_UCMAlgorithm_2	9	0.01%
Trace_to_UCMAlgorithm_3	22	0.01%
Trace_to_UCMAlgorithm_4	4	0.00%
Trace_to_UCMAlgorithm_5	23	0.01%
Trace_to_UCMAlgorithm_6	9	0.01%
Trace_to_UCMAlgorithm_7	9	0.01%
Trace_to_UCMAlgorithm_8	22	0.01%

Table 34 TraceToUCM execution trace line count.

Once the UCM diagrams were produced, we gave them to participants who are not familiar with the TraceToUCM software. Care was taken to not re-use participants that had participated in the case studies involving the reverse-engineering case study, to avoid any bias linked to learning effect.

All of the participants that were contacted are in the computer science field. Sixteen people volunteered to participate: 12 are graduate students (at the University of Ottawa or at Carleton University), and the 4 others are professional developers. All except 2 participants were familiar with the UCM notation.

8.2. Validation of UCM Diagrams

The participants were shown the nine UCM diagrams in Appendix E. They were told that they would be looking at nine behavioral models of a software system that reads in an execution trace and produces several UCM diagrams. They were asked to pick the top three UCM diagrams that they felt gave enough information to describe what the program did. Table 35 gives a summary of the participant's responses (with their rankings of the top 1 to 3 algorithms).

Subjects	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Small and simple removed
S1			2					3	1
S2			3					2	1
S3			3					2	1
S4	1	3		1					
S5		3				2	1		
S6			3			2			1
S7		1	2		3				
S8				2			1	3	
S9			3					2	1
S10			2					3	1
S11			1		1			3	(4)
S12			(4)		2			2	1
S13	1			1		3	3		
S14	1	3		1	3	3			(6)
S15					3			2	1
S16	(4)			(4)		2	2		1

Ranking	Alg 1	Alg 2	Alg 3	Alg 4	Alg 5	Alg 6	Alg 7	Alg 8	Small & simple
First	3	1	1	3	1	0	2	0	9
Second	0	0	3	1	1	3	1	5	0
Third	0	3	4	0	3	2	1	4	0
Count	3	4	8	4	5	5	4	9	9

Table 35 Summary of results: UCM visualized trace as documentation.

A few of the participants had problems deciding on how to order their top 3 picks. Some could not decide and picked multiple algorithms for each rank. This was especially difficult because a few algorithms produced identical UCM maps (e.g., Algorithms 1 and 4, and Algorithms 6 and 7), but this was not noticed by everyone. For example, participant S4 ranked Algorithms 1 and 4 to be ex æquo in first place (with the third place going to Algorithm 2), and S14 picked Algorithms 2, 5 and 6 to be ex æquo in third place (with S14's third choice hence being ranked sixth). One of the participants could not decide between the algorithms because each highlighted a different aspect of the feature.

Though the “small and simple” trace ranks first in Table 35, with 9 first choices, the ranking of the second and third places is less clear, especially as people had a hard time deciding on a ranking. We therefore decided to rate the remaining algorithms based on the count of their first three ranks (i.e., each participant picked three algorithms as useful). The last line in Table 35 provides this count.

The ranking shows that the UCM diagram that was drawn right after the filtering step (without any utility detection algorithm) did the best, followed by Algorithms 8, and then Algorithm 3:

1. UCM diagram drawn after small and simple methods are filtered (Figure 29, see next page)
2. UCM based on CPU usage, Algorithm 8 (Figure 40)
3. UCM based on McCabe complexity, Algorithm 3 (Figure 35)

Note that the above 3 algorithms are original contributions of this thesis and the results observed suggest that they perform slightly better than related work (Algorithm 2 comes from Hamou-Lhadj [46], Algorithm 5 comes from Dugerdil and Repond [31], and Algorithm 6 comes from [118]).

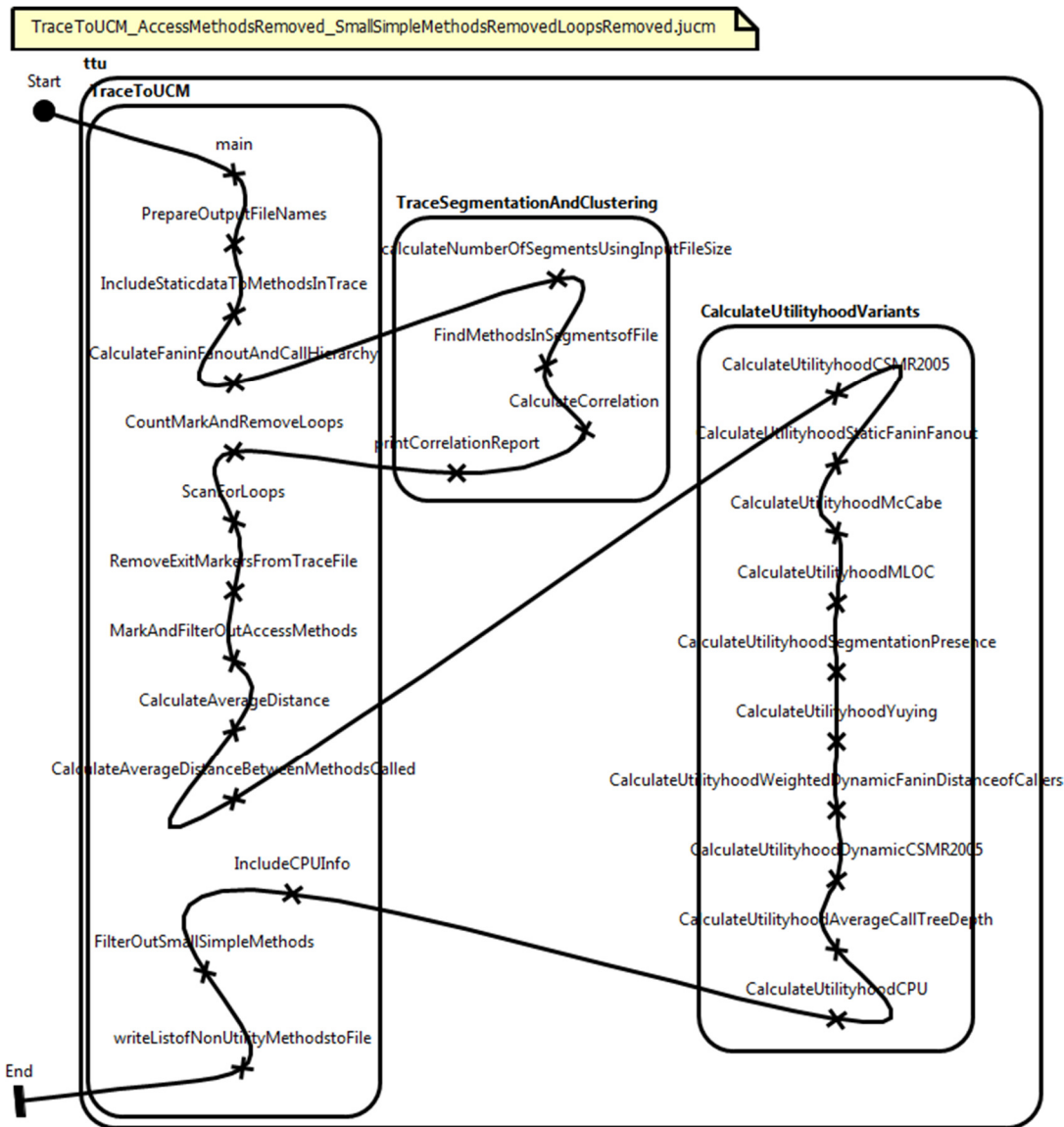


Figure 29. TraceToUCM diagram after small and simple methods are filtered.

8.3. Lessons Learned

As the designers of the TraceToUCM system, we were interested to see how much the participants of this study could understand about the behaviour of the program from the UCM diagrams. We had a chance to observe two of the participants do the study. Both participants read out the responsibilities (methods) on the UCM path sequentially to come up with a narrative of what the system may be doing. One said that since this is a program that produces UCM diagrams as an output, he expected to see a method saying

something like “createUCM” at the end of the UCM path. This can be improved by reading the UCM diagram sequentially and adding any perceived missing methods, like createUCM in this case, to the include file so it would not get filtered out.

Most of the participants picked the UCM diagram that was produced immediately after the filtering step, which had cut out more than 99% of the methods. There simply was not enough left to cut for the other algorithms. If the UCM diagrams were to be redrawn by setting the threshold (using the `-precise` switch) to be 27 for all of them, we could make file size constant and have reliable data on the preferences of the participants. In the case studied, 5 of the algorithms had 9 items or less, which puts them at a disadvantage and skewed the results towards the algorithms that had 22 to 27 items. Algorithm 9 got the most votes to be number one. This could be because it shows the most information without being cluttered.

We learned that some of the method names are misleading or incomplete. One participant said that responsibilities need to have better names. This is an important comment to receive as a designer of a system, and it is also an easy problem to fix; when our methodology is used as documentation, the designer can simply make the changes in his/her code and generate another model. This may even be considered a strength of the approach, a way of highlighting naming inconsistencies. However, this points out a limitation when our methodology is used to reverse-engineer old code: the UCM maps are only as readable as the existing method names.

One of the subjects commented that instead of using generated UCMs to document a whole feature, he would use it to document the behaviours of one large class. He said he would generate *several traces* of one large class with *different inputs* and include the generated UCM diagrams in the design documents as figures. He said they have a legacy program they maintain and they have a class that they know “behaves” differently based on the input parameters. He said it would be nice to map the behavior of the class based on different inputs. This is indeed possible with the tool by excluding all but one class in the tool, as explained in Chapter 6.

By far, the most common comment was that participants did not like having to choose only one diagram. To summarize, they said it would be good to have *several UCM diagrams per feature*, each labelled with what aspect they emphasize. For example,

we could have one UCM diagram that showed the top CPU users' methods for that feature, and another one showing the most complex methods, and another one showing the largest methods.

In general, we believe that automatically generated behavioral models, with some improvement, could help address the need for software documentation. The study could also be repeated using participants who are not as experienced with UCM as those used for this study. It should also be repeated with traces reduced to the exact same length, to remove the length variable from the evaluation. For example, in the experiment above the trace generated without the use of utility detection algorithms had a size of 27, and the – 'precise' switch of TraceToUCM can ensure that traces from the eight other algorithms have that size too.

8.4. Summary

This chapter presented a case study that investigates whether UCM diagrams produced from reduced traces can be used to aid in feature documentation and, if so, which algorithm produces the best diagram. The execution trace used was too small to decisively pick an algorithm to be the best; however, the results showed that reduced traces, visualized by using UCM notation, can be helpful in documenting software. We learned that it may not be desirable to pick one "best" algorithm since a few of the participants expressed they had multiple favorites.

In the next chapter, we discuss the validation techniques used for our work and some of the threats to the validity of the methodology.

Chapter 9. Discussion

In this chapter, we discuss the validation technique used for our work. Figure 6 highlights the three steps of the TraceToUCM process, namely trace filtering, utility detection, and visualization. These are used to structure our validation. The first section of this chapter covers the validation of the trace filtering technique. The second section covers the validation of the utility detection algorithms themselves, including a discussion of which of the proposed algorithms from Chapter 5 detected the utilities correctly. The third section details the validation of the visualization approach used. This chapter concludes with a discussion of the general threats to the validity of this work, and a summary.

9.1. Validation of the Trace Filtering Step

At this step of the approach, we take the original trace and filter out methods with certain characteristics, like size of method, before we put the traces through the eight algorithms.

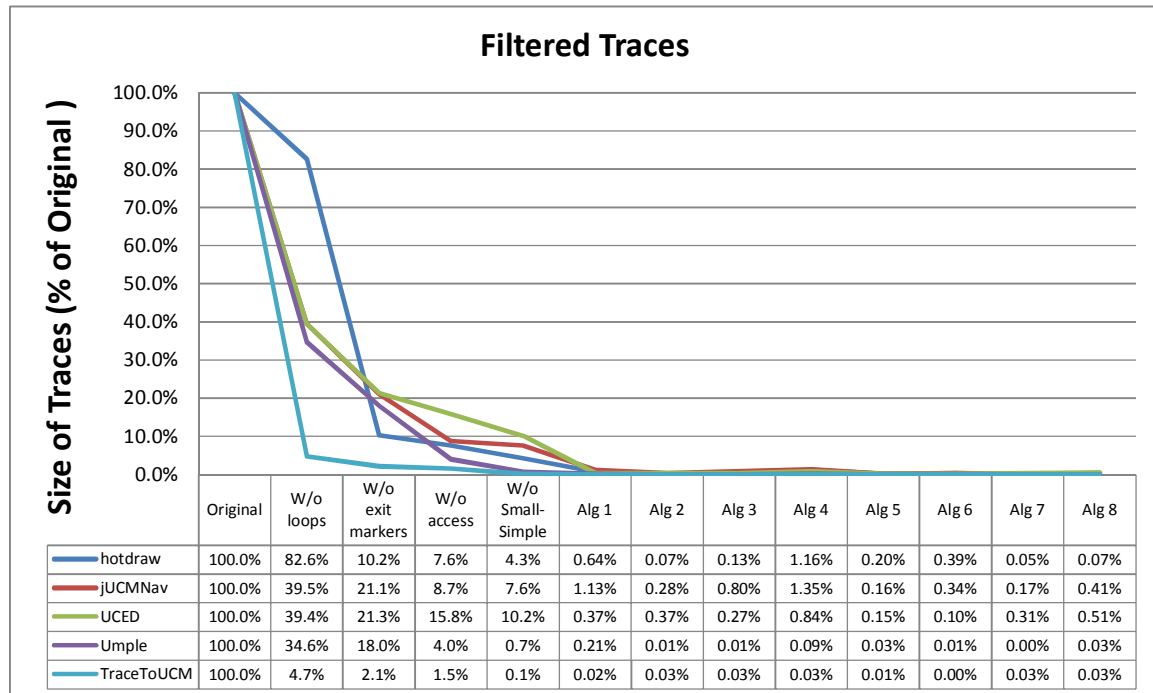


Figure 30. Size of filtered traces, in percentage of the original size.

Figure 30 shows the results of the filtering step. It is interesting to see that traces were reduced by 80% to 98% simply by removing repeated loops, access methods, and very small and simple methods. This is encouraging because by tuning the filtering process, the burden on the algorithms to reduce the traces is lightened.

9.1.1 Validation Approach

To validate if the filters were working as expected, we used two different levels of detail: one for the smaller programs (more detailed) and one for three large programs (less detailed). For the 14 small programs with short execution traces (Table 2), validation was done by comparing the original trace file with the filtered trace to see what had been removed. We also looked at the source code to verify that the methods we expected to have filtered out were in fact the methods that were filtered out. For the three large programs (jUCMNav, Umple and UCed), validation of the filtering was also done by manually inspecting the table that was generated to store the static and dynamic data for each of the methods in the original trace. Table 36 shows one such table for the DND software system. The data table is labelled o1 in Figure 6.

Three properties of the methods were used to filter the trace before the utility detection algorithms were used to identify and eliminate utility methods. They were size of method (MLOC), complexity of method (McCabe), and whether a method was an access method (DataAccessMethod). In the following subsections we look at each of those properties one by one.

TraceItemName	Dynamic Parameters											Static Parameters			
	FanIn	FanOut	UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR
main	0	2	0	2	2	1	0	0	39.5	0.41	1	4	1	1	1
open	1	24	1	12	25	1	0	0	39.1	35.9	1	123	5	2	1
createDragWidget	1	3	1	2	4	1	0	0	0.03	0.03	1	51	1	4	1
createWidget	4	1	3	1	5	2.5	0	0	0.01	0	4	116	15	6	3
createDragOperations	1	3	1	1	4	1	0	0	0.01	0.01	1	62	1	4	1
createDragTypes	1	6	1	2	7	1	0	0	0.02	0.02	1	76	2	4	1
addDragTransfer	4	0	2	0	4	2.75	0	1	0	0	4	7	2	2	1
createDropWidget	1	3	1	2	4	1	0	0	0.01	0.01	1	51	1	4	1
createDropOperations	1	8	1	1	9	1	0	0	0.01	0.01	1	146	4	4	1
createDropTypes	1	5	1	2	6	1	0	0	0	0	1	51	1	3	1
addDropTransfer	4	0	2	0	4	2.75	1	1	0	0	4	7	2	2	1
createFeedbackTypes	1	5	1	1	6	1	0	0	0.01	0.01	1	61	1	3	1
createDragSource	2	2	2	1	4	1	0	0	0.05	0.02	2	211	12	6	0
createDropTarget	2	2	2	1	4	1	0	0	0.03	0.02	2	142	8	5	0
Correlation with fan-in	0.89	-0.3	1	-0.3	-0.1	0.68	0.25	0.37	-0.5	-0.1	0.83	0.34	0.75	0.56	0.3
Correlation with fan-out	-0.3	0.94	-0.3	1	0.93	-0.3	-0.2	-0.3	0.71	0.97	-0.4	0.2	0.01	-0.3	0
Correlation with MLOC	-0	0.29	0.34	0.2	0.3	-0.4	-0.3	-0.5	-0.1	0.2	-0.1	1	0.73	0.73	-0.2
Correlation with McCabe	0.46	-0	0.75	0.01	0.06	0.23	-0.1	-0.2	-0.1	0.06	0.45	0.73	1	0.71	0.25
Correlation with EncounteredInTrace	-0.2	0.98	-0.1	0.93	1	-0.2	-0.1	-0.2	0.54	0.95	-0.2	0.3	0.06	-0.2	0.04
Correlation with AverageCallHierarchy	0.93	-0.4	0.68	-0.3	-0.2	1	0.56	0.83	-0.2	-0.1	0.96	-0.4	0.23	-0.1	0.48
Correlation with DataAccessMethod	0.5	-0.2	0.25	-0.2	-0.1	0.56	1	0.68	-0.1	-0.1	0.51	-0.3	-0.1	-0.3	0
Correlation with Leaf	0.73	-0.3	0.37	-0.3	-0.2	0.83	0.68	1	-0.2	-0.1	0.75	-0.5	-0.2	-0.4	0

Table 36 Data table (o1) with static and dynamic data on all methods in a trace.

9.1.2 Cyclomatic Complexity of Methods

Cyclomatic complexity, developed by Thomas J. McCabe in 1976 [64][69], is a measure of software complexity that Section 4.3.2 discussed in detail. We gathered the complexity measures for methods using Metrics 1.3.6 [53].

We had a theory that simple methods would tend to be small. We ran a correlation test to determine whether a) this observation was true, and b) what other characteristics the complexity of methods was correlated with. Table 37 gives the correlation of method complexity with other parameters (this is the same table as Table 9, repeated here for convenience). We noted that there was a positive correlation between complexity and relative size of method and complexity and nested block depth 0.628 and 0.647 respectively.

Based on this correlation, we decided to filter out all methods that were simple (i.e., that had a low complexity index). We reason that since very simple methods tend to

be small, they are likely utilities. Therefore we went through the methods and filtered out all methods that had a McCabe complexity of two or less.

System name	Number of methods	Dynamic Data								Static		
		UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	MLOC	NBD	PAR
HelloWorld	2	0.093	0.177	0.042	-0.126	-0.411	0.146	0.000	-0.294	0.483	0.496	0.296
HoverHelp	6	-0.189	0.615	0.092	-0.658	-0.586	-0.348	0.926	0.022	0.612	-0.586	
ClipBoard	9	0.316	0.387	0.419	0.316	-0.316	-0.316	0.350	0.050	0.663	0.637	0.350
BrowserExample	10	-0.109	0.034	-0.112	-0.205	-0.321	-0.370	0.271	-0.144	0.991	0.558	0.890
DND	14	0.750	0.006	0.060	0.226	-0.127	-0.187	0.569	-0.094	0.733	0.709	0.249
TextEditor	14	-0.027	-0.410	0.373	0.177	-0.166	0.670	0.594	0.020	0.165	0.723	0.375
Address Book	16	-0.034	-0.012	-0.141	0.308	-0.150	0.011	-0.206	0.162	0.254	0.625	-0.331
JavaViewer	20	0.205	0.239	0.773	0.241	-0.195	-0.177	0.528	-0.057	0.907	0.759	0.072
ImageAnalyser	26	0.035	0.088	-0.090	0.008	0.093	0.116	-0.145	-0.105	0.291	0.647	0.073
LayoutExample	35	0.039	-0.103	0.107	0.040	-0.143	0.025	0.056	0.013	0.422	0.399	0.627
TraceToUCM on jhotdraw TPTP	66	-0.480	0.496	0.466	-0.472	-0.045	-0.737	-0.183	0.544	0.888	0.844	-0.077
PaintExample	104	-0.066	0.180	-0.024	-0.239	-0.355	-0.222	-0.122	0.006	0.574	0.736	0.059
Graphics	213	0.093	0.177	0.042	-0.126	-0.411	0.146	0.000	-0.294	0.483	0.496	0.296
Control	288	0.017	0.077	0.085	-0.007	-0.289	-0.241	0.077	0.001	0.523	0.578	0.044
jucmnav TPTP_Nov26AfterExclu	305	0.145	-0.006	0.191	0.220	-0.197	0.046	0.328	0.027	0.643	0.487	0.011
Umple_compiling_BCMS TPTP	351	-0.091	0.525	-0.025	0.041	-0.271	-0.293	-0.067	0.012	0.605	0.627	0.267
Uced TPTP_open file_Nov12Af	458	-0.039	0.088	0.040	0.087	-0.209	-0.112	0.026	-0.025	0.701	0.701	0.281
JHotDraw	566	-0.055	0.151	0.008	-0.017	-0.256	-0.200	0.076	-0.036	0.699	0.766	0.090
		-0.013	0.170	0.063	0.016	-0.252	-0.146	0.062	-0.017	0.628	0.647	0.156

Table 37 Correlation of static and dynamic variables with McCabe.

The result of the filtering for smaller programs was validated by inspecting the source code and the trace file, and for the bigger programs by also doing spot checks.

9.1.3 Size of Methods

Method lines of code (MLOC) is a count of non-blank and non-comment lines in the body of a method. Section 4.3.2 gives a more detailed discussion of MLOC.

We hypothesized that small methods were not likely to be the main methods of a program. To test this hypothesis, we calculated the correlation between the MLOC values of methods in the execution trace with several of the dynamic and static parameters that we collected. The results of the correlation are recalled in Table 38 (this is the same table as Table 10, repeated here for convenience).

System name	Number of methods	Dynamic Data								Static Data		
		UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	McCabe	NBD	PAR
HelloWorld	2	-0.019	0.261	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	0.483	0.570	0.473
HoverHelp	6	0.012	-0.088	-0.140	-0.378	-0.442	-0.442	-0.381	-0.127	0.022	0.595	-0.442
BrowserExample	9	-0.146	0.081	-0.075	-0.231	-0.272	-0.445	0.256	-0.108	0.991	0.446	0.932
ClipBoard	9	0.466	0.195	0.575	0.308	-0.466	-0.493	0.678	-0.201	0.663	0.593	0.066
DND	14	0.338	0.198	0.300	-0.350	-0.341	-0.502	0.225	-0.110	0.733	0.733	-0.225
TextEditor	14	0.253	-0.325	-0.013	-0.055	-0.308	-0.076	0.228	-0.260	0.165	0.437	-0.213
Address Book	16	0.179	0.198	0.525	-0.101	-0.341	-0.582	0.346	0.695	0.254	0.720	-0.077
JavaViewer	20	0.276	0.262	0.647	-0.015	-0.262	-0.235	0.318	-0.099	0.907	0.879	0.009
ImageAnalyser	26	0.025	0.137	-0.128	-0.268	-0.108	-0.280	-0.143	-0.102	0.291	0.504	-0.250
LayoutExample	35	-0.022	0.209	0.015	-0.003	-0.298	-0.438	-0.039	-0.155	0.422	0.702	0.046
TraceToUCM_on_jhotdraw_TPTP	66	-0.753	0.765	0.716	-0.422	-0.039	-0.691	-0.185	0.787	0.888	0.743	-0.106
PaintExample	104	-0.089	0.306	-0.059	-0.240	-0.261	-0.199	-0.176	-0.057	0.574	0.711	0.016
Graphics	213	-0.019	0.261	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	0.483	0.570	0.473
Control	287	-0.012	0.165	0.154	-0.049	-0.422	-0.415	0.095	-0.018	0.523	0.553	0.056
jucmnav_TPTP_Nov26AfterExclu	305	0.028	0.107	0.252	0.047	-0.183	-0.066	0.134	0.028	0.643	0.335	0.146
Umple_compiling_BCMS_TPTP	351	-0.031	0.695	-0.001	-0.043	-0.059	-0.078	-0.029	0.005	0.605	0.270	0.113
Uced_TPTP_open_file_Nov12Af	458	-0.075	0.151	-0.019	-0.080	-0.251	-0.099	-0.083	-0.039	0.701	0.640	0.232
JHotDraw	566	-0.078	0.350	0.013	-0.132	-0.268	-0.252	0.005	-0.008	0.699	0.591	0.103
		-0.051	0.299	0.074	-0.104	-0.261	-0.205	-0.002	0.005	0.627	0.530	0.142

Table 38 Correlation of static and dynamic variables with MLOC.

We can see that a method's lines of code (MLOC) correlates positively with the method's complexity, to its nested block depth, and to a lesser extent to its fan-out. Methods that are bigger tend to be more complex, are more likely to have nested block of depth in the code, and tend to call other methods more.

We also observed that bigger methods tend to not be access methods (with a negative correlation of -0.261 between MLOC and data access methods), and they tend to not be leaves (with a 0 fan-out). Based on those observations, we felt we could cut methods that have less than 5 lines of code and a complexity index of two or less with a low likelihood of cutting out important methods.

9.1.4 Data Access

In an initial iteration of this work, published in [47], we eliminated access methods by looking at the method names. We assumed that all methods that have “.get” and “.set” at the start of the method names were access methods. This is not always the case, as

demonstrated on the simple example used at the time (TConfig) [47]. In our current case studies, two of the “main” methods selected by the jUCMNav expert are named *.getModel*: “seg/jUCMNav/editors/resourceManagement/UrnModelManager/getModel” and “seg/jUCMNav/editors/UCMNavMultiPageEditor/getModel”. In the initial iteration of this work, both would be filtered out. To address this problem, we looked at different static and dynamic parameters of methods that could be used to select getters and setters more reliably. For example, not all getters and setters had as high fan-in values as expected.

System name	Number of methods	Dynamic							Static			
		UniqueFanIn	UniqueFanOut	EncounteredInTrace	AverageCallHierarchy	Leaf	TotalSegmentPresence	CpuUsageAverage	MLOC	McCabe	NBD	PAR
HelloWorld	2	-0.071	-0.204	-0.004	0.422	-0.180	0.100	-0.033	-0.536	-0.411	-0.679	-0.596
HoverHelp	6	0.000	-0.497	-0.325	0.346	1.000	0.062	-0.632	-0.442	-0.586	-0.598	1.000
Clipboard	9	-1.000	0.035	-0.227	0.125	-0.125	-0.395	0.673	-0.466	-0.316	-0.534	-0.395
DND	14	0.248	-0.194	-0.116	0.563	0.679	-0.069	-0.114	-0.341	-0.127	-0.301	0.000
TextEditor	14	0.113	-0.394	-0.135	0.613	0.548	-0.129	-0.118	-0.308	-0.166	-0.196	0.239
BrowserExample	15	-0.083	-0.278	-0.215	-0.200	0.447	-0.293	-0.201	-0.272	-0.321	-0.415	0.031
Address Book	16	0.098	-0.268	-0.318	-0.098	0.488	-0.429	-0.333	-0.341	-0.150	-0.335	0.293
JavaViewer	20	-0.078	-0.271	-0.168	0.296	0.420	-0.045	-0.106	-0.262	-0.195	-0.261	-0.118
ImageAnalyser	26	-0.047	-0.166	-0.116	0.010	0.312	-0.106	-0.059	-0.108	0.093	-0.134	-0.143
LayoutExample	35	-0.110	-0.371	-0.170	-0.222	0.500	-0.190	0.099	-0.298	-0.143	-0.315	-0.138
TraceToUCM on jhotdraw TPTP	66	0.015	-0.017	-0.034	-0.124	0.046	-0.073	-0.021	-0.039	-0.045	-0.015	-0.138
PaintExample	104	0.023	-0.266	-0.199	-0.261	0.219	-0.052	0.058	-0.261	-0.355	-0.309	-0.134
Graphics	213	-0.071	-0.204	-0.004	0.422	-0.180	0.100	-0.033	-0.536	-0.411	-0.679	-0.596
Control	293	-0.092	-0.216	-0.152	0.156	0.804	-0.144	0.033	-0.422	-0.289	-0.415	-0.156
jucmnav_TPTP_Nov26AfterExclu	305	0.135	-0.125	0.097	0.099	0.274	0.195	-0.040	-0.183	-0.197	-0.316	-0.094
Umple_compiling_BCMS TPTP	351	0.209	-0.218	0.006	0.129	0.579	0.026	-0.052	-0.059	-0.271	-0.408	-0.277
Uced TPTP_open_file_Nov12Af	458	0.022	-0.106	-0.058	0.064	0.204	0.054	0.032	-0.251	-0.209	-0.253	-0.254
JHotDraw	566	0.118	-0.260	0.069	0.230	0.249	0.047	0.080	-0.268	-0.256	-0.307	-0.238
		0.056	-0.194	-0.021	0.139	0.324	0.024	0.011	-0.261	-0.252	-0.349	-0.228

Table 39 Correlation of static and dynamic variables with data access methods.

Table 39 shows that the correlation between access methods and fan-in is 0.056. We also expected strong negative correlation between getters/setters and fan-out. But that is not the case. Clearly, getters and setters are being used for more than data access. Therefore we fine-tuned the algorithm used in [47] so that instead of throwing out all *.get* and *.set* methods, we only eliminate *.get* and *.set* methods that had 2 or fewer lines of code or a

complexity of 2 or less. If MLOC was greater than or equal to 3 or McCabe was greater than or equal to 3, we kept it.

This anti-filter has caught one of the getModel methods selected by the designer because it had MLOC greater than 2 but it has not caught the second one. The data table for jUCMNav (Table 40) shows that with a complexity index of five and 13 lines of code, UrnModelManager/getModel may not be a simple getter. It should be left and proceed to the utility detection step.

TraceItemName	UniqueFanIn	UniqueFanOut	DataAccessMethod	MLOC	McCabe	NBD
seg/jUCMNav/editors/actionContributors/ActionRegistryManager/createActions	0	14	0	403	20	2
seg/jUCMNav/actions/SelectionHelper/setInternals	67	0	0	245	98	6
seg/jUCMNav/actions/concerns/ManageConcernsAction/calculateEnabled	1	6	0	13	6	1
seg/jUCMNav/editors/resourceManagement/UrnModelManager/getModel	1	0	0	13	5	3
seg/jUCMNav/actions/ChangeStubTypeAction/calculateEnabled	1	3	0	13	5	1
seg/jUCMNav/model/commands/cutcopypaste/PasteCommand/getFirstPathNode	1	0	0	11	6	4
seg/jUCMNav/editors/UCMNavMultiPageEditor/getModel	5	0	1	1	1	1
com/collabnet/subversion/merge/Activator/getMergeResultsLocation	1	0	1	1	1	0
seg/jUCMNav/editpolicies/feedback/PathNodeNonResizableEditPolicy/getFigure	2	0	1	1	1	1
seg/jUCMNav/editpolicies/feedback/ConnectionFeedbackEditPolicy/getFigure	1	0	1	1	1	1
seg/jUCMNav/editors/MultiPageEditorLocation/getEditor	1	0	1	1	1	1

Table 40 Data table for methods in jUCMNav trace.

9.1.5 Possible Improvement of the Trace Filtering Step

More work needs to be done to fine-tune the filter so important methods are not cut. This is especially of concern for jUCMNav, where 159 of the 347 methods encountered in the sample execution trace are labelled .get and .set. A couple of static parameters that could

help make the filters better might be PAR (Number of Parameters) and NSF (Number of Attributes). We can see from Table 39 that there is a slightly negative correlation with PAR. This should be explored further. One way to do this would be to go through the code and manually find out which methods are truly just getters and setters, label them distinctly and then re-run the correlation test. That would give a better picture of the correlation of access methods and other parameters.

9.2. Validation of the Utility Detection Step

9.2.1 Which Algorithm Works Best

The validation of the utility detection step is covered in detail in Chapter 7 and Chapter 8. In this section, we give a brief summary of the validation technique used. We do this for sake of completeness.

To validate our utility selection algorithms, we created UCM diagrams using filtered traces by the eight algorithms. Figure 31 zooms in on the differences between the eight algorithms overviewed in Figure 30.

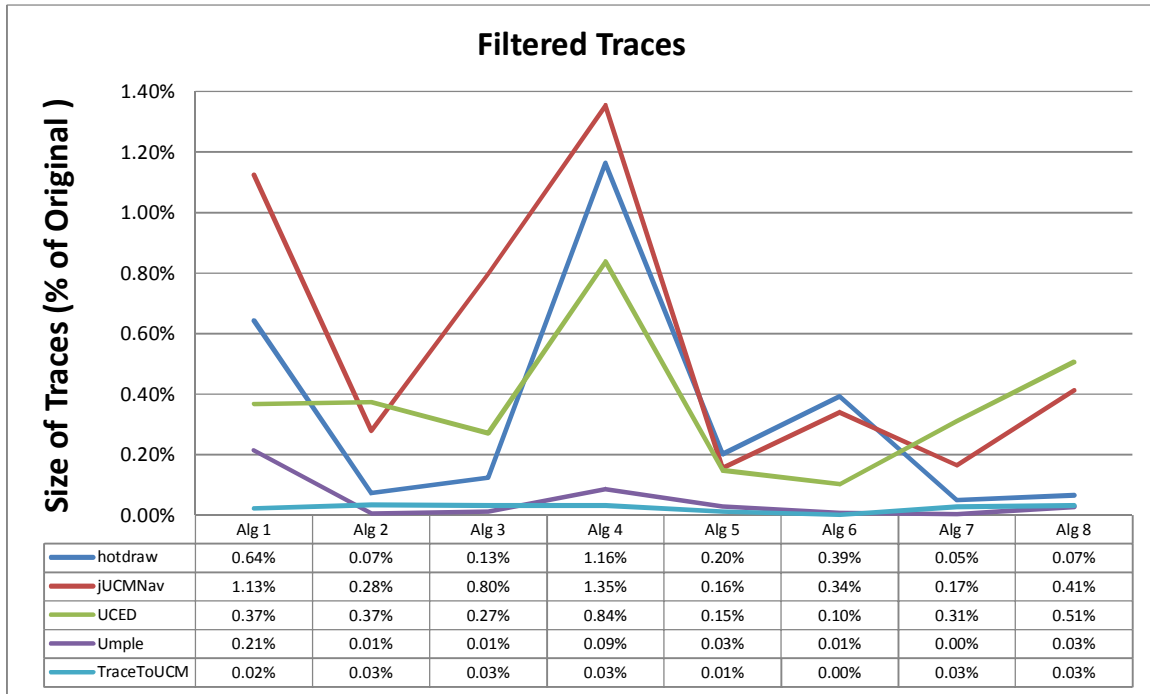


Figure 31. Size of filtered traces (algorithms only), in percentage of the original size.

We verified our results by showing the generated UCM map to the system designers or maintainers. In order to depict the proper behavioural method an algorithm needs to isolate the principal methods of a feature.

The questionnaires are given in Appendices B, C and D. We have two questions for each of the diagrams: 1) Does Algorithm X's diagram contain the main methods of the feature? and 2) Does the UCM diagram matches the mental image you have of the feature? The question asks for an answer on a five-point scale, which we converted to a numerical scale from -2 to +2 for analysis.

We added the results from all the designers for each software system, and the algorithm that scored the highest was considered the best algorithm for the particular system. Sections 7.3, 7.4, and 7.5 give the results and analysis in greater detail.

9.2.2 Precision and Recall

Another approach we have used to assess an algorithm was to test which one of the algorithms picks a list of main methods that are similar to the ones provided by the system's creators. In order to compare the performance of the eight algorithms independent of the visualization, we asked participants to provide a list of the main methods for the selected features. This list was considered the canonical (i.e., presumed correct) list and we evaluated each algorithm according to how good it was at picking out the methods from the list. The process and the results are explained in greater detail, using precision and recall analysis, in Section 7.7.1.

9.3. Validation of the Visualization Step

9.3.1 Validation Approach

In the first pass, the visualization for the smaller programs was done by the thesis author by inspecting the source code and going through the execution of the program using a debugger to see if the behaviour depicted in the UCM diagram was consistent with the behaviour of the program. For the three large programs (jUCMNav, Umple and UCed), validation of the visualization method used was done by asking the participants the following question:

Assuming the correct methods have been selected would you use UCM diagrams to document the behaviour of a feature in design documents?

Strongly disagree Disagree Neutral Agree Strongly agree

The question is Part II of the questionnaires given in Appendix B, C and D for jUCMNav, Umple and UCed respectively. Seven out of the eight participants answered that they agreed or strongly agreed, and none disagreed (see 0). Even though we had nine participants, we only counted eight because one of the participants worked both on Umple (uS4) and UCed (uCS1).

This was used to validate the visualization method independent of the method selection algorithms. A future improvement would be to give designers 3 different representations of the same reduced trace (UCM diagram, MSC diagram and a textual print out of the reduced trace) and ask them for their preference.

Participants	Part II
uS1	agree
uS2	agree
uS3	agree
uS4	neutral
jS1	agree
jS2	agree
jS3	agree
uCS1	neutral
uCS2	strongly agree

Table 41 Replies of participants for part II of the questionnaire.

9.3.2 Use of UCM for Visualization

The semi-formal UCM notation shows how things work in abstract terms. It can also manage complexity through the use of stubs and plugins (hierarchical decomposition). The UCM notation enables binding responsibilities, mapped to methods in our context, to architectural components (e.g., classes). This could be done at other level of abstractions as well. For example, if a more detailed representation is required, then responsibilities

could be mapped to statements. On a coarser level of abstraction, responsibilities could be mapped to classes involved (rather than their methods).

9.4. Limitations and Threats to Validity

There are some threats to the validity of our methodology. In this section, we look at the threats using a general threat framework proposed by Wohlin *et al.* [116], which is used by grouping the factors that limit our ability to draw valid conclusions from empirical experiments into four major classes: external, internal, construct and conclusion threats [90].

9.4.1 External Validity Threats

External validity is the ability to generalize the observed results to a larger population. Important potential threats include the following:

1. Participants were not representative. Use Case Maps are used widely around the University of Ottawa and therefore most of the graduate students were familiar with UCM. This is not the case for all software engineers in general. This may make our results difficult to generalize because the participants were more familiar with them than the average software engineer outside of the University of Ottawa community.
2. Programs were not representative. All of the software studies were open source Java systems. Although care was taken to use tools that will work with object-oriented programming languages in general, the methodology needs to be tested with non-open source and non-Java systems.

9.4.2 Internal Validity Threats

Internal validity refers to the ability to correctly infer connections between dependent and independent variables. It deals with whether the experiment design guarantees that the relationship observed between treatment and outcome is a causal relationship and not the result of a factor upon which the researcher has no control. In our case, there are four possible internal validity threats pertaining to the experiment design.

1. Software error: Although care was taken to debug and verify the TraceToUCM tool, it is not a production system and therefore could contain bugs that affect the results of the experiment.
2. The same person, namely the author, selected the programs to be studied, the algorithms and the subjects who were invited to participate. This possibly introduces bias. However, to partially bias issues, all the developers of the three systems under study were invited to participate, the small programs were selected from an open source repository (Standard Widget Toolkit examples [104]), and the protocol was reviewed by the university's research ethics board.
3. We used TPTP to collect an execution trace and CPU data in two different runs. Care was taken to make sure the runs are identical, however when dealing in milliseconds one can never guarantee it and therefore it would be best to do the experiment using a tool that can collect all the data in the same run. Furthermore, better integration between TPTP, Metrics 1.3.6, and would make the solution more user-friendly.
4. In the filtering step, useful data may get thrown out by the Small and Simple filter. A good example where this could occur would be a recursive function that is very small. We try to safeguard against this by keeping the methods with the highest average CPU usage no matter what its size, complexity or method name. In addition, the *inclusion input file* (labeled I6 in Figure 6) can be used to ensure important methods are not cut even if they score low by the filters and algorithms.

9.4.3 Construct Validity Threats

Construct validity refers to the ability of dependent variables in the experiment to capture the effect being measured. We designed and explored two types of case studies to validate whether behavioural maps of a feature can be extracted from execution traces by filtering out utilities. We tested the use of the behavioral maps for software comprehension (with three open source software applications) and for software documentation (with our own TraceToUCM tool).

The first type of case study was constructed incrementally. In the first iteration (published in [47]), we studied one system, TConfig, using one utility detection algorithm

and validated the result with its (only) designer. Using lessons learned from the first iteration, we designed the second iteration with five algorithms (four new ones and one from the first iteration) and also compared our algorithms with 3 algorithms found in the literature [31][46][118].

In the first iteration, trace processing and UCM diagram creation were manual. These can potentially add a lot of variability to the quality of the filtered trace and the UCM diagrams, which would interfere with the ability of the dependent variable to capture the effect being measured.

We were able to mitigate this risk by automating the trace processing and UCM diagram generation in the second iteration, making the process repeatable, less error-prone and more user-friendly and efficient.

However, we were not able to eliminate the risks entirely as the layout program that is used by jUCMNav, Graphviz, does not produce a UCM diagram that is free of overlap and easy to read. Manual intervention was necessary to move overlapping component names and intertwined paths. Manual intervention introduces variance for how the UCM diagrams are laid out and may have biased participant's answers. Automating the layout will mitigate this risk. Therefore, our dependent variable, understandability of the UCM diagram, may have been hampered by the layout tool.

9.4.4 Conclusion Validity Threats

Conclusion validity is the ability to draw conclusions from statistical tests. The main threat here is that our sample size for real systems is only three, with few designers available for each, which prevents statistical significance from being reached. To mitigate this threat, we collected and analysed traces from Standard Widget Toolkit (SWT) example code [86] and validated the data by inspecting the code. Work needs to be done using larger collections of software systems, where dozens of developers are available.

For the software documentation case study in Chapter 8, one subject did not notice that the UCM diagrams produced by Algorithms 1 & 4 were identical. Algorithms 1 & 4 produced filtered traces and UCM diagram that had only four methods. S4, S13, S14 and S16 rated them to be the same. S8 did not notice they were the same and rated Algorithm 4 second and did not pick Algorithm 1 to be one of the top three. Similarly Algo-

rithms 6 & 7 produced UCM diagrams that are identical, with nine methods. S13 and S16 rated them the same, but S5, S6, S8 and S14 did not notice that the traces were the same and rated them differently. We believe this is because the diagrams for Algorithms 6 & 7 are bigger. The fact that some participants did not spot that these two pairs of diagrams are duplicates is a threat to conclusion. Not all of the participants were as observant as expected.

9.5. Summary

In this chapter, we described the approach we used to validate the three stages of our work: filtering, utility detection and visualization. We then identified and discussed the major threats that may apply to the validity of our work.

The next chapter concludes the thesis by summarizing the main contributions, comparing our work with closely related approaches, and listing topics for further research.

Chapter 10. Conclusion

In this chapter, we summarize the thesis contributions, discuss the best performing algorithms, rate our work against other methodologies and highlight future research opportunities.

10.1. Thesis Contributions

To be successful, software maintenance requires effective program comprehension techniques. In this research, we have used a combination of static and dynamic analysis techniques to make program comprehension simpler, particularly the detection of utilities and the summarizing of traces. We have grouped the contributions made in this research into two groups: major and minor contributions. In this section, the contributions of this research are highlighted, together with their corresponding chapters.

10.1.1 Major Contributions

There are three main contribution of this work:

1. A **framework** that can be used to **compare different methodologies** that use trace analysis. This framework has a list of criteria that can be used to evaluate program comprehension and methodologies (Chapter 3). The criteria can be re-used by academic researchers for similar studies. The framework can easily be extended as new methodologies are published.
2. A **methodology and tool for analyzing static and dynamic parameters** of object-oriented software (Chapter 4). This includes:
 - a. An approach to reduce large amounts of data contained in execution traces and to acquire sufficient knowledge about the system under study from the trace (Chapter 4). The reduced trace can be saved as simple text, using UCM notation or using Message Sequence Charts (MSCs) generated from UCMs using jUCMNav.

- b. An approach to organize and collate the large amount of data contained in source code, execution traces and profiling data to understand the software by identifying crucial program artefacts. The approach can be used by software designers who want to characterize how methods are being used in the system they are working on by looking at one or all of the artefacts produced by the tool (Section 6.3). The methodology can also be used by academic researchers to compare new algorithms against each other. One can have all the static data for the whole system or a combination of the static data and dynamic data. The data is contained in a text file that can easily get imported into a spreadsheet and analyzed to characterize the software being studied.
 - c. New automated tool support, part of TraceToUCM (Chapter 6).
3. **An approach and a tool to visualize software execution traces.** This includes:
- a. A mapping of software execution elements to UCM elements, UCM being a standard scenario notation (Section 4.5). The mapping can be used by software designers to transform trace components to UCM diagrams. The threshold setting and ranking of methods are customizable. One can easily change the setting to produce the UCM diagrams from the different algorithms to have different thresholds.
 - b. A tool, part of TraceToUCM, that exploits the mapping and generates UCM diagrams automatically. The user can use the generated diagrams as is, or transform them to MSCs automatically with jUCMNav.

10.1.2 Minor Contributions

The following are three minor contributions that were made by our work:

- 1. An analysis of the validity of using both static and dynamic data to better understand large software systems (Chapter 2).
- 2. A literature review on studies that used static and dynamic data analysis for software comprehension (Chapter 3). We summarized and compared studies that adopt the strengths of the two data analysis types and minimize the effect of their weaknesses.

3. An improved understanding of the use of utility filtering for software comprehension. This is demonstrated through the development of effective algorithms for detecting utilities (Chapter 5).
4. Case studies used to illustrate and validate these algorithms for reverse engineering (Chapter 7).
5. Case studies used to illustrate and validate the use of the UCM diagrams as documentation (Chapter 8). Designers can use UCM diagrams to document their code in design documents. They can also use this feature to check for naming inconsistencies in their own code (Section 8.3).
6. A more precise definition and characterization of utility classes and utility methods (Chapter 7, Chapter 8 and Chapter 9).
7. A small, validated set of dynamic and static parameters that can be used to identify utilities: size of a method, complexity of a method, and to lesser extend nested block depth of a method. These can be reused by academic researchers to further characterize utilities (Chapter 7, Chapter 8 and Chapter 9).

10.2. Comparison with Similar Work

In this section, we use the evaluation criteria defined in Section 3.1 to evaluate our own work and compare it to the studies reviewed in Chapter 3. Table 42 shows a summary of our evaluation in Section 3.1, with an additional row (the last one) that covers our work. The possible scores for each criteria are (0) Not addressed, (1) Addressed with some limitations or (2) Criteria has been met fully.

Criteria used	Scalability	Scalability for Trace Size	Visualization	Level of Prior Knowledge Required	Validation	Usability	Intrusive data collection	Target Programming Language
Dynamic Fan-in and Fan-out Metrics for Program Comprehension [118]	0	0	0	0	1	0	0	1
Scenario-Driven Dynamic Analysis for Comprehending Large Software [99]	1	0	2	2	2	1	1	1
Integrating Static and Dynamic Analysis to Improve Comprehension [26]	1	0	2	2	1	1	1	0
Understanding the Behaviour of Java Programs [105][106]	1	1	2	1	2	1	0	2
The CONCEPT Project [95]	1	0	2	0	1	1	1	2
Mapping Program Features to Code [115]	1	0	0	2	2	1	1	2
Aiding Program Comprehension by Static and Dynamic Feature Analysis [35]	2	0	1	2	1	2	1	2
Recovering High-Level Views from Static and Dynamic Analysis [94]	0	1	2	1	1	1	1	2
Analysis of Execution Traces for Program Comprehension [46]	1	2	1	1	2	0	1	2
Automatic Generation of Abstract Views for Software Comprehension [31]	2	2	2	2	0	2	1	1
Solving the Feature Location Problem by Measuring the Component Modification Impact [96]	2	2	0	2	1	2	1	2
This Thesis Reverse Engineering BehaviorBehavioural Models by Filtering out Utilities from Execution Traces	2	2	1	2	1	1	1	2

Table 42 Comparison of thesis methodology with reviewed papers.

- **Scalability: (2)** Full marks for scalability because no matter the size of the target software, the user can focus the behavioural model extraction on one feature at a time.
- **Scalability for trace size: (2)** We give our methodology full marks because the solution can be used regardless of the input file size. jUCMNav supports several levels of nested stubs, and by using stubs we can visualize large traces. For extra-large trace files the processing could be done a few classes at a time by using the exclusion feature of the tool; the resulting visualization could then be viewed a few classes at a time.

- **Visualization: (1)** We give our methodology a score of 1 because the layout program that is used by jUCMNav was not able to automatically handle layout of the UCM diagrams. Manual intervention was necessary to move overlapping component names and intertwined paths. We did not give a 0 mark because there is a work-around. Someone trying to use our approach to understand a system could produce a large and messy diagram, he/she can then select an item to delete, and repeat the process as needed. The user can also add the selected method in a file called ‘exclusion’ so that the method is not included in subsequent runs.
- **Level of prior knowledge required: (2)** No prior knowledge of the code is required to use this method. The user only needs to know how to run the selected feature.
- **Validation: (1)** Validation of the UCM diagrams was done with participants who already have familiarity with UCM notations, which adds some bias.
- **Usability: (1)** We give our methodology 1 for essentially the same reason as the visualization criteria above. The technique is user friendly because all that the user needs to do is exercise the feature of interest, collect data, and run Trace-ToUCM, a process that is mostly automated. However the program used to layout the UCM diagram does not always produce a readable map. Human intervention is required to move overlapping and crisscrossing lines, which is time consuming. The bigger the map, the more manual intervention is required.
- **Intrusive data collection: (1)** The technique has an observer effect from the collection of CPU data and is therefore not suitable for time-critical systems.
- **Target programming language: (2)** The technique of comparing execution traces is a valid technique to use for all systems written with an object-oriented programming language.

Our methodology scores similarly with the best one we looked at, which is Dugerdil and Repond [31] method to automatically generate abstract views of software comprehension. Dugerdil and Repond’s work, discussed in Section 3.2.10, has much similarity to our work, in that the final result is a visualized behavioural model of the system. Unlike our approach, they include all classes in the system; they do not make the assumption that a

class might be an implementation-only (utility) class. This means they have less chance of losing sight of functional code that is housed in a utility function. On the other hand, they collect events between application classes, as opposed to methods, so they do not start with a detailed execution trace that includes all events like we do.

Eisenbarth, Koschke and Simon [35] present a technique, evaluated in Section 3.2.7, to map the system's externally visible behaviour to the relevant parts of the source code. They combine dynamic and static analysis to derive correspondences between features and code components, with a goal of reverse engineering a partial description of the software architecture. Their methodology scores high on usability criteria set in Chapter 3 because their methodology is more user-friendly. They score lower than us for validation because they did not specify how they validated the components that were identified were indeed the same ones that were used by the features.

Rohatgi et al. [96] present an approach, evaluated in Section 3.2.11, which uses an execution trace of a feature and a dependency graph of the software to rank classes according to their relevance to the feature. The authors extract distinct classes from the execution trace of a feature and rank them according to their relevance to the feature. The ranking is based on the impact of a class modification on the remaining parts of a system. In general, the approach presented by Rohatgi et al. scored similarly to the work in this thesis with two main differences: usability and visualization. For usability, their approach scores higher than the work in this thesis because their approach has minimal human intervention. However, their approach scores lower than ours on the other criterion because their result is not visualized.

In the next section, we look at opportunities for further research and improvement of our methodology.

10.3. Future Work

Research opportunities that can further improve our methodology and tool, as well as other research directions, can be summarized as follows:

1. This work focused on programs written in Java. There is much legacy code written in non-Java programming languages. The study needs to be repeated using software from different programming languages and paradigms.

2. A further characterization of utilities could be done. One could investigate how context-dependent utilities are used in different contexts. For the same piece of software is there one fixed set of utilities? Or do utilities change from one feature to another? Are there differences between utilities of a single trace (or feature), and utilities of many traces (or features)?
3. In order to support very large software systems, one can raise the level of abstraction from methods and classes to classes and packages. The default UCM component is generic and can be mapped to any software components, even files. Therefore, the UCM mapping could be customized to represent as much or as little detail as required. The approach presented in this thesis can be modified to implement options so that the UCM responsibilities can represent classes (or statements) as opposed to methods.
4. jUCMNav is used to draw the UCM diagrams automatically. The layout tool that jUCMNav builds on, namely Graphviz, has a lot of difficulties with producing human readable UCM diagrams with minimal overlaps. We had to do a lot of manual layout. There is a lot of opportunity to improve the layout algorithm, and remove some bias from the kind of studies done here.
5. The execution traces we studied had about 90% reduction in size by removing access methods, loops and small and simple methods (Table 43). Once the small and simple methods and access methods are eliminated, the tool can go ahead and draw the UCM diagram. The tool can be extended to use multiple levels of abstraction by using multiple levels of stubs to simplify the UCM diagram; the UCM notation and jUCMNav both support multiple levels of stubs. Figure 32 shows an example of multi-level UCM diagrams.

Table 43 shows the reduction in number of lines in execution trace files.

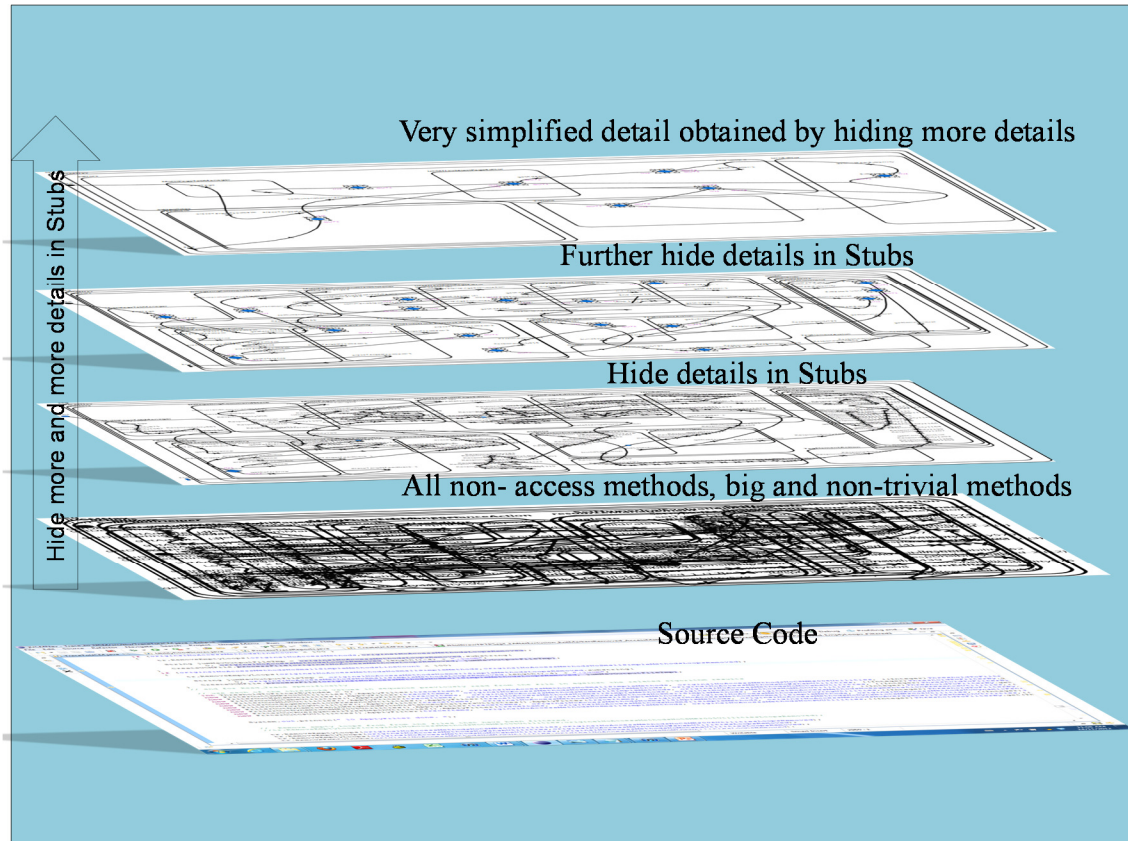


Figure 32. Multiple layers of UCM stubs to visualize execution traces.

Traces	JHotDraw	jUCMNav	UCED	Umple	Trace-ToUCM
Original	69455	31459	19543	202334 9	162483
W/o loops	12070	12411	7698	701019	7704
W/o exit markers	7103	6623	4164	363687	3449
W/o access	5300	2743	3088	80731	2399
W/o SmallSimple	2966	2391	1987	14183	27
Alg 1	447	354	72	4331	4
Alg 2	51	88	73	123	9
Alg 3	87	251	53	245	22
Alg 4	808	426	164	1765	4
Alg 5	140	49	29	607	23
Alg 6	273	107	20	172	9
Alg 7	35	52	61	75	9
Alg 8	46	130	99	577	22

Table 43 Number of lines in trace files.

As observable from this table, the greatest reduction in size comes by removing loops, access methods and small and simple methods. Currently we visualize if the number of lines in the execution trace is less than or equal to 100 after we eliminate the small and simple methods, the limit could be set 1000. The tool can be customized to have only a few items in each page before a stub is created. The process can be repeated until the diagram becomes less cluttered and easier to read. In addition to improving on the utility algorithms, this will make the approach more scalable.

6. The segmentation algorithm can be improved to produce a smoother transition between the numbers of segments that a trace is divided in. In the current segmentation algorithm transitions are not smooth. A trace of 999,999 lines will be divided in segments of 10,000 lines each, whereas one of 1,000,002 lines (only 3 more) will result in segments of 100,000 each. The algorithm can be improved to produce a smoother mathematical function involving logarithms. See Section 7.7.3 for more details.
7. The threats to validity, discussed in Section 9.4.4, could be addressed by running the studies again using more participants, more software systems and more than one trace from each software system.
8. Another improvement on the generalizability of the visualization/documentation case study would be to give the designers three different representations of the same reduced trace (UCM diagram, MSC diagram and a textual print out of the reduced trace) and ask them to rate them for understandability.
9. Finally, the current approach might benefit from a more specific handling of multi-threaded (or even distributed) applications. At the moment, elements in a trace might come from concurrent threads and hence lead to increased visualization complexity (as UCM responsibilities would alternate between components rather than being shown as concurrent sub-sequences connected with an AND-fork) to loops that are not correctly detected (because different iterations might have different orderings resulting from multiple interleaving threads). Improvements could be inspired from work done by Briand *et al.* [17] on the reverse engineering of behaviour from distributed Java applications.

10.4. Closing Remarks

The broad underlying objective of this thesis was to improve on the state of the art in program comprehension by using static and dynamic analysis. We believe that we have achieved our objective to a large extent. Beyond that, we hope that the ideas and tool-supported methodology presented in this thesis will encourage researchers to build on our work and use dynamic and static analysis techniques to further the field of software engineering in general and program comprehension in particular.

References

- [1] Aggarwal A. and Jalote P.: “Integrating Static and Dynamic Analysis for Detecting Vulnerabilities”. In *30th Annual International Computer Software and Applications Conference*, pp. 343-350, 2006.
- [2] Amyot D. and Mussbacher G.: “User Requirements Notation: The First Ten Years, The Next Ten Years”. Invited paper, *Journal of Software (JSW)*, Vol. 6, No. 5, Academy Publisher, pp. 747-768, May 2011.
- [3] Amyot D., Mussbacher G., Ghanavati S., and Kealey J.: “GRL Modeling and Analysis with jUCMNav”. In *5th International i* Workshop (iStar 2011)*, Trento, Italy, CEUR-WS, Vol-766, pp. 160-162, August 2011.
- [4] Amyot D., Mussbacher G., and Mansurov, N.: “Understanding Existing Software with Use Case Map Scenarios”. In *3rd SDL and MSC Workshop (SAM’02)*, Aberystwyth, U.K., June 2002. LNCS 2599, Springer, pp. 124-140, 2002.
- [5] Antoniol G. and Di Penta M.: “Library Miniaturization Using Static and Dynamic Information”, In *Proceedings of the International Conference on Software Maintenance*, p.235, 22-26, September 2003.
- [6] Antoniol G. and Guéhéneuc, Y.G.: “Feature identification: a novel approach and a case study”, In *Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on*, IEEE CS, pp. 357-366, 2005.
- [7] Antoniol G. and Guéhéneuc, Y.G.: “Feature identification: An epidemiological metaphor”. *Software Engineering, IEEE Transactions on*, 32(9), 627-641, 2006.
- [8] Antoniol G., Di Penta M., and Zazzara M.: “Understanding Web Applications through Dynamic Analysis”. In *Proceedings of the 12th International Workshop on Program Comprehension*, pp. 120-129, 2004.
- [9] Ball T.: “The Concept of Dynamic Analysis”. In *ESEC / SIGSOFT FSE*, ACM Press, pp. 216-234, 1999.
- [10] Ball T.J. and Eick S.G.: “Software Visualization in the Large”. *IEEE Computer* 29(4), pp. 33-43 1996.
- [11] Badreddin, O.: “Umple: A Model-Oriented Programming Language”. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, Volume 2, IEEE CS, pp. 337-338, 2010.
- [12] Badreddin O., Forward A., and Lethbridge T. C.: “Model Oriented Programming: An Empirical Study of Comprehension”. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, ACM Press, pp. 73-86, 2012.

- [13] Beck K. and Johnson R.E.: "Patterns generate architectures". In *Proceedings ECOOP'94*, LNCS 821, pp. 139–149, July 1994.
- [14] Blaine A. P., Baecker, R.M., and Small, I.S., "A Principled Taxonomy of Software Visualization". In *Journal of Visual Languages and Computing*, vol. 4, no. 3, pp. 211-266, 1993.
- [15] Bondi A.B.: "Characteristics of scalability and their impact on performance". In *Proceedings of the 2nd international workshop on Software and performance*, 2002, pp. 195-203, 2002.
- [16] Brant J., Foote B., Johnson R.E., and Roberts D.: "Wrappers to the Rescue". In *Proceedings ECOOP'98*, LNCS 1445, pp. 396–417, 1998.
- [17] Briand L.C., Labiche Y., and Leduc J.: "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". *IEEE Trans. Software Eng.*, 32(9), pp. 642-663, 2006.
- [18] Buhr R.J.A.: "Use Case Maps as Architectural Entities for Complex Systems". In *IEEE Transactions on Software Engineering*, Vol. 24, No. 12, pp. 1131-1155, December 1998.
- [19] Burtscher M., Ganusov I., Jackson S. J., Ke J., Ratanaworabhan P., and Sam N. B., "The VPC Trace-Compression Algorithms". *IEEE Transactions on Computers (IEEE-TC'05)*, November 2005.
- [20] Capozucca A., Cheng B.H.C., Georg G., Guelfi N., Istioan P., Mussbacher G.: "Requirements Definition Document for a Software Product Line of Car Crash Management Systems", <http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf> Last accessed on April 28, 2013.
- [21] Canfora G., Mancini L., and Tortorella M. A.: "Workbench for Program Comprehension During Software Maintenance". In *Proc. 4th Int'l Workshop on Program Comprehension*, IEEE CS, pp. 30-39, 1996.
- [22] Chikofsky E.J.: "Reverse Engineering and Design Recovery: A taxonomy", *IEEE Software*, pp. 13-17, January 1990.
- [23] Corbi T.: "Program Understanding: Challenge for the 1990s". In *IBM Systems Journal*, 28(2), pp. 294-306, 1989.
- [24] Cornelissen B., Zaidman A., van Deursen A., Moonen L., and Koschke R.: "A systematic survey of program comprehension through dynamic analysis". *Software Engineering, IEEE Transactions on*, 35(5), 684-702, 2009.
- [25] Demeyer S., Ducasse S. and Nierstrasz O.: *Object-Oriented Reengineering Patterns*, San Francisco, CA: Morgan Kaufmann, 2003.
- [26] Di Lucca G. A. and Di Penta M.: "Integrating Static and Dynamic Analysis to Improve the Comprehension of Existing Web Applications". In *WSE*, pp. 87-94, 2005.

- [27] Di Lucca G.A., Fasolino A.R. Tramontana F.: “Reverse Engineering Web Application: the WARE approach”. In *Journal of Software Maintenance and Evolution: Research and Practice*, Volume 16, Issue 1-2, January-April 2004.
- [28] Dit B., Revelle M., Gethers M., and Poshyvanyk D.: “Feature location in source code: a taxonomy and survey.” In *Journal of Software: Evolution and Process*, 25(1), 53-95, 2013.
- [29] Ducasse S. and Pollet D.: “Software architecture reconstruction: A process-oriented taxonomy”. *IEEE Trans. Softw. Eng.* Vol 35: pp. 573–591, 2009
- [30] Dugerdil Ph. and Jossi S. “Empirical Assessment of Execution Trace Segmentation in Reverse-Engineering”. In *Proc. Third Int. Conf. on Software and Data Technologies (ICSOFT 2008)*, Volume SE/MUSE/GSDCA, pp. 20–27, July 2008.
- [31] Dugerdil P. and Repond, J “Automatic Generation of Abstract Views for Legacy Software Comprehension”. In *3rd Indian / ACM Conference on Software Engineering (ISEC'2010)*, 2010
- [32] Eaddy M., Aho A. V., Antoniol G., and Guéhéneuc Y.G.: “Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis”. In *Program Comprehension (ICPC 2008), The 16th IEEE International Conference on*, pp. 53-62, IEEE CS, 2008.
- [33] Eclipse, <http://www.eclipse.org/>
- [34] Eclipse Test & Performance Tools Platform Project, <http://www.eclipse.org/tptp/>
- [35] Eisenbarth T., Koschke R., and Simon D.: “Aiding Program Comprehension by Static and Dynamic Feature Analysis”. In *17th IEEE International Conference on Software Maintenance*, Florence, Italy, pp. 602–611 November 2001.
- [36] Eisenbarth T., Rainer K., and Daniel S.: “Locating features in source code”. *Software Engineering, IEEE Transactions*, 29, no. 3, pp. 210-224, 2003.
- [37] Eisenstadt M., Price B. A., and Domingue J.: “Software Visualization as a Pedagogical Tool”. *Instructional Science*, 1993, 21: pp. 335-365, 1993.
- [38] Elberzhager F., Münch J., and Nha V. T. N.: “A systematic mapping study on the combination of static and dynamic quality assurance techniques”. *Information and Software Technology*, 54(1), pp. 1-15, 2012.
- [39] Ernest M.: “Static and Dynamic Analysis: Synergy and Duality”. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Washington, USA, 2004.
- [40] Exton C.: “Understanding context and its role in software engineering tasks or Human Software Interaction”. Online presentation available at: http://informatique.umons.ac.be/ftp_infofs/2011/Exton-EvolUMons2011.pdf
- [41] Forward A, Lethbridge T.C., and Brestovansky D.: “Improving Program Comprehension by Enhancing Program Constructs: An Analysis of the Umple language”. In *International Conference on Program Comprehension (ICPC)*, pp. 311-312. IEEE CS, 2009.

- [42] Gall H., Jazayeri M., Klosch R., Lugmayr W., and Trausmuth G.: "Architecture Recovery in ARES". In *Proc. of the 2nd International Software Architecture Workshop (ISAW-2)*, San Francisco, 1996.
- [43] Greevy O. and Ducasse S.: "Correlating features and code using a compact two-sided trace analysis approach". In *Software Maintenance and Reengineering (CSMR 2005), Ninth European Conference on*, pp. 314-323. IEEE CS, 2005.
- [44] Green T., Blandford A., Church L., Roast C. and Clarke S.: "Cognitive Dimensions: Achievements, New Directions, and Open Questions". *Journal of Visual Languages and Computing* 17, pp. 328-365, 2006.
- [45] Haldar V., Chandra D., and Franz M.: "Dynamic Taint Propagation for Java". In *Twenty-First Annual Computer Security Applications Conference (ACSAC)*, pp. 303-311. IEEE CS, 2005.
- [46] Hamou-Lhadj A.: *Techniques to Simplify the Analysis of Execution Traces for Program Comprehension*. Ph.D. Dissertation, School of Information Technology and Engineering (SITE), University of Ottawa, Canada, 2005
<http://www.site.uottawa.ca/~tcl/gradtheses/ahamou/AbdelwahabHamouLhadjPhDThesis.pdf>.
- [47] Hamou-Lhadj A., Braun E., Amyot D. and Lethbridge T.: "Recovering Behavioural Design Models from Execution Traces". In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 112-121. IEEE CS, 2005.
- [48] Hamou-Lhadj A. and Lethbridge T.: "The Compact Trace Format". In *1st Int. Workshop on Meta-models and Schemas for Reverse Engineering*, Co-located with 10th WCRE, Victoria, BC, November 2003.
- [49] Hamou-Lhadj A. and Lethbridge T.C.: "A Metamodel for the Compact but Loss-less Exchange of Execution Traces". *Software and Systems Modeling*, pp. 77-98. Springer, 2010.
- [50] Hamou-Lhadj A. and Lethbridge T.C.: "A Survey of Trace Exploration Tools and Techniques". In *CASCON 2004*, pp. 42-55. ACM, 2004.
- [51] Hamou-Lhadj, A. and Lethbridge, T.C.: "Understanding the Complexity Embedded in Large Routine Call Traces with a Focus on Program Comprehension Tasks". *IET Software*, 4 (2), pp. 161-177, 2010.
- [52] Hansen W. J.: "Measurement of Program Complexity by the Pair: (Cyclomatic Number, Operator Count)". *ACM SIGPLAN Notices* Vol. 13, Issue 3, pp. 29 - 33, March 1978.
- [53] Henderson-Sellers B.: *Object-Oriented Metrics: Measures of Complexity*, Upper Saddle River, N.J.: Prentice Hall, 1996.
- [54] Hendrix T.D., Cross II J.H, Maghsoodloo S., and McKinney M.L.: "Do Visualizations Improve Program Comprehensibility? Experiments With Control Structure Diagrams for Java". *ACM Proceedings of SIGCSE*, pp. 382-386. ACM, 2000.
- [55] Henry S. and Kafura K.: "Software Structure Metrics Based on Information Flow". *IEEE Transactions on Software Engineering*, 7(5), pp. 510-518, 1981.

- [56] Huang Y., Yu F., Hang C., Tsai C., Lee, D. and Kuo, S.: “Securing Web Application Code by Static Analysis and Runtime Protection”. In *Proc. of the 13th international conference on World Wide Web*, May 17-20, pp. 40-52. ACM, 2004.
- [57] ISO/IEC 14764:2006 *Standard for Software Engineering—Software Life Cycle Processes—Maintenance*. Second edition, September 2006.
- [58] ITU-T, Recommendation Z.120 (02/11), *Message Sequence Chart (MSC)*. International Telecommunication Union, Geneva, Switzerland, February 2011.
- [59] ITU-T, Recommendation Z.151 (10/12) *User Requirements Notation – Language Definition*. International Telecommunication Union, Geneva, Switzerland, October 2012.
- [60] JHotDraw, <http://www.jhotdraw.org/>
- [61] Johnson S.: *Lint, a C program checker*. Computer Science Technical Report 65, Bell Laboratories, December 1977.
- [62] jUCMNav 5.4.0,
<http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/WebHome>
- [63] Kealey, J.: *Enhanced Use Case Map Analysis and Transformation Tooling*. Master’s thesis, SITE, University of Ottawa, Canada, November 2007.
- [64] Kemerer C. F.: “Software Complexity and Software Maintenance: A Survey of Empirical Research”, *Annals of Software Engineering*, 1(1), pp. 1–22, 1995.
- [65] Keromytis A. D., Salvatore J. S., Junfeng Y., Angelos S., Anup G., Dawson E., Marc D., Matthew E., and Darrell K.: “The MINESTRONE Architecture Combining Static and Dynamic Analysis Techniques for Software Security”. In *SysSec Workshop (SysSec)*, pp. 53-56. IEEE CS, 2011.
- [66] Koenemann J., and Robertson S. P.: “Expert problem solving strategies for program comprehension”. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 125-130. ACM, 1991.
- [67] Koschke R., and Quante J.: “On dynamic feature location”. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, pp. 86-95. ACM, 2005.
- [68] Koskimies K., Mannisto T., Systä T. and Tuomi J.: “Automated Support for Modeling OO Software”. *IEEE Software*, 15(1), pp. 87–94, January/February 1998.
- [69] Kreaseck B., Ramos L, Easterday S, Strout M, and Hovland P.: “Hybrid Static/Dynamic Activity Analysis”. In *Proceedings of the 3rd International Workshop on Automatic Differentiation Tools and Applications (ADTA '04)*, Reading, England, 2006.
- [70] Krogmann K.: “Reconstruction of software component architectures and behaviour models using static and dynamic analysis”. In *Vol. 4. KIT Scientific Publishing*, 2012.

- [71] Lee H. B. and Zorn B. G.: “BIT: A tool for Instrumenting Java Bytecodes”. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, pp. 73-82, 1997.
- [72] Lee S., Cho S., and Heo S.: “SAAT: Reverse Engineering for Performance Analysis”. In *Second International Workshop on Dynamic Analysis (WODA 2004) WIOS Workshop - 26th International Conference on Software Engineering*, pp. 40–47, 2004.
- [73] Lehman M.M.: Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 9, pp. 1060-1076, 1980.
- [74] Lethbridge T.C., Forward A., and Badreddin O.: “Umlification: Refactoring to Incrementally Add Abstraction to a Program”. In *Working Conference on Reverse Engineering*, pp. 220-224. IEEE CS, 2010.
- [75] Lientz B. P. and Swanson B. E.: *Software Maintenance Management*. Addison Wesley, 1980.
- [76] Linos P., Chen Z., Berrier S., and O'Rourke B.: “A Tool for Understanding Multi-Language Program Dependencies”. In *Proc. IEEE 11th Int'l Workshop Program Comprehension (IWPC 03)*, pp. 64-72. IEEE CS, 2003.
- [77] Livshits V.B. and Lam M.S.: “Finding Security Vulnerabilities in Java Applications with Static Analysis”, In *Usenix Security Symposium*, Aug. 2005.
- [78] Malton A. and Pahelvan A.: “Enhancing Static Architectural Design Recovery by Lightweight Dynamic Analysis (Position Paper)”. In *PCODA: Program Comprehension through Dynamic Analysis*, pp. 54-57, 2005.
- [79] Marin M., Van Deursen A., Moonen L.: “Identifying Aspects Using Fan-In Analysis”. In *Working Conference on Reverse Engineering*, pp. 132-141. IEEE, 2004
- [80] Mayrhauser V. and Vans A.M.: “Program Understanding Behaviour During Adaptation of Large Scale Software”. In *Proc. 6th Int'l Workshop Program Comprehension (IWPC 98)*, pp.164-172. IEEE CS, 1998.
- [81] McCabe T.J.: “A Complexity Measure Software Engineering”, *IEEE Transactions*. Volume: SE-2, Issue: 4, December 1976.
- [82] Metrics 1.3.6, <http://metrics.sourceforge.net/>
- [83] Miga A., Amyot D., Bordeleau F., Cameron C., and Woodside M.: “Deriving Message Sequence Charts from Use Case Maps Scenario Specifications”. In *10th SDL Forum (SDL'01)*, LNCS 2078, pp.268-287. Springer, 2001.
- [84] Moody D.L.: “The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering”, *IEEE Transactions on Software Engineering*, 35, pp. 756-779, 2009.
- [85] Müller H. A., Jahnke J.H., Smith D.B., Storey M., Tilley S.R. and Wong K.: “Reverse Engineering: A Roadmap”. In *Proceedings of the Conference on The Future of Software Engineering*, pp. 47-60. ACM, 2000.

- [86] Müller H. A., Wong K., and Tilley S.: “Understanding Software Systems Using Reverse Engineering Technology”. In *62nd ACFAS*, Montréal, Canada, 1994.
- [87] Mussbacher G., Ghanavati S., and Amyot D.: “Modeling and Analysis of URN Goals and Scenarios with jUCMNav”. In *Proc. 17th IEEE Int. Requirements Engineering Conference*, pp. 383-384. IEEE CS, 2009
- [88] Nosek J.T. and Prashant P.: “Software Maintenance Management: The Change in the Last Decade”, *Journal of Software Maintenance Research and Practice*, 2(3), pp. 57-174. John Wiley & Sons, 1990.
- [89] Parnas D.L.: “Software Aging”. In *Proc. of the 16th Intl. Conf. on Software Engineering (ICSE-16)*, pp. 279-287. IEEE CS, 1994.
- [90] Patel C., Hamou-Lhadj A. and Rilling J.: “Software Clustering Using Dynamic Analysis and Static Dependencies”. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pp. 27-36. IEEE CS, 2009.
- [91] Perry D.E., Porter A.A., and Votta L.G.: “Empirical studies of software engineering: a roadmap”. In *Proceedings of the Conference on The Future of Software Engineering*, pp. 345-355. ACM, 2000.
- [92] Price B.A., Baecker R.M., and Small I.S.: “A Principled Taxonomy of Software Visualization”, *Journal of Visual Languages and Computing*, 4(3), pp. 211-266, 1993.
- [93] Rajlich V. and Wilde N.: “The Role of Concepts in Program Comprehension”. In *Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02)*, pp. 271-278. ACM, 2002.
- [94] Richner T. and Ducasse S.: “Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information”. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, Oxford, pp 13-22. IEEE CS, 1999.
- [95] Rilling J., Seffah A., and Bouthlier C.: “The CONCEPT Project – Applying Source Code Analysis to Reduce Information Complexity of Static and Dynamic Visualization Techniques”. In *Proceedings of the First International Workshop on Visualizing Software*, pp. 90-99. IEEE CS, 2002.
- [96] Rohatgi A., Hamou-Lhadj A., and Rilling J.: “Approach for solving the feature location problem by measuring the component modification impact.” *IET Software*, 3(4), pp. 292-311, 2009.
- [97] Rohatgi A., Hamou-Lhadj A., and Rilling J.: “An approach for mapping features to code based on static and dynamic analysis”. In *Program Comprehension (ICPC 2008), 16th IEEE International Conference on*, pp. 236-241. IEEE, 2008.
- [98] Rohatgi A., Hamou-Lhadj A., and Rilling J.: “Feature location based on impact analysis”. In *Proc. of 11th IASTED Int. Conf. on Software Engineering and Applications*, pp. 517-522. Acta Press, 2007.
- [99] Salah M., Mancoridis S., Antoniol G., and Di Penta M.: “Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems”. In *Proc. of the*

- 10th European Conference on Software Maintenance and Reengineering*, pp. 71-80. IEEE CS, 2006.
- [100] Salomon D.: *Data Compression: The Complete Reference*, Springer, 2nd edition, ISBN: 0-387-95045-1, 2000.
 - [101] Schmerl B. R., Aldrich J., Garlan D., Kazman R., and Yan H.: “Discovering architectures from running systems”. *IEEE Trans. Software Eng.*, 32(7):454–466, 2006.
 - [102] Singer, J.: *Static program analysis based on virtual register renaming*. Technical Report UCAM-CL-TR-660, Computer Laboratory, University of Cambridge, UK, 2006.
 - [103] Somé S.: “An Environment for Use Cases Based Requirements Engineering”. In *12th IEEE Int. Requirements Engineering Conf.*, pp. 364-365. IEEE CS, 2004.
 - [104] SWT Examples, <http://www.eclipse.org/swt/examples.php>
 - [105] Systä T., Yu P., and Muller H.: “Analyzing Java Software by Combining Metrics and Program Visualization”. In *Proc. 4th European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pp. 199-209. IEEE CS, 2000.
 - [106] Systä T.: “Understanding the Behaviour of Java Programs”. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pp. 214-223. IEEE CS, 2000.
 - [107] Sun Microsystems Inc.: *Java(tm) 2 SDK, standard edition 1.2*. <http://java.sun.com/products/jdk/1.2/>, 2000.
 - [108] Tilley S.: *A Reverse-Engineering Environment Framework*, CMU/SEI-98-TR-005, Carnegie-Mellon University, USA, 1998.
 - [109] Tilley S. R., Paul S., and Smith D. B.: “Towards a Framework for Program Comprehension”. In *Proceedings of the 4th Workshop on Program Comprehension*, pp. 19- 29. IEEE CS, 1996.
 - [110] The Umple Language <http://cruise.site.uottawa.ca/umple/>.
 - [111] Use Case Editor (UCEd)
http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html.
 - [112] Walkinshaw N., Roper M., and Wood M.: “Feature location and extraction using landmarks and barriers”. In *Software Maintenance (ICSM 2007), IEEE International Conference on*, pp. 54-63. IEEE CS, 2007.
 - [113] Walker R.J, Murphy G. C, Freeman-Benson B.N., Wright D., Swanson D., and Isaak J.: “Visualizing dynamic software system information through high-level models”. In *Proc. 13th Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pp. 271-283. ACM, 1998.
 - [114] Wilde N. and Huitt R.: “Maintenance Support for Object-Oriented Programs”. *Transactions on Software Engineering*, Vol.18, No.12, pp. 1038-1044, December 1992.

- [115] Wilde N. and Scully M.C.: “Software Reconnaissance: Mapping program features to code”. *Journal of Software Maintenance: Research and Practice*, John Wiley & Sons, pp. 49-62, January-February 1995.
- [116] Williams A.: *TConfig Java Test Tool*. Available: <http://www.site.uottawa.ca/~awilliam>. Last accessed on April 16, 2009.
- [117] Wohlin C., Runeson P., Host M., Ohlsson M.C., Regnell B., Wesslen A.: *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 2000.
- [118] Yuying W., Qingshan L., Ping C. and Chunde R.: “Dynamic Fan-in and Fan out Metrics for Program Comprehension”. In *PCODA: Program Comprehension through Dynamic Analysis*, pp. 38-42, 2005.
- [119] Zhang R, Shiqiu H., Zhengwei Q., and Haibin G.: “Combining Static and Dynamic Analysis to Discover Software Vulnerabilities”. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pp. 175-181. IEEE CS, 2011.

Appendix A: Questionnaire

Recovering Behavioural Design Models from Execution Traces by using Utilities

I, Edna Braun, am conducting research into the utilityhood and reusability of classes. I am a PhD student in the School of Information Technology and Engineering (SITE) at the University of Ottawa. I am supervised by Dr. Daniel Amyot and Dr. Timothy Lethbridge. I am requesting that you answer some simple questions about the _____ (name of software system), which I understand you help develop.

To complete this survey should take between 10 and 30 minutes. The questionnaire, once filled out will be kept confidential, and your name will not be associated with any of the research results. Only aggregate data will be used. The raw data will be kept by the supervisor for 7-10 years in case of an audit.

Any information requests or complaints about the ethical conduct of the project may be addressed to The Protocol Officer for Ethics in Research, University of Ottawa, Tabaret Hall, 550 Cumberland Street, Room 159, Ottawa, ON K1N 6N5, tel.: 613-562-5841, email: ethics@uottawa.ca.

You may withdraw from the project at any time. If you do, your data will not be used. If you are a student, whether or not you participate in the study will have no effect on your grades or other academic evaluation.

If you agree to participate, please sign below:

I _____ consent to participate in the above research study conducted by Edna Braun.
Signature: _____

Part I

For the purpose of this work please use the following definition of *utility*:

A utility is any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.

In order to reverse engineer the Behavioural Model of UCED by filtering out utility methods, we have run the UCED application “open a project” and opened a project shown below. The feature uses the basic feature of UCED and collected trace files. The utility methods have been filtered out using eight different algorithms. The filtered trace files (containing the non-utility/main methods) are used to draw the Use Case Maps below.

Please take a look at the following 8 UCM diagrams and answer the questions below each diagram by highlighting or underlining your answer.

Part II

Assuming the correct methods had been selected would you use UCM diagrams to document the behaviour of a feature in your design documents?

Strongly disagree Disagree Neutral Agree Strongly agree

Appendix B: Behavioural Maps for jUCMNav

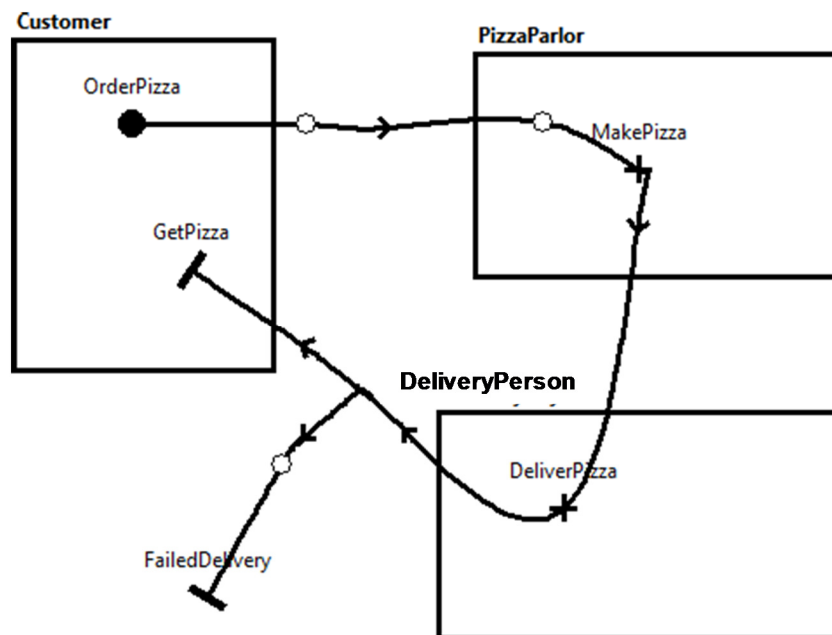
This Appendix contains the behavioural maps used in the experiment for the “Open a file feature” feature of the jUCMNav- System

Part I

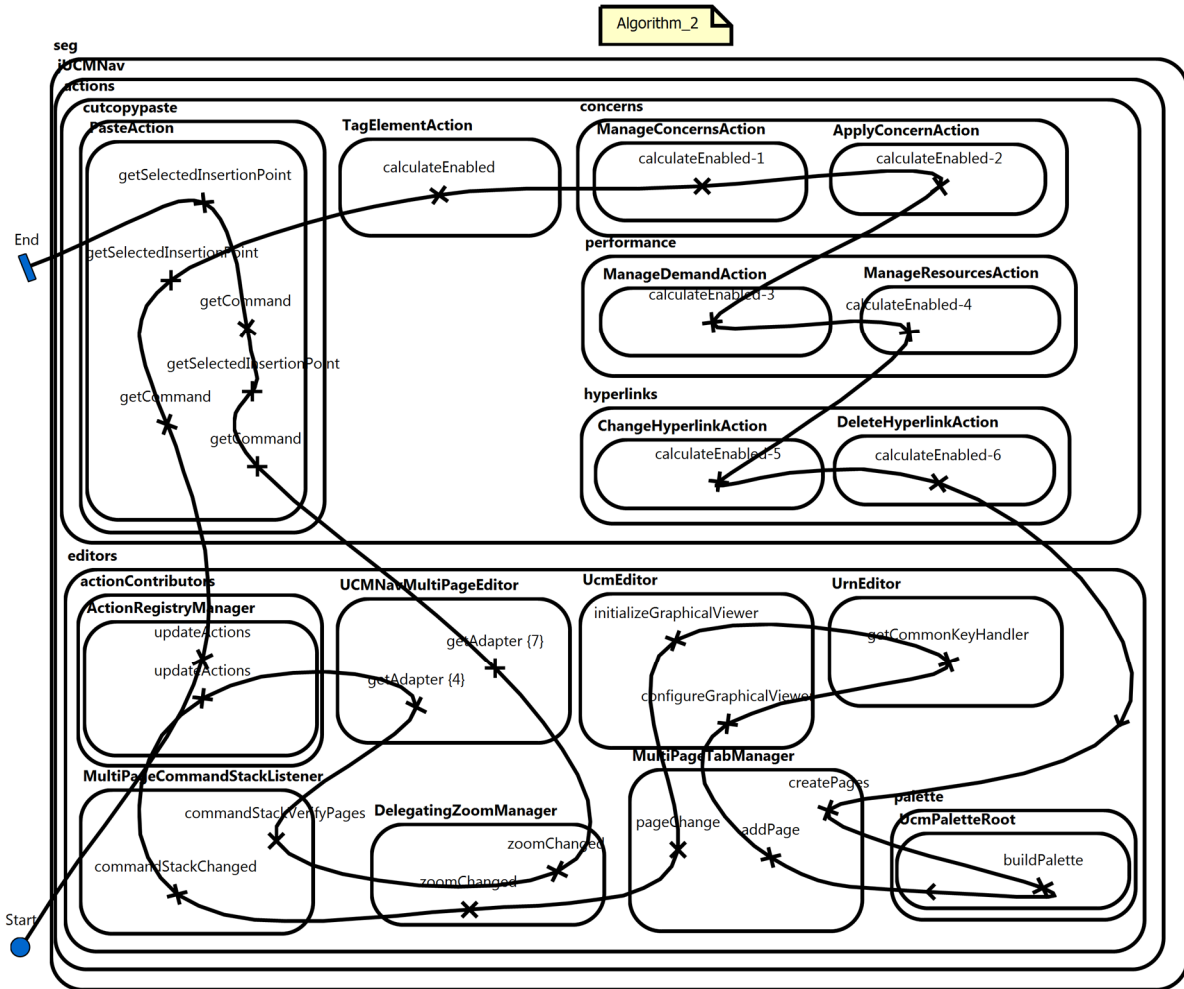
For the purpose of this work please use the following definition of *utility*:

A utility is any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.

In order to reverse engineer the behavioural model of JUCMNAV by filtering out utility methods, we have exercised the “open and load a UCM file” feature and collected tracefiles. The utility methods have been filtered out using eight different algorithms. The filtered trace files (containing the non-utility/main methods) are used to draw the eight Use Case Maps below.



Please take a look at the following 8 UCM diagrams and answer the questions below each diagram by highlighting or underlining your answer.

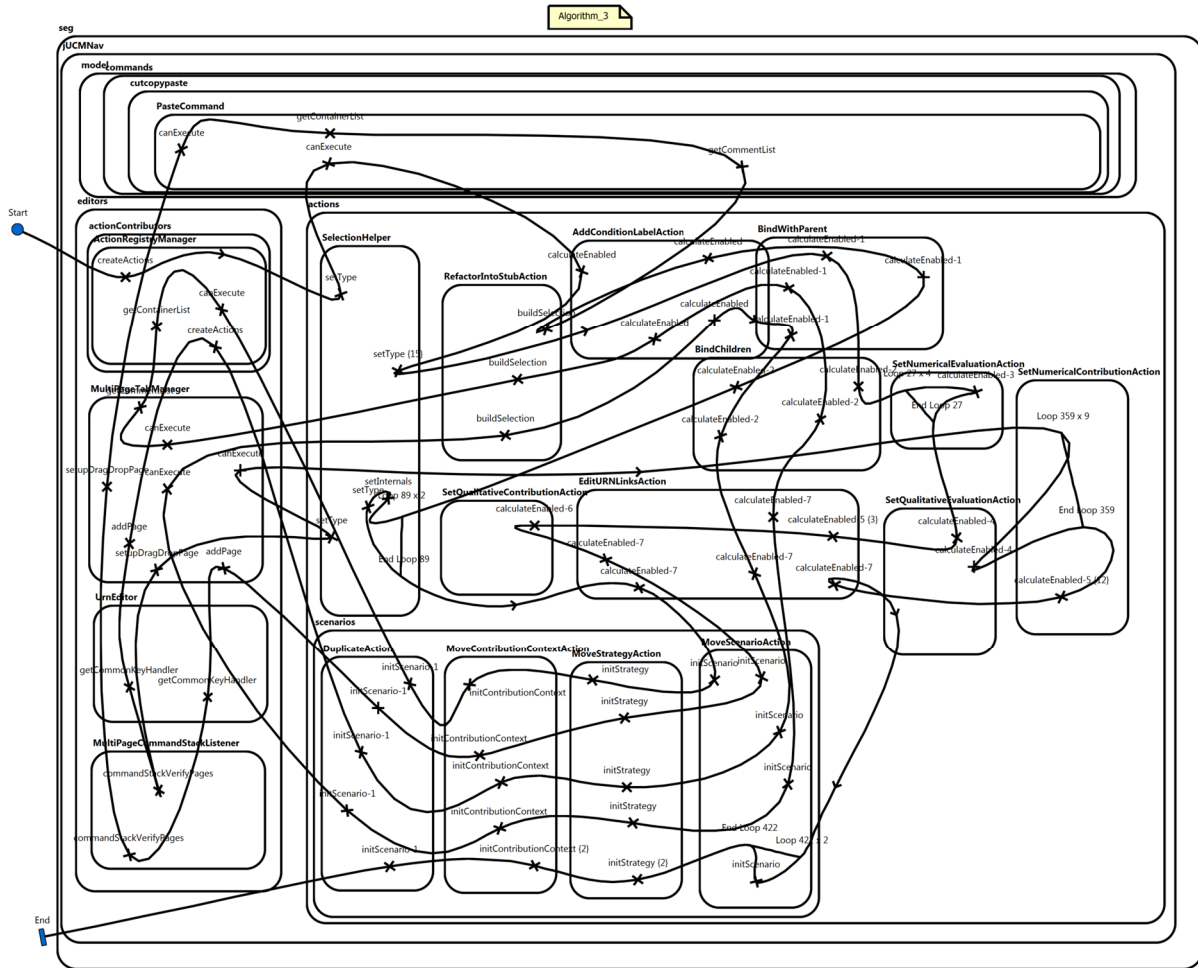


A. Algorithm 2 diagram contains the main methods used in JUCMNAV code for “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the JUCMNAV “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

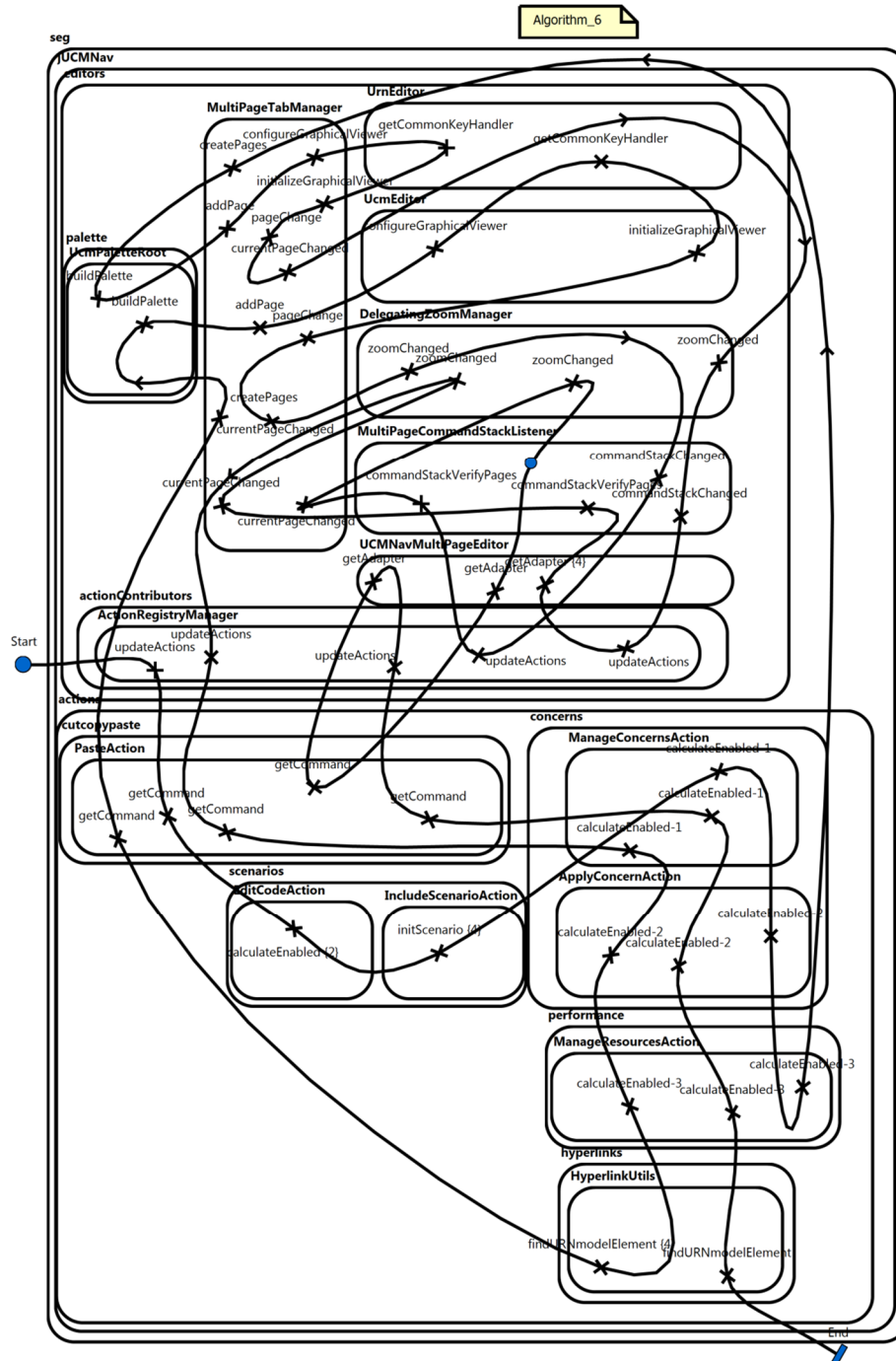


A. Algorithm 3 diagram contains the main methods used in JUCMNav code for “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the JUCMNav “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

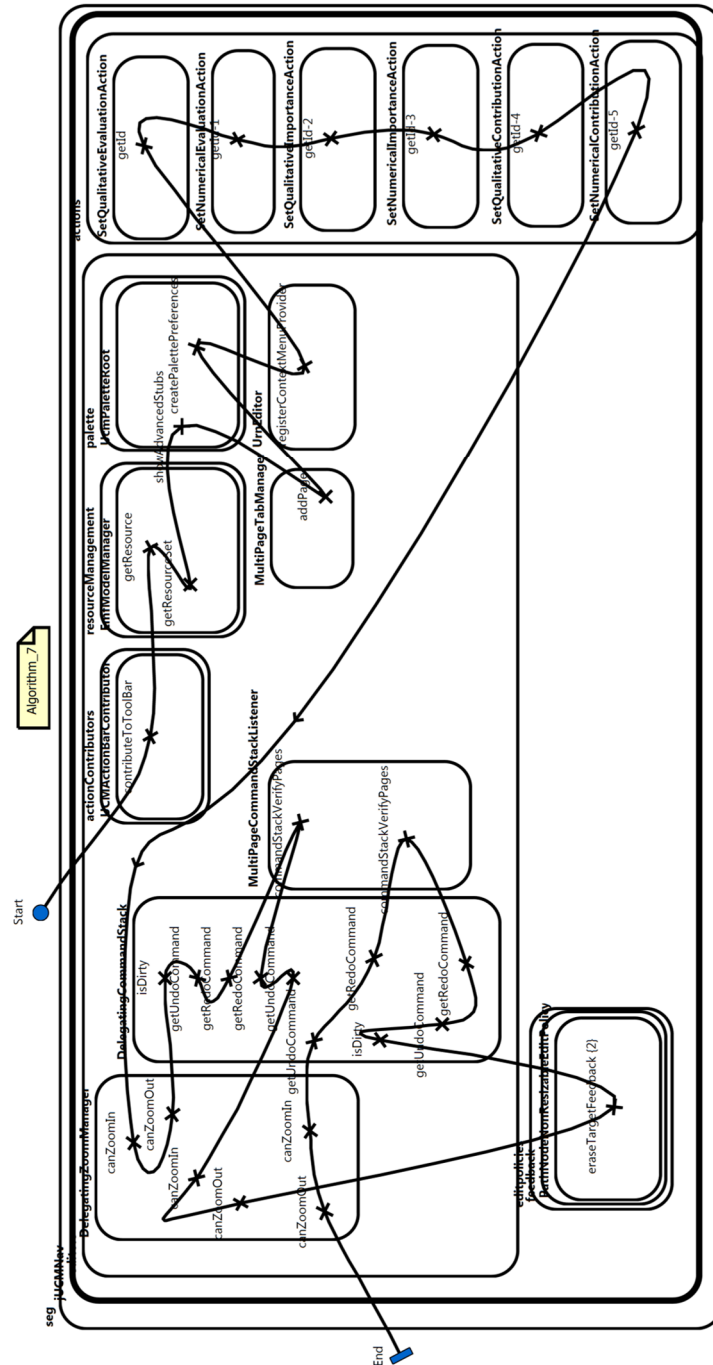


A. Algorithm 6 diagram contains the main methods used in JUCMNAV code for “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the JUCMNAV “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

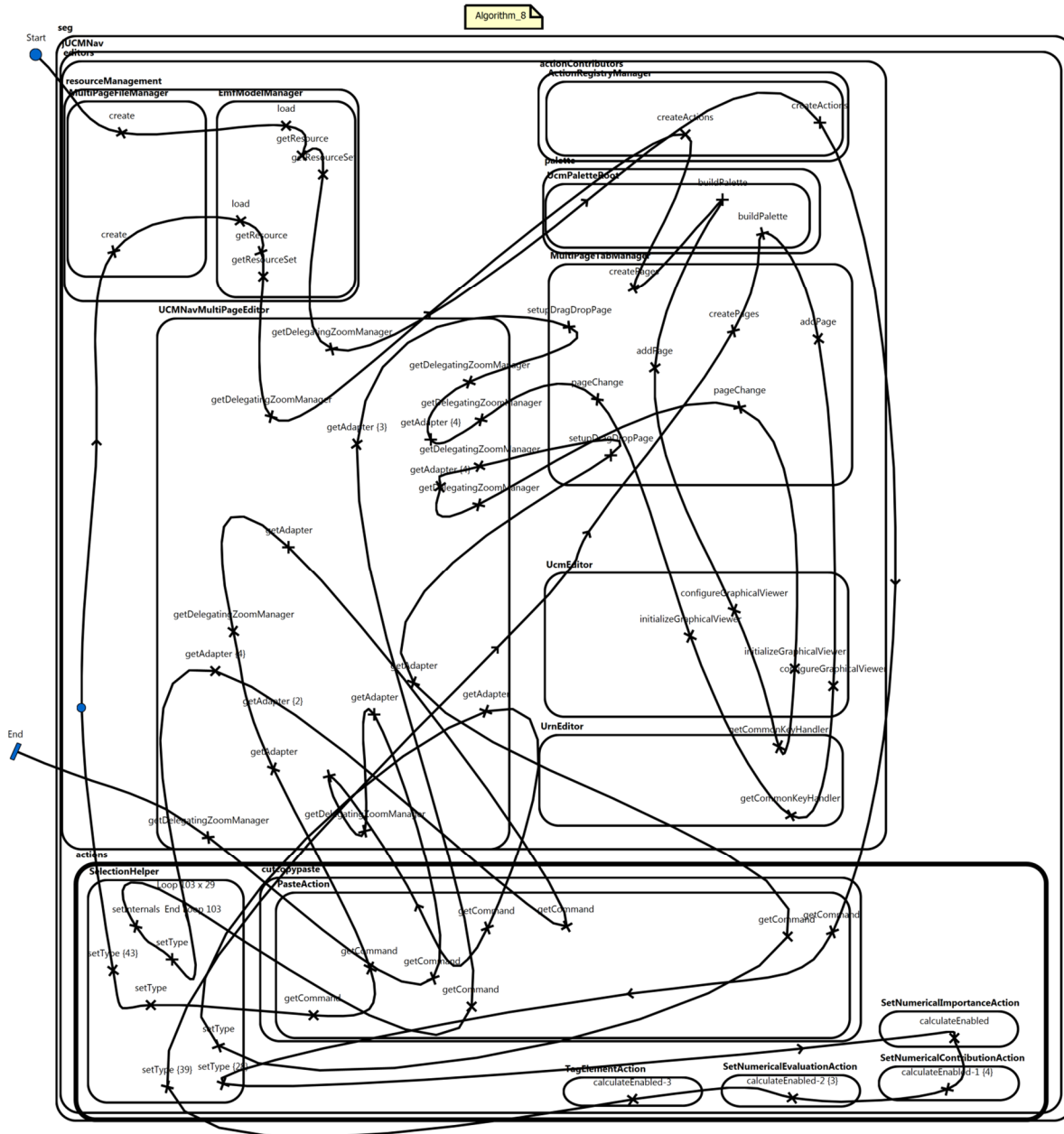


A. Algorithm 7 diagram contains the main methods used in JUCMNAV code for “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the JUCMNAV “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree



A. Algorithm 8 diagram contains the main methods used in JUCMNAV code for “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the JUCMNAV “Open and load a UCM file” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

Part II

1) Assuming the correct methods have been selected would you use UCM diagrams to document the behaviour of a feature in design documents?

Strongly disagree Disagree Neutral Agree Strongly agree

Appendix C: Behavioural Maps for Umple

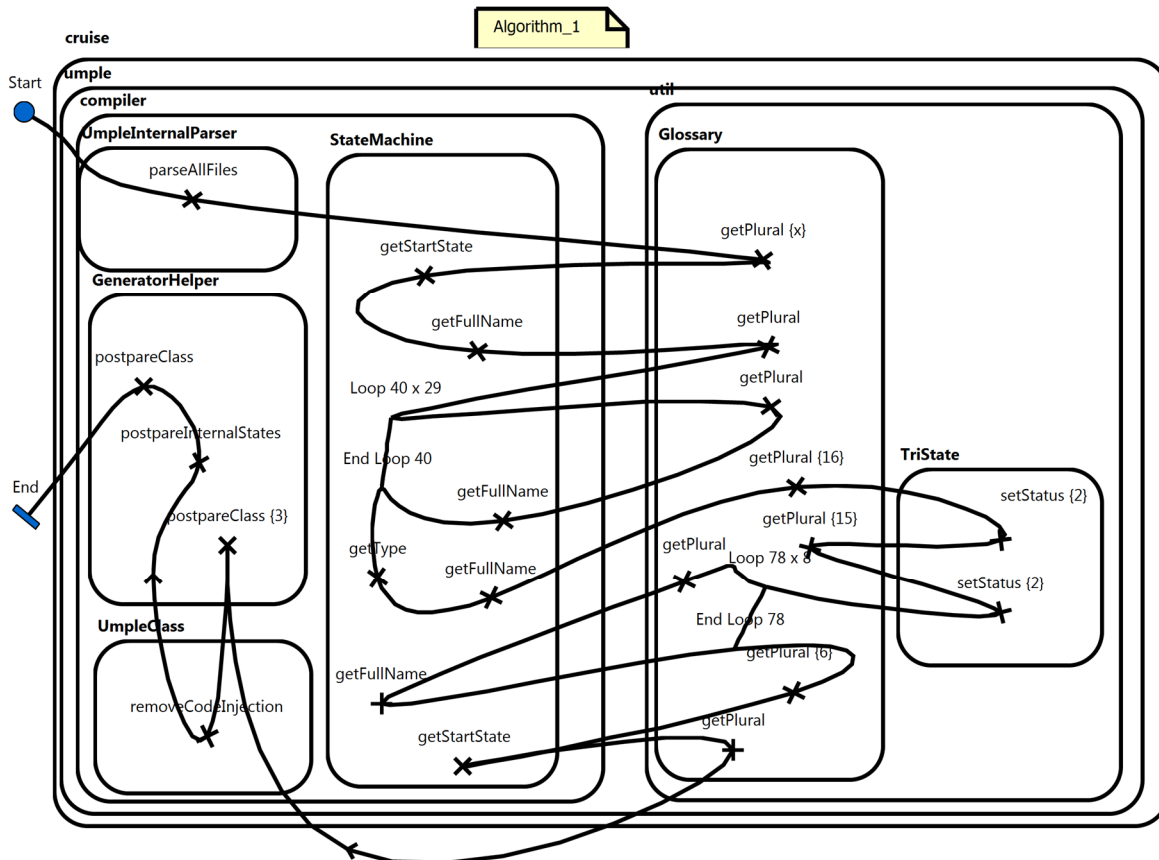
This Appendix contains the behavioural maps used in the experiment for the “Compile a file” feature of the Umple- System

For the purpose of this work please use the following definition of *utility*:

A utility is any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.

In order to reverse engineer the behavioural model of Umple by filtering out utility methods, we have run Umple to compile an Umple program and collected tracefiles. The utility methods have been filtered out using eight different reverse engineering algorithms. The filtered trace files (containing the non-utility/main methods) are used to draw the Use Case Maps describing the Umple compilation process.

Please take a look at the following 8 UCM diagrams and answer the questions below each diagram by highlighting or underlining your answer.

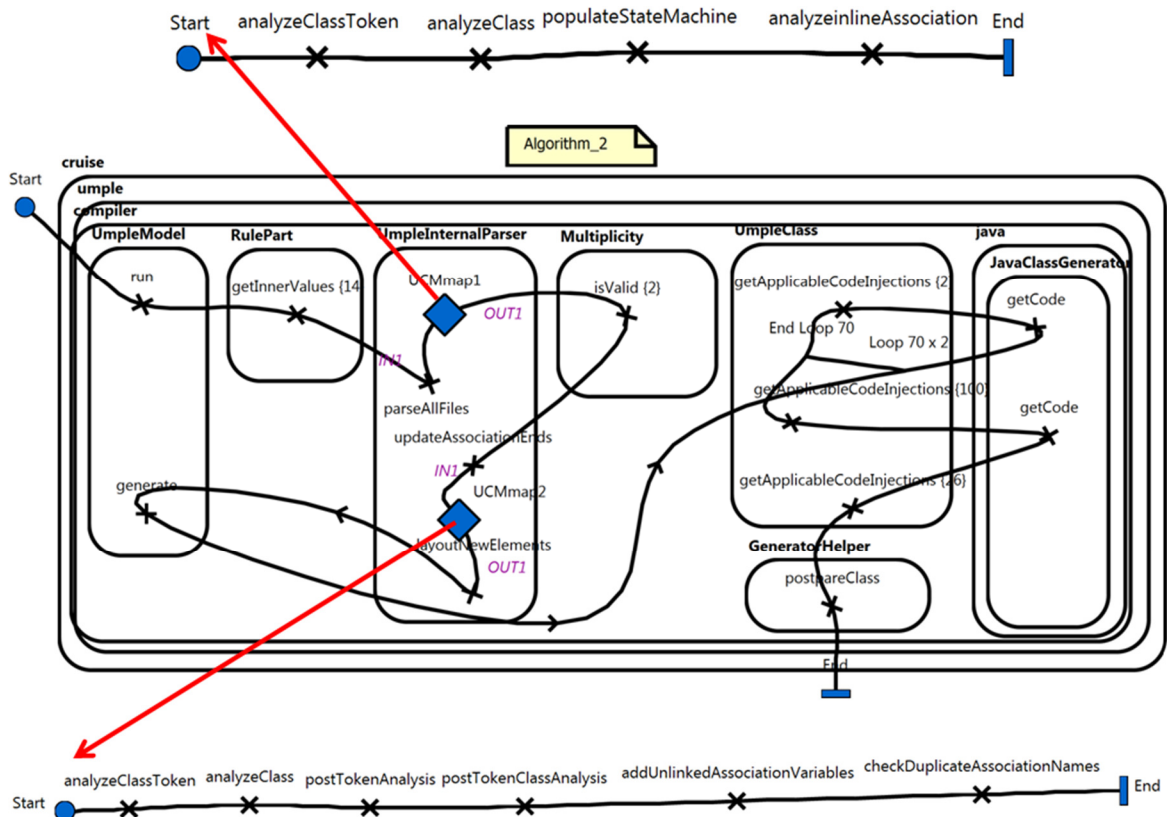


A. Algorithm 1 diagram contains the main methods used for the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

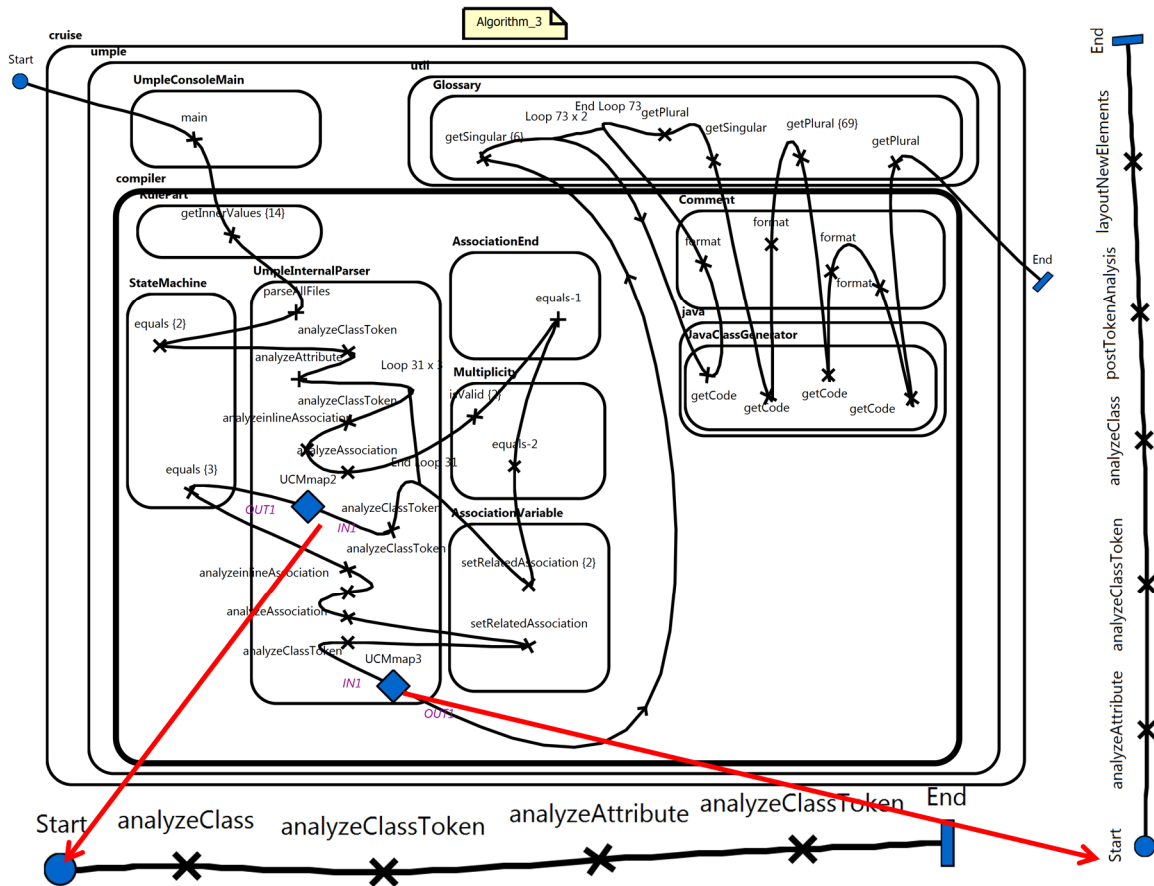


A. Algorithm 2 diagram contains the main methods used for the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

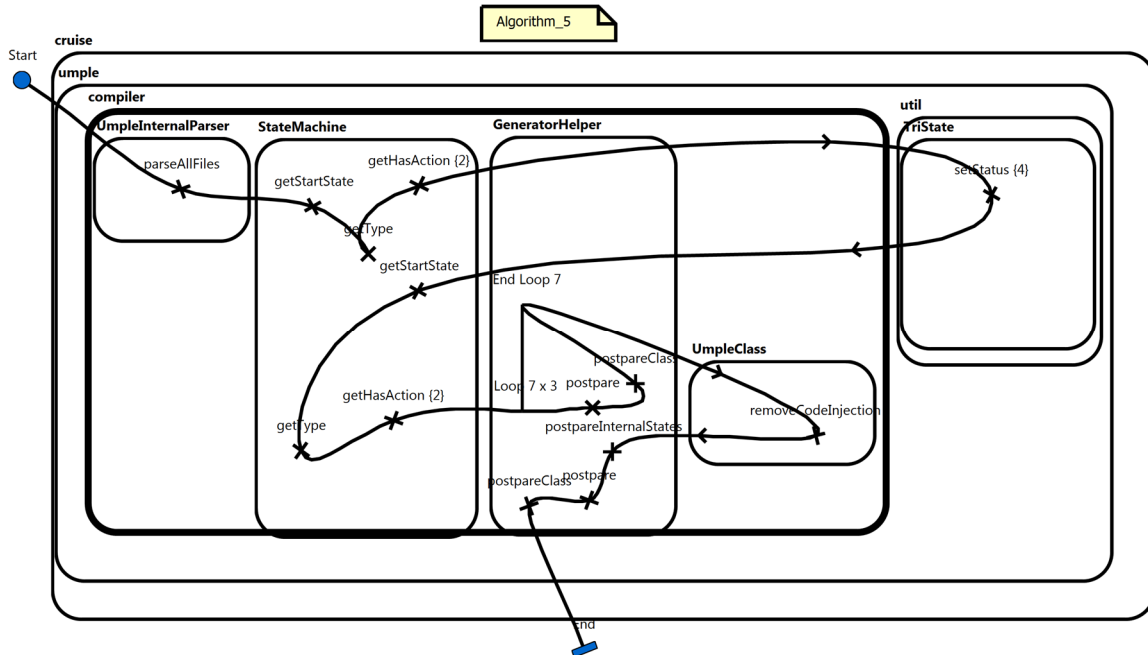


A. Algorithm 3 diagram contains the main methods used for the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

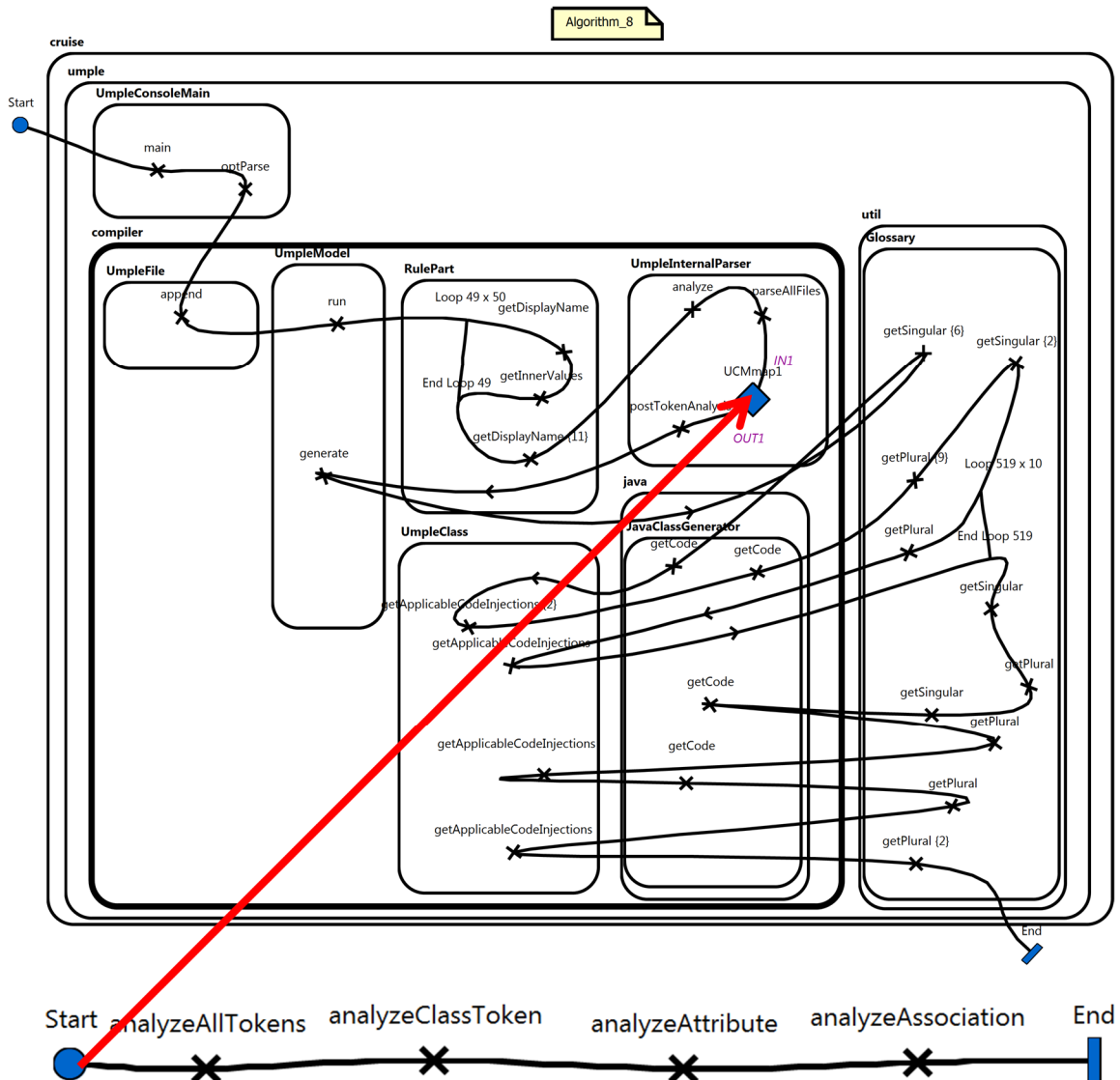


A. Algorithm 5 diagram contains the main methods used for the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree



A. Algorithm 8 diagram contains the main methods used for the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the Umple compile feature?

Strongly disagree Disagree Neutral Agree Strongly agree

Part II

1) Assuming the correct methods has been selected would you use UCM diagrams to document the behaviour of a feature in design documents?

Strongly disagree Disagree Neutral Agree Strongly agree

Appendix D: Behavioural Maps UCEd

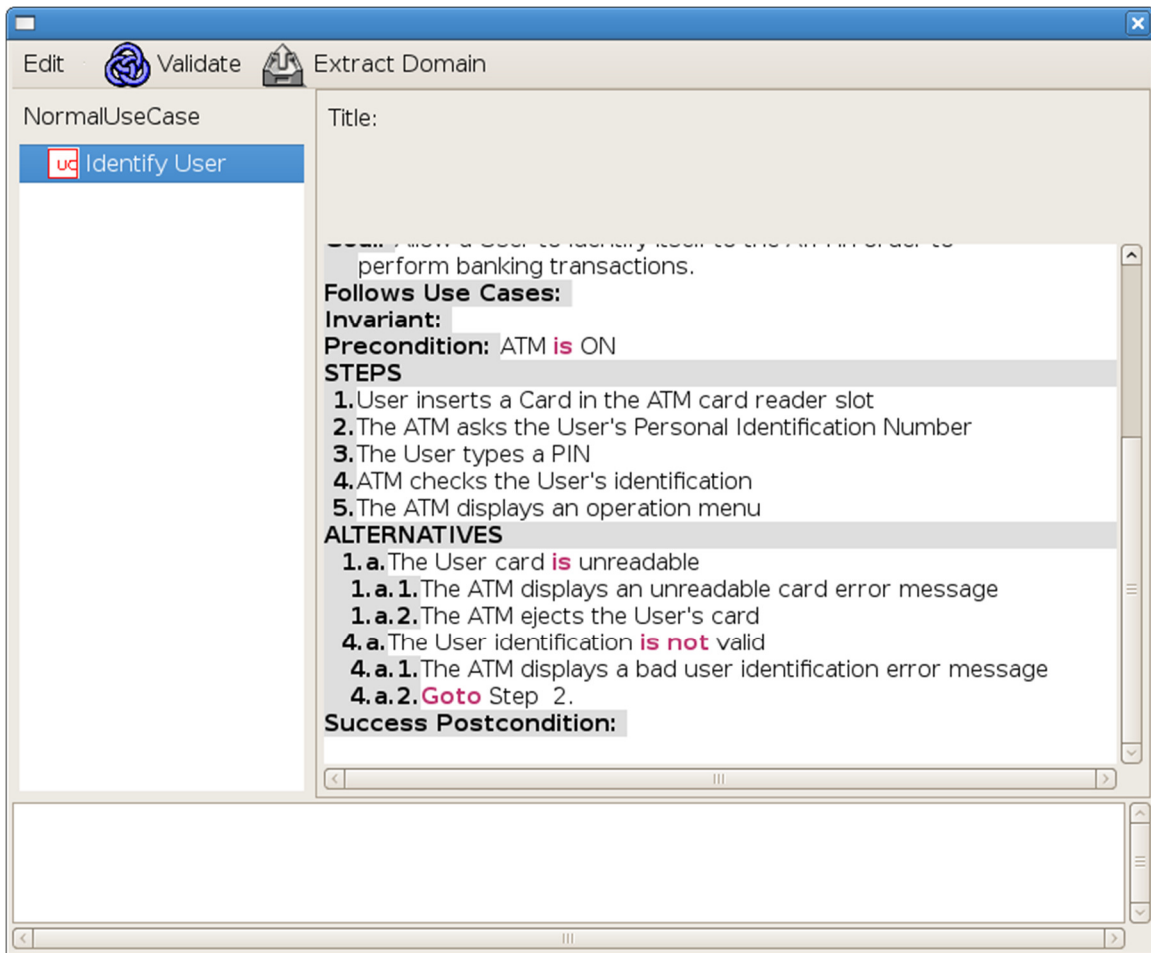
This Appendix contains the behavioural maps used in the experiment for the “Open a file” feature of the UCEd System

Part I

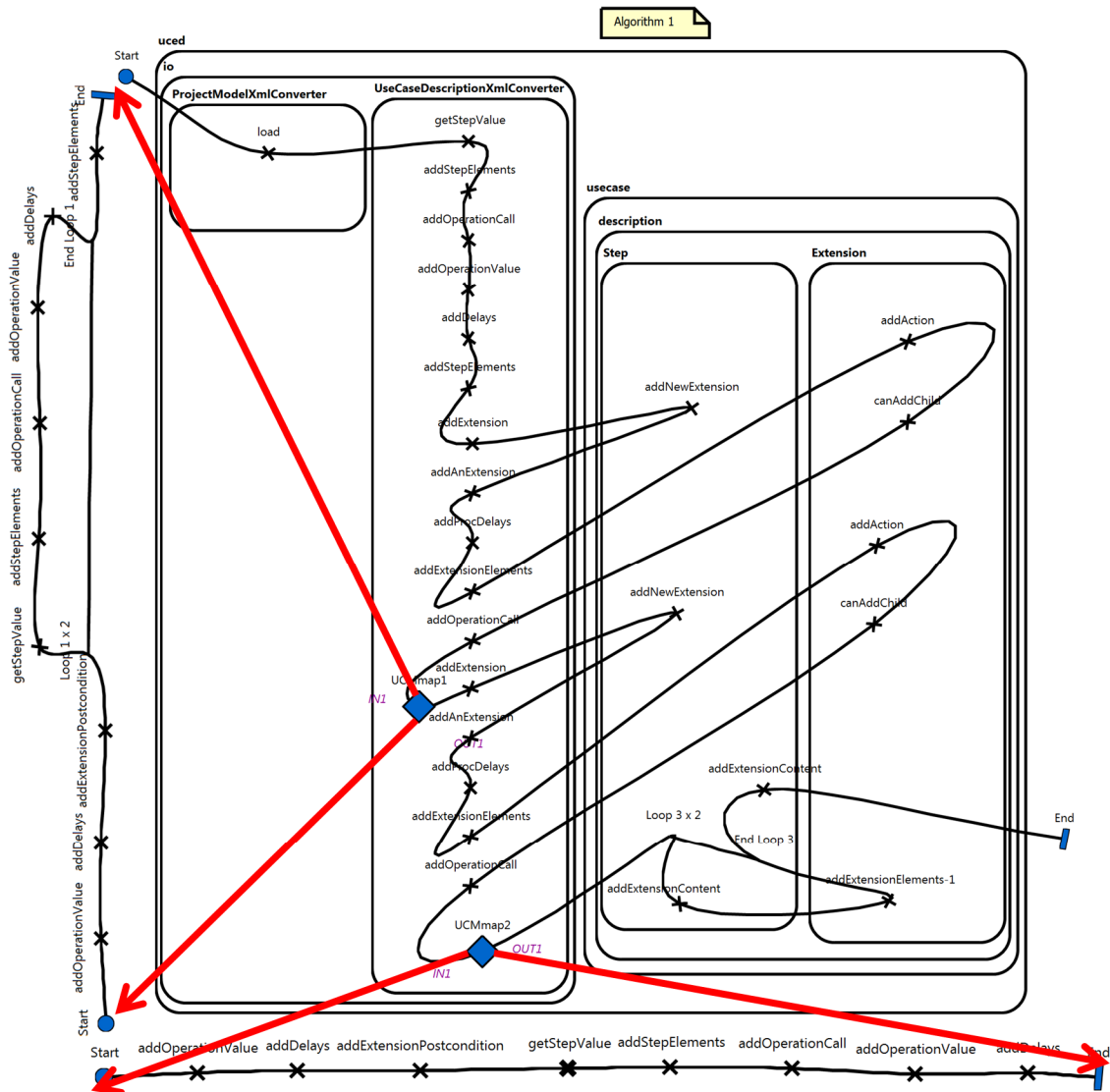
For the purpose of this work please use the following definition of *utility*:

A utility is any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program.

In order to reverse engineer the behavioural model of UCEd by filtering out utility methods, we have run the UCEd application “open a project” and opened a project shown below. The project file that is selected to be opened contains basic feature of UCEd. We collected execution trace files while the file was being opened. The utility methods have been filtered out using eight different algorithms. The filtered trace files (containing the non-utility/main methods) are used to draw the Use Case Maps below.



Please take a look at the following 8 UCM diagrams and answer the questions below each diagram by highlighting or underlining your answer.

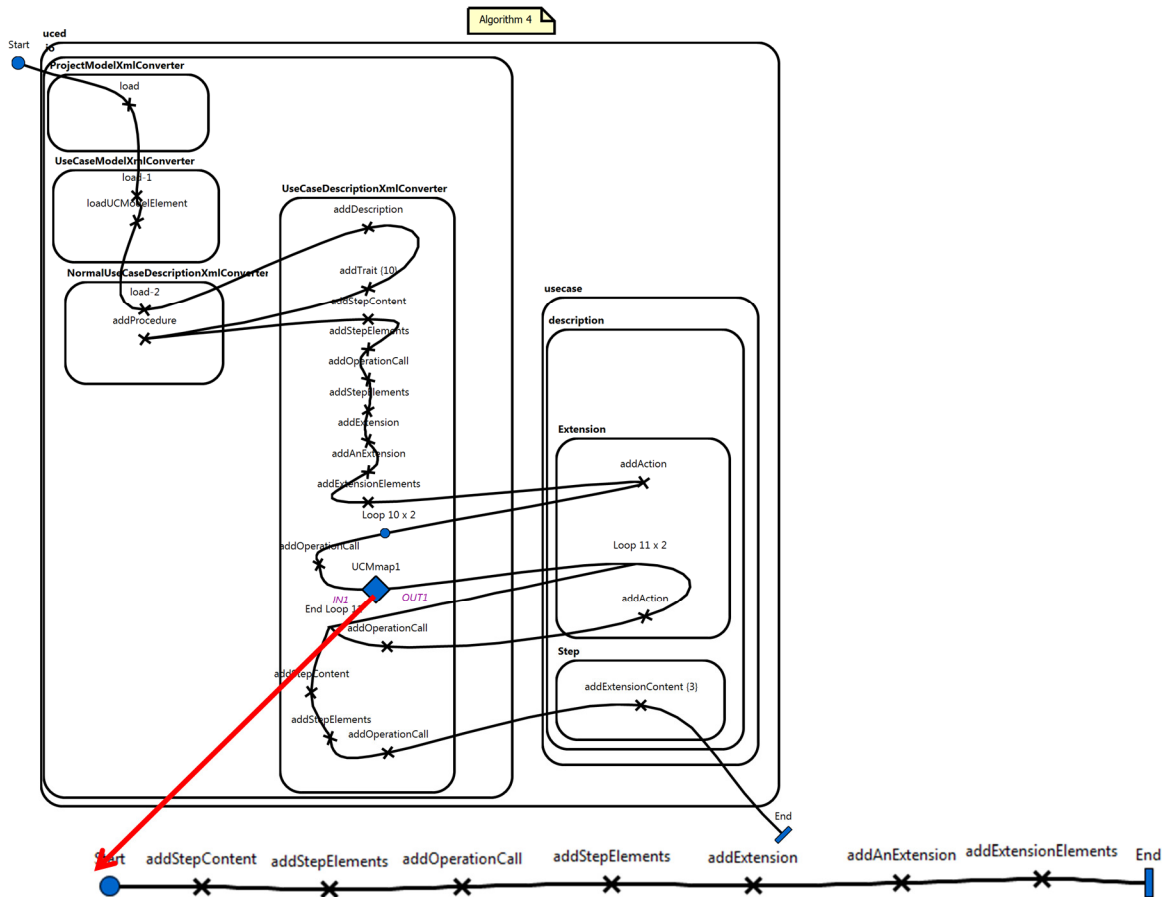


A. Algorithm 1 diagram contains the main methods used for the “open a project” feature in UCed code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCed “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

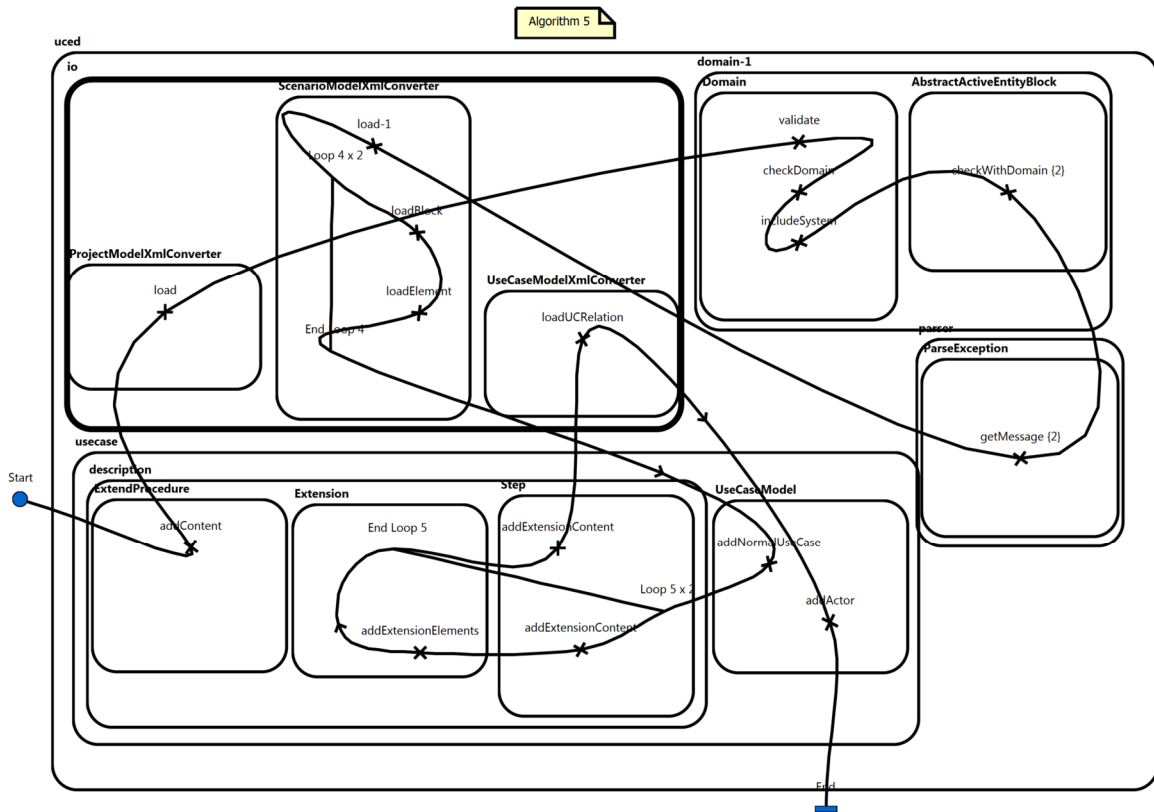


A. Algorithm 4 diagram contains the main methods used for the “Open a project feature” in UCed code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCed “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

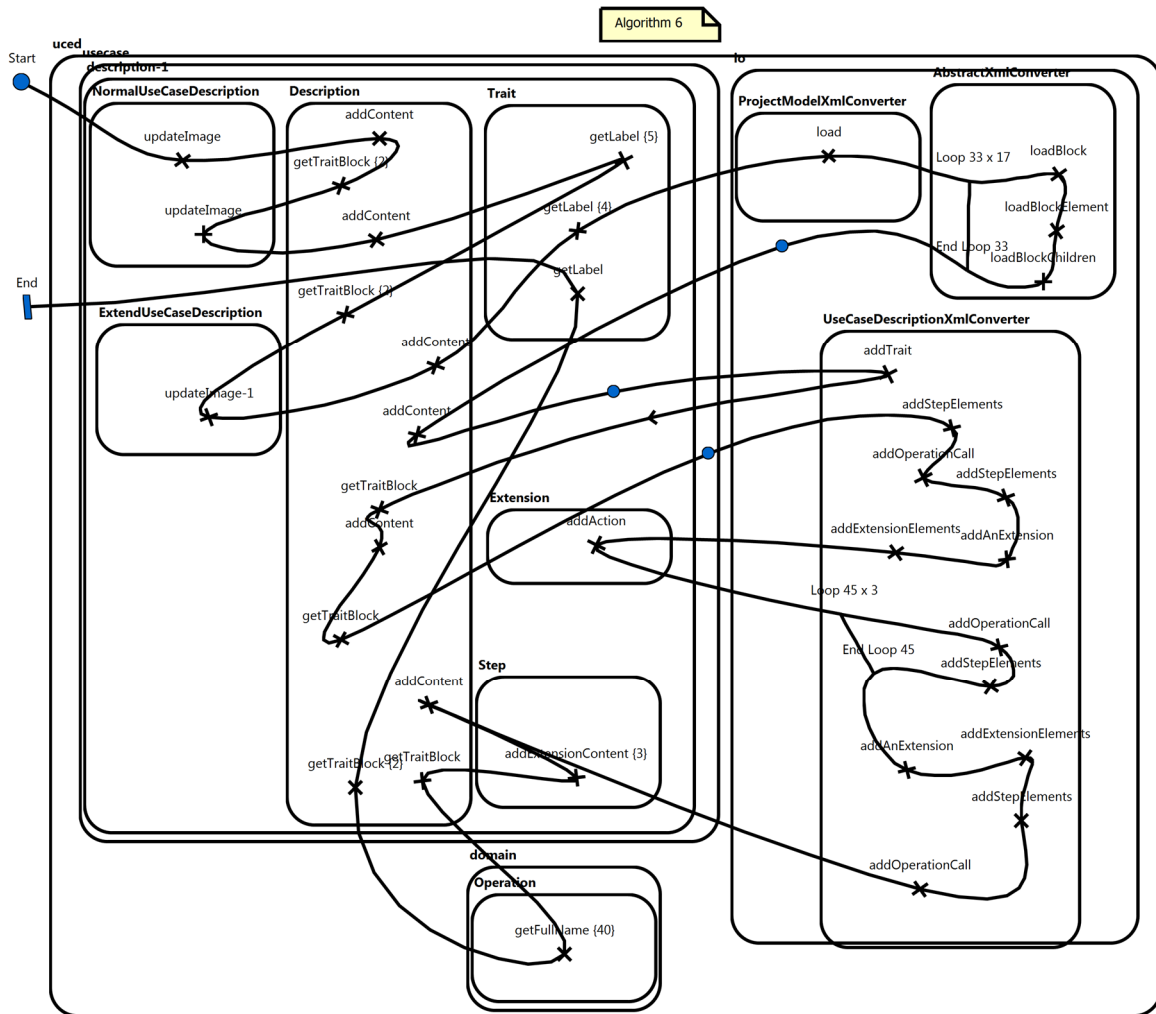


A. Algorithm 5 diagram contains the main methods used for the “Open a project feature” in UCed code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCed “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

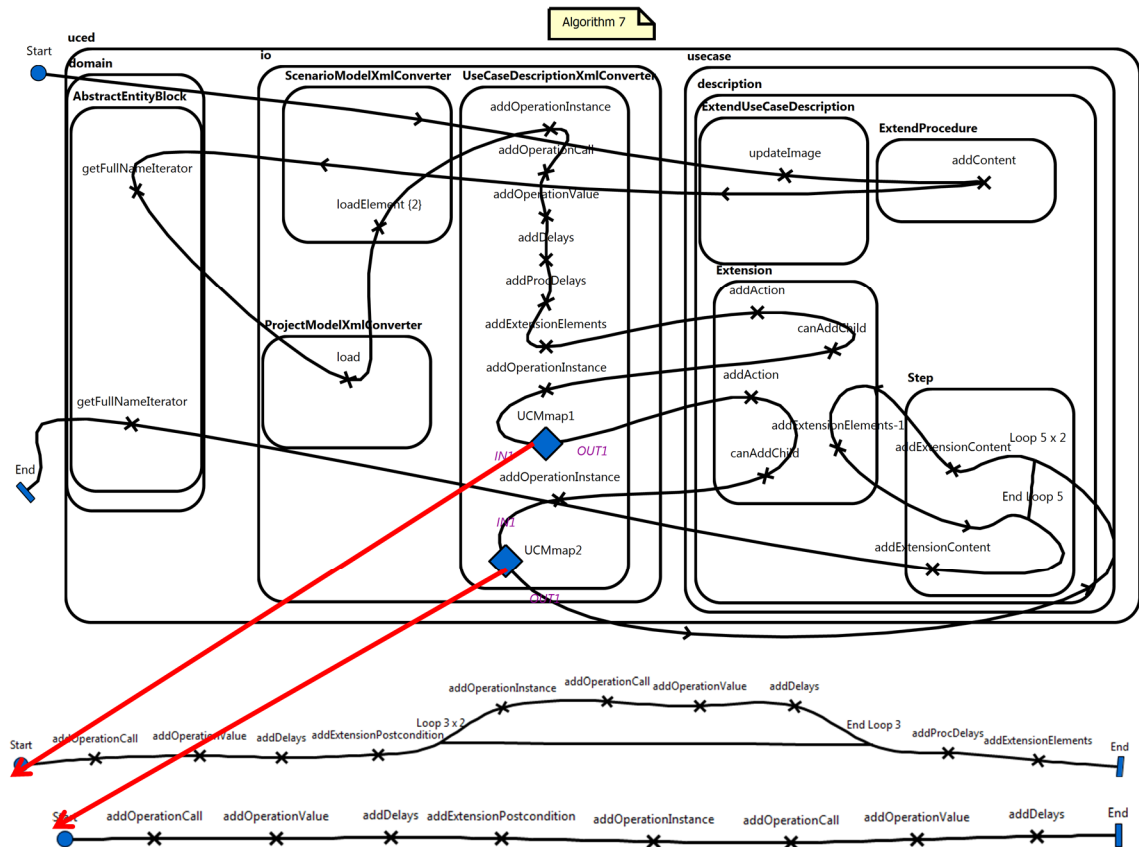


A. Algorithm 6 diagram contains the main methods used for the “Open a project feature” in UCEd code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCEd “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

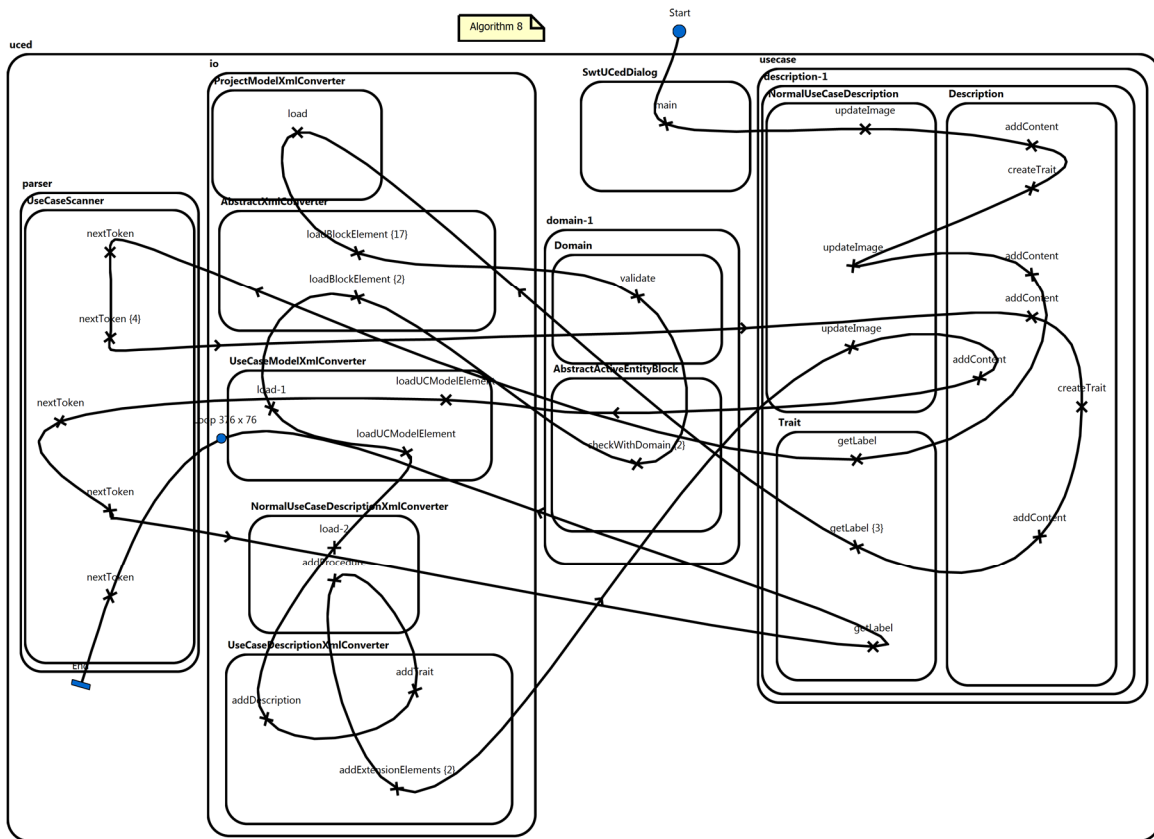


A. Algorithm 7 diagram contains the main methods used for the “Open a project feature” in UCED code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCED “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree



A. Algorithm 8 diagram contains the main methods used for the “Open a project feature” in UCed code?

Strongly disagree Disagree Neutral Agree Strongly agree

B. The UCM diagram matches the mental image you have of the UCed “open a project” feature?

Strongly disagree Disagree Neutral Agree Strongly agree

Part II

1) Assuming the correct methods have been selected would you use UCM diagrams to document the behaviour of a feature in design documents?

Strongly disagree Disagree Neutral Agree Strongly agree

Appendix E: Behavioural Maps for the TraceToUCM Tool

This Appendix contains the behavioural maps used in the experiment for the “Open a file” feature of the TraceToUCM application. The algorithms used are in the comments.

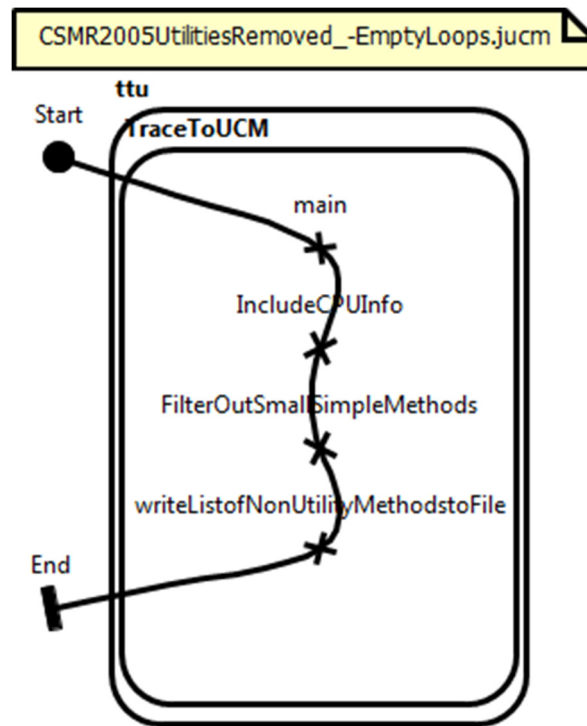


Figure 33. UCM for TraceToUCM filtered using Algorithm 1.

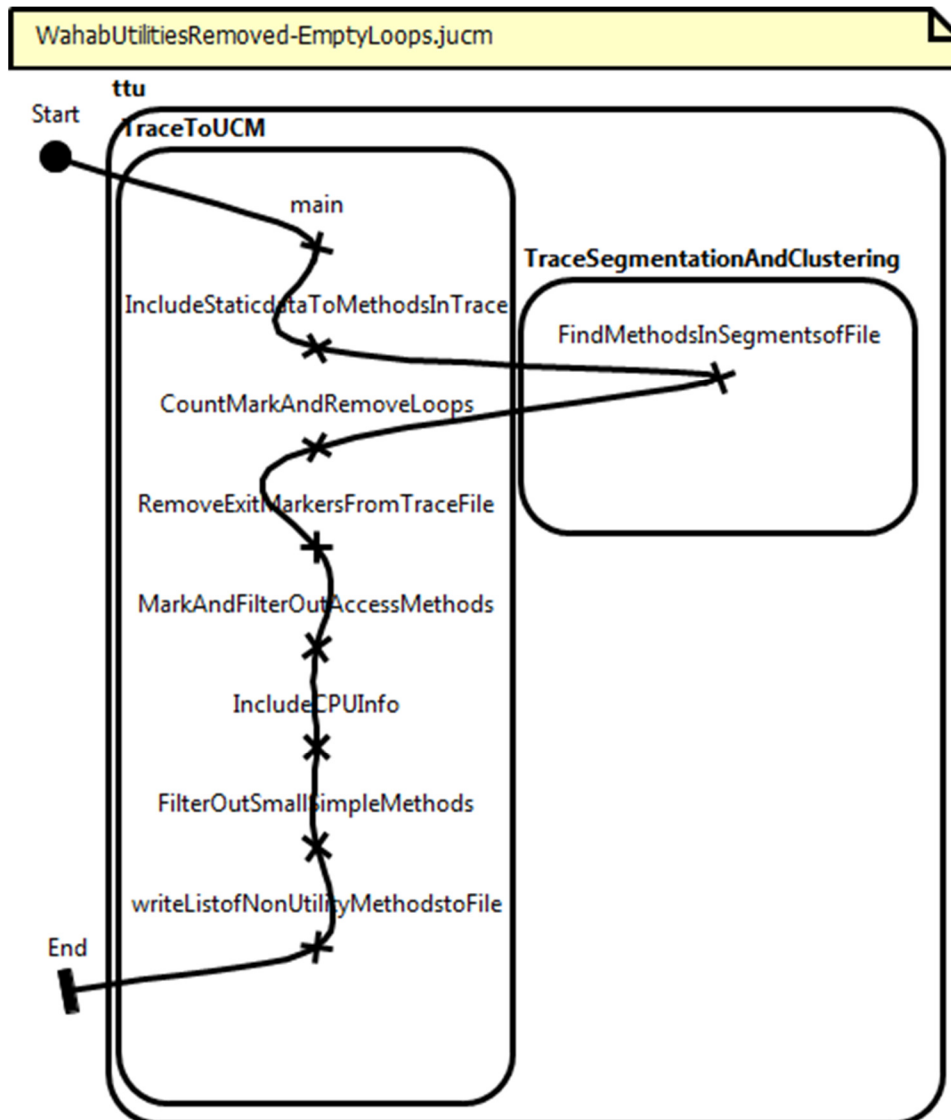


Figure 34. UCM for TraceToUCM filtered using Algorithm 2.

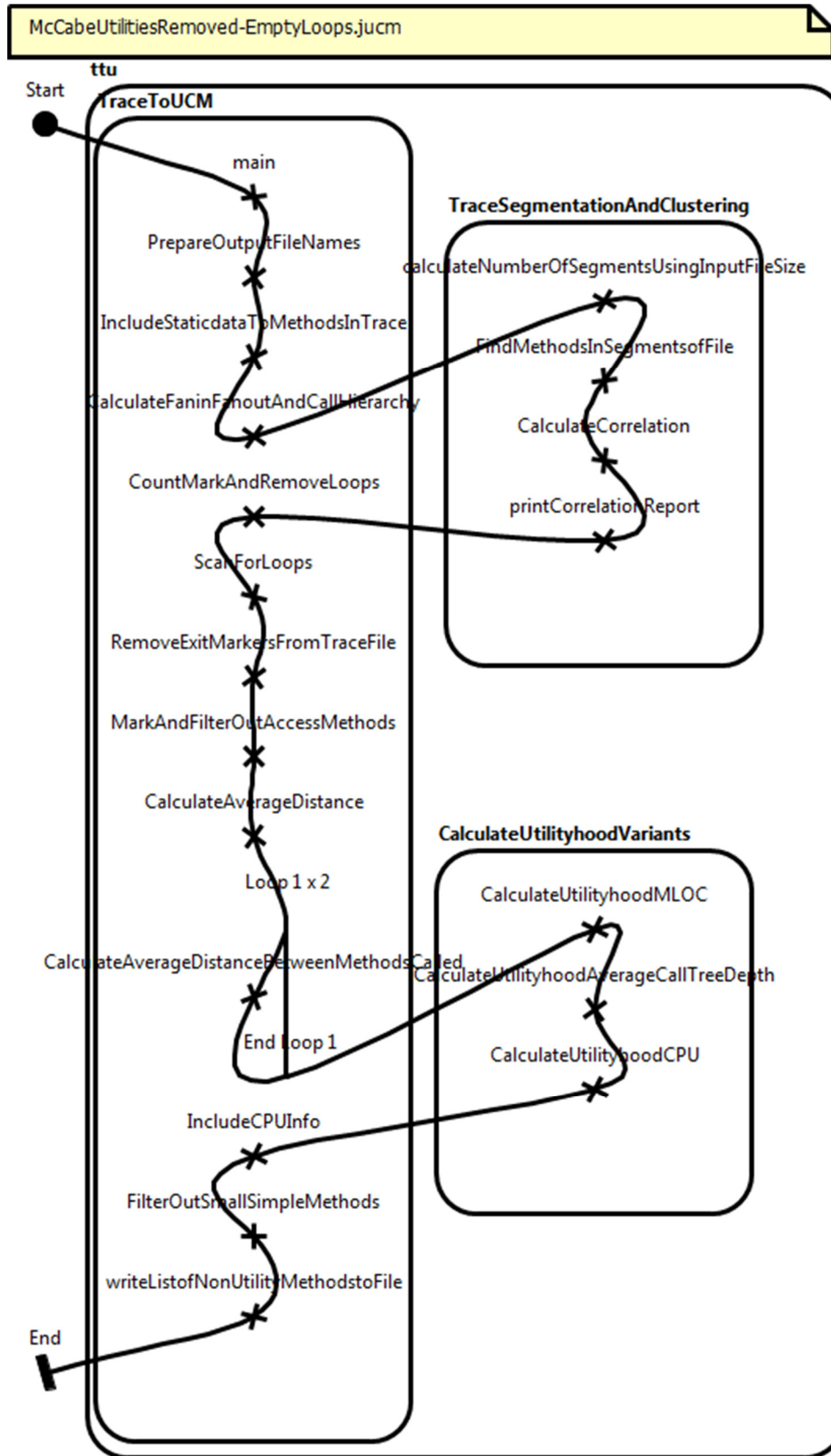


Figure 35. UCM for TraceToUCM filtered using Algorithm 3.

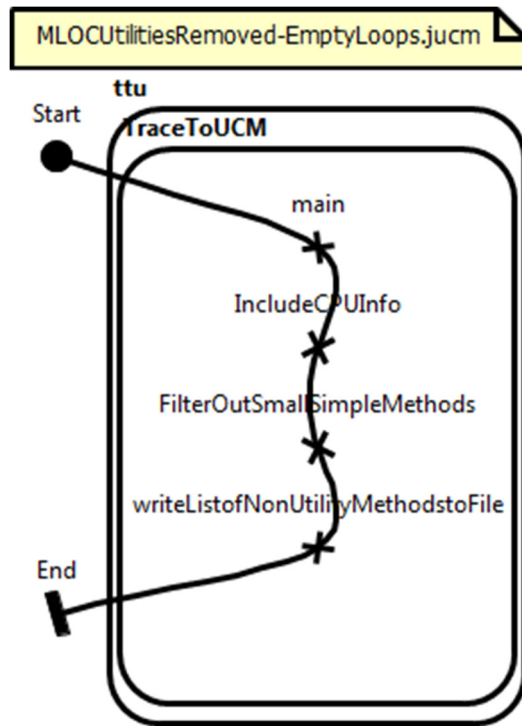


Figure 36. UCM for TraceToUCM filtered using Algorithm 4.

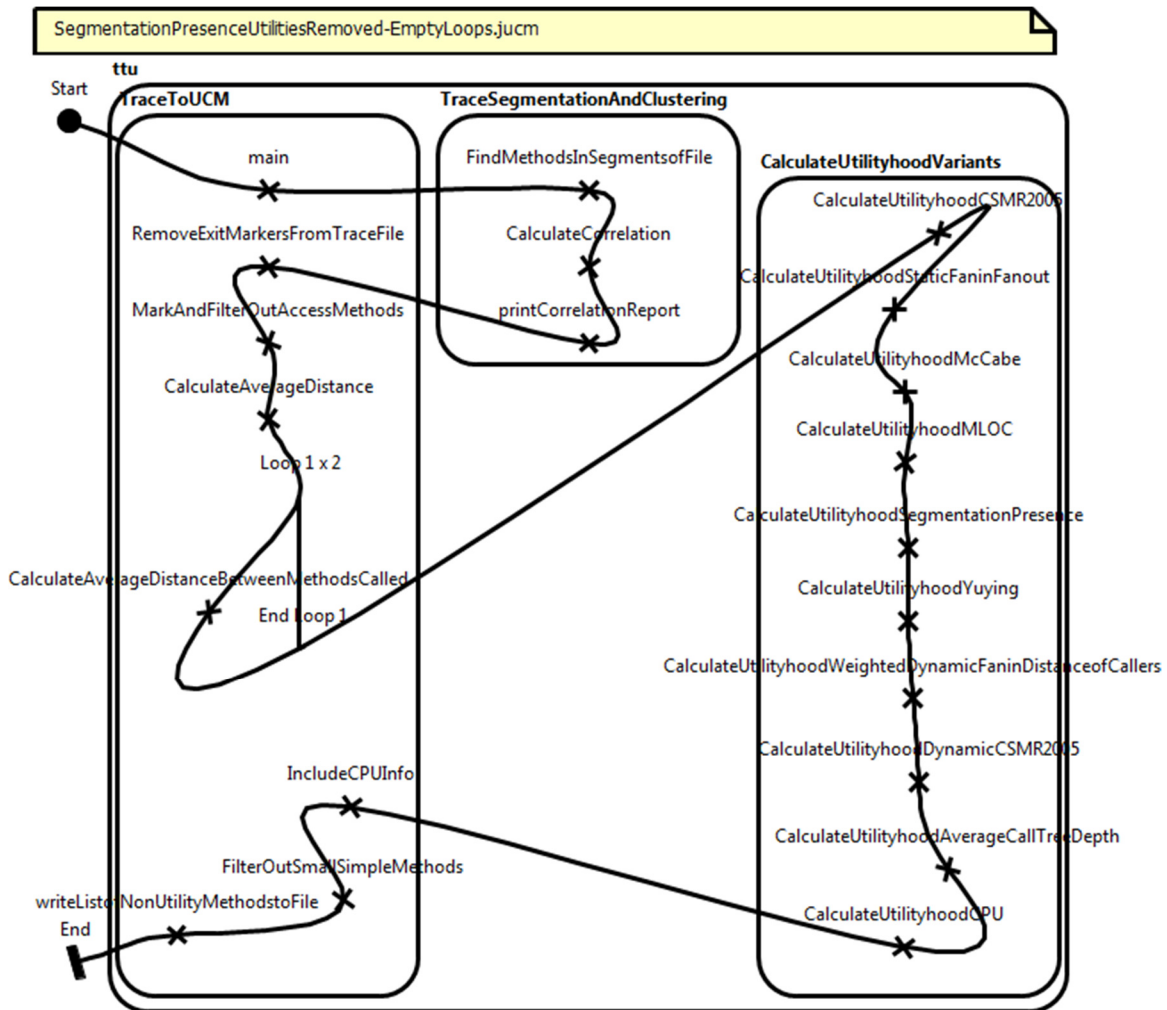


Figure 37. UCM for TraceToUCM filtered using Algorithm 5.

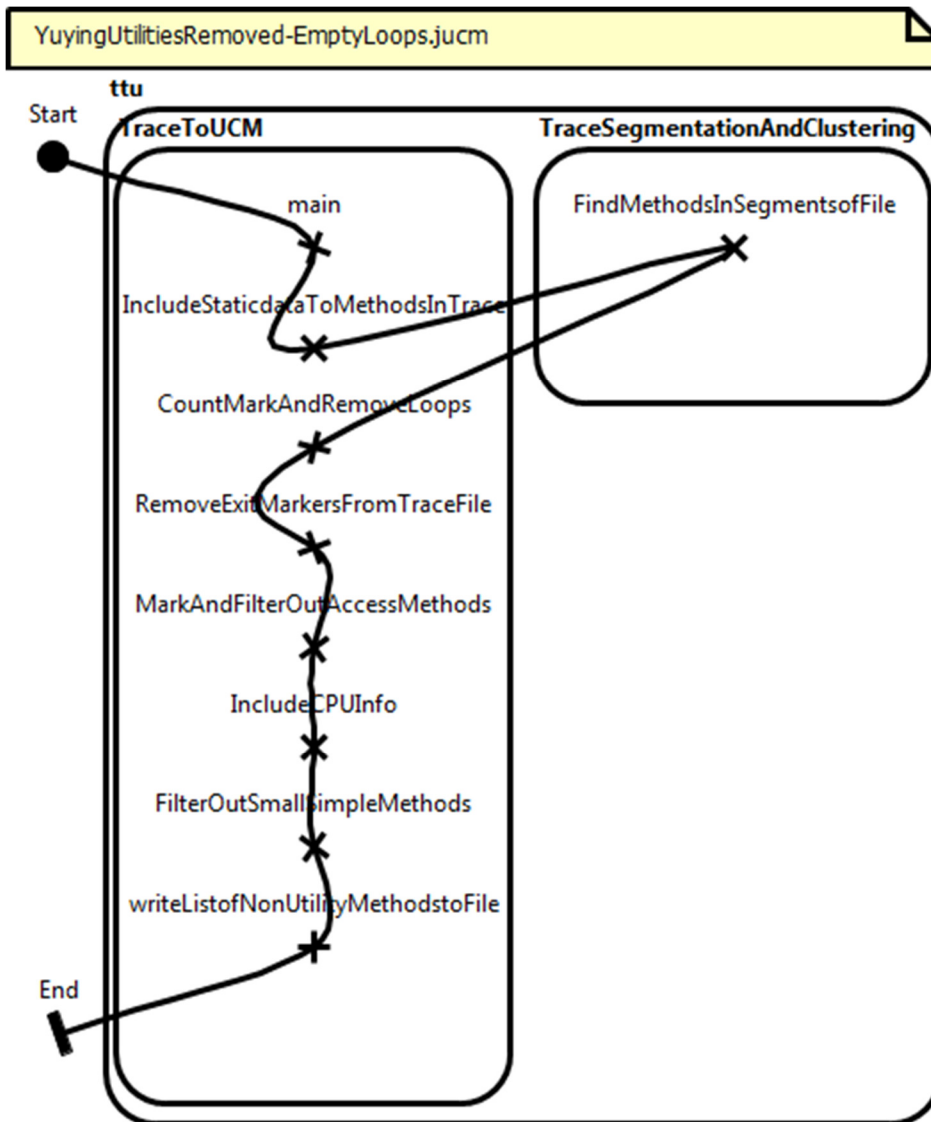


Figure 38. UCM for TraceToUCM filtered using Algorithm 6.

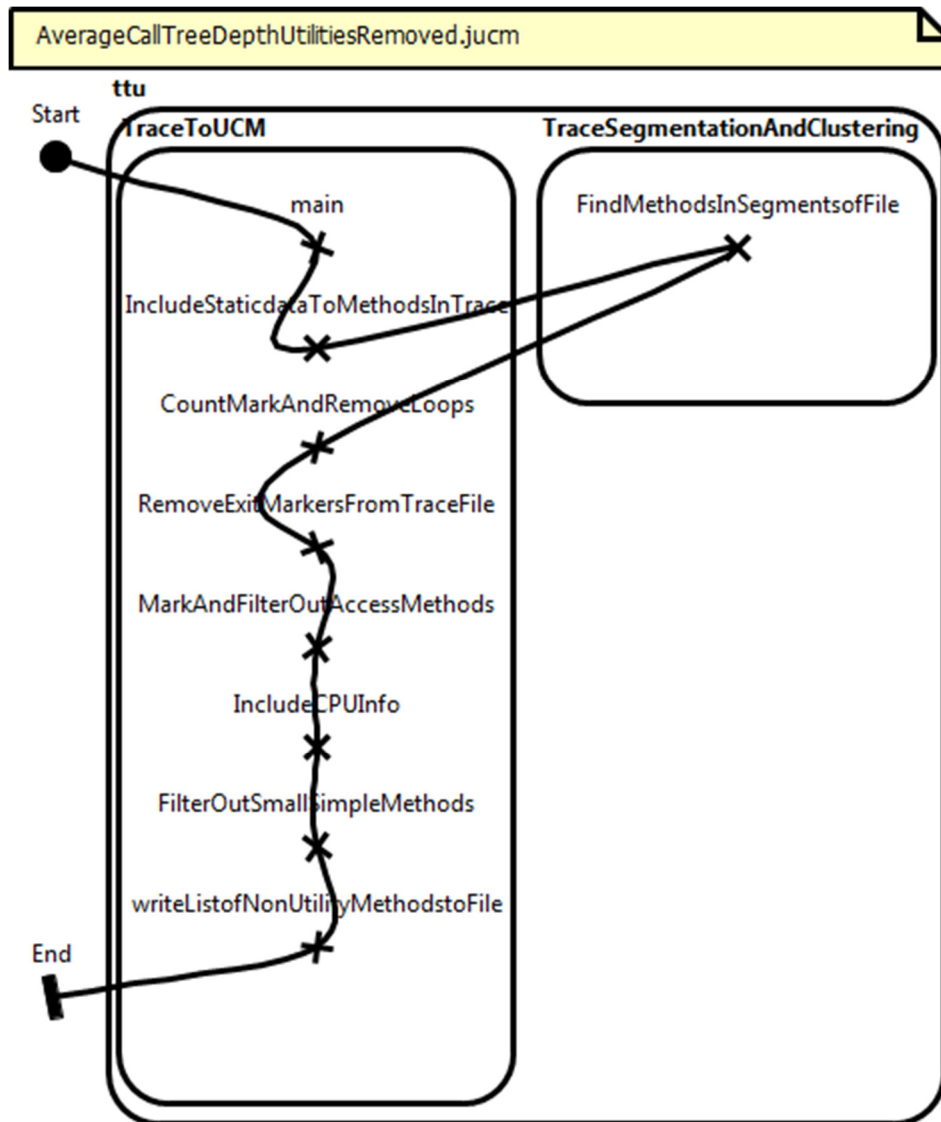


Figure 39. UCM for TraceToUCM filtered using Algorithm 7.

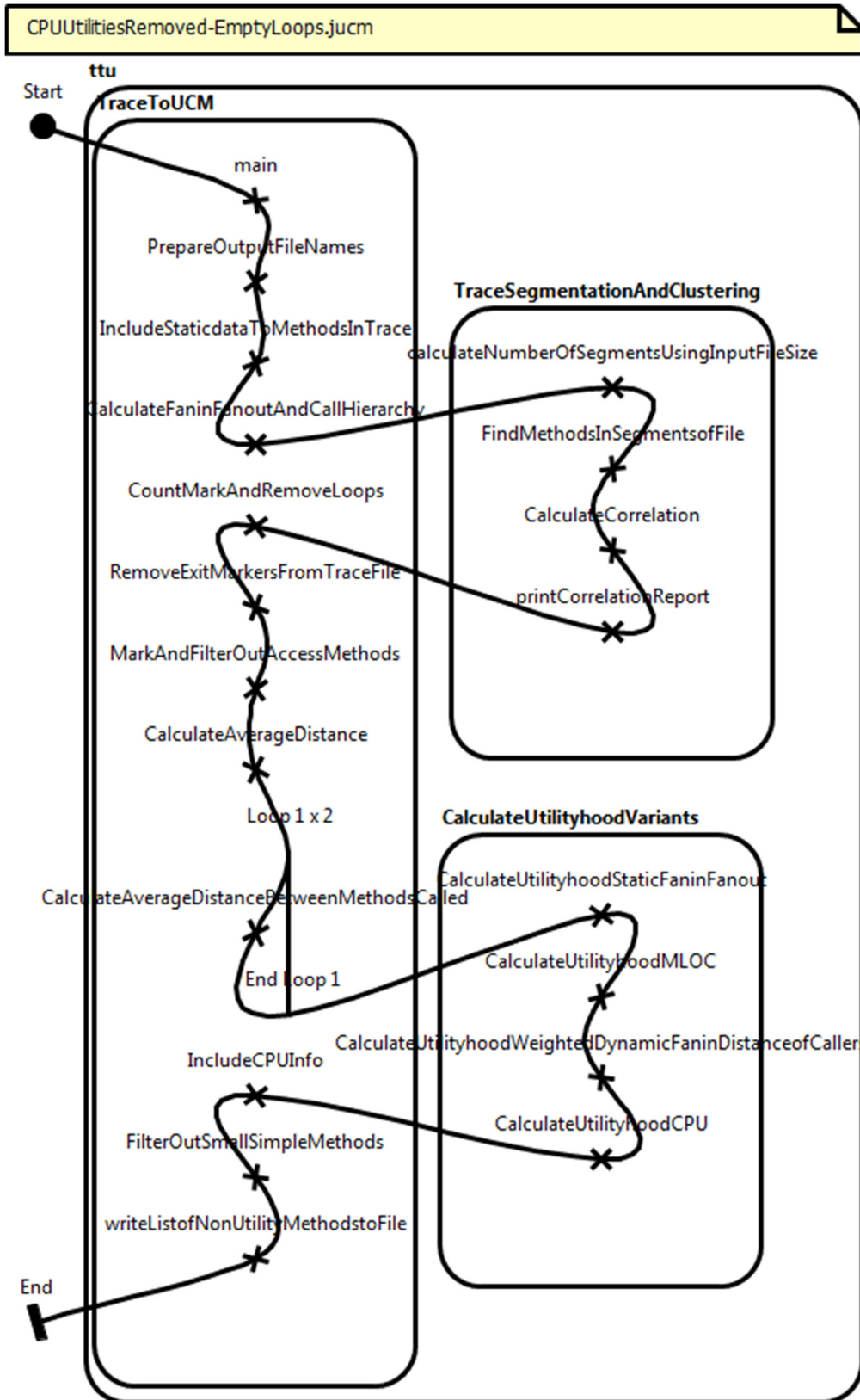


Figure 40. UCM for TraceToUCM filtered using Algorithm 8.

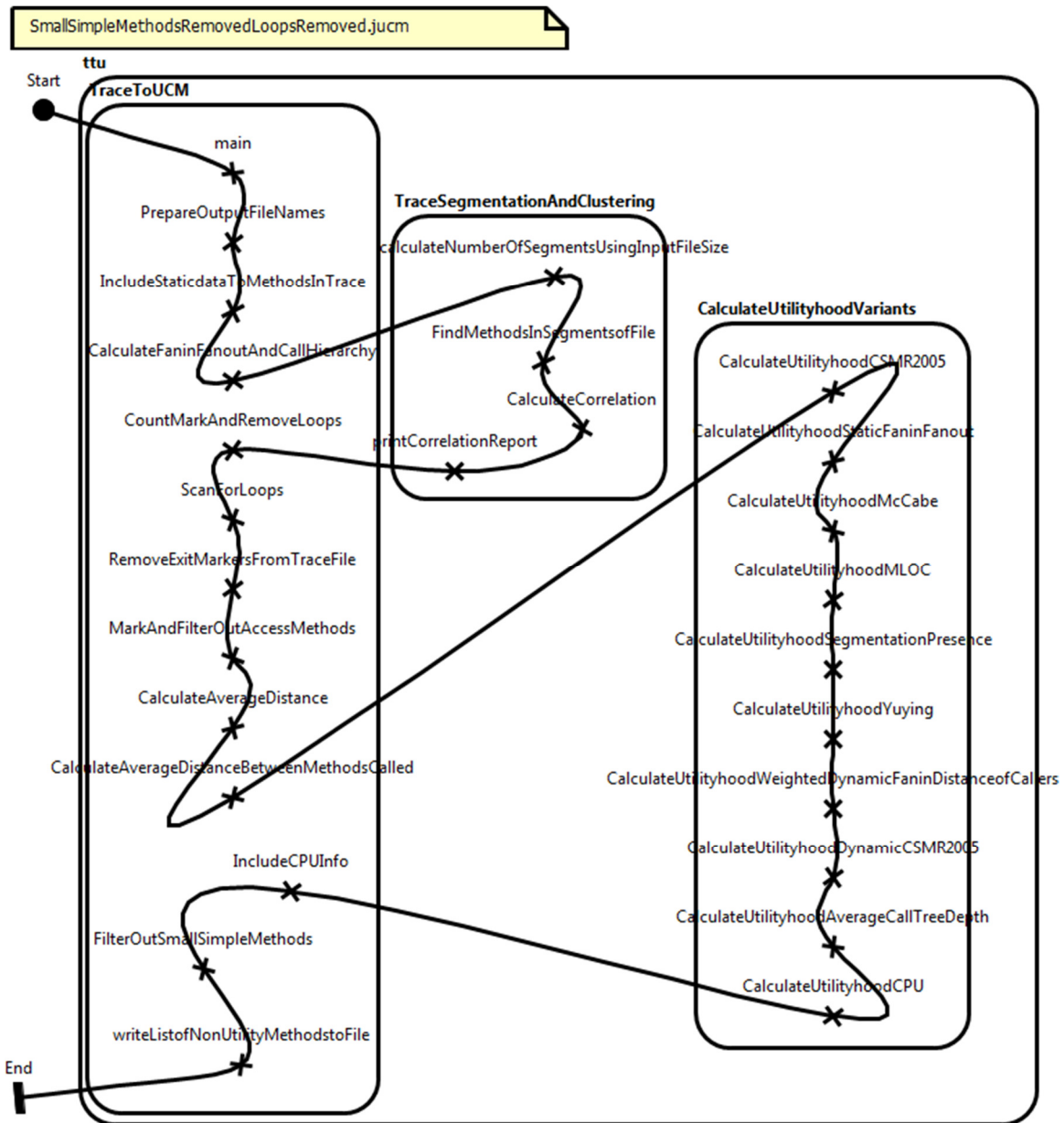


Figure 41. UCM for TraceToUCM after small and simple methods are filtered.

Appendix F: Algorithms

The main trace processing algorithm reads in an execution trace, and calculates *Dynamic* fan-in, *Dynamic* fan-out, call-hierarchy, and so on for each of the methods that were invoked in the execution trace. Data from the static file and CPU file are also collated. See the corresponding pseudo code in Figure 42.

Name: Process_Static_and_Dyanmic_parameters

Input : Trace_file, CPU_usage_file, static_info_file

Output: Condensed execution file, an array with entry for each method in the tracefile and associated dynamic static parameters, a data table containing dynamic and static parameter related to methods in the trace file, Populated the record TraceItem (shown in Table 36)

Steps:

- for (every line in the method in the execution file)
- create an entry in TraceItem (shown in Figure 43)
- store the name of the method, class, package information
- find the method in CPU usage file & copy CPU usage data to TractItem attributes
- find the method in static_info_file & copy data to TraceItem attributes
- Derive dynamic information from the execution trace as follows

dynamic fan-in and dynamic fan-out

- We count fan-in and fan-out by going through the entire trace and building a list of methods that are called by a particular method, and also a list of methods that call that same method.
- We then count the number of entries in the lists to create the fan-in and fan-out values for the method

Unique fan-in of how many unique methods called this method

Unique fan-out how many unique methods this method called

Calculate **EncounteredinTrace**

- Keep a tally of how many times the method is encountered in the execution trace

Calculate **AverageCallHierarchy**

- Add the call hierarchy for each time the method is encountered and divided by the number of times it is encountered.

Leaf or not? If fan-out == 0 then Leaf is TRUE

Access method or not? If method name contains “.get” or “.set” then Access method is TRUE

Figure 42. Pseudocode for dynamic and static data processing.

```

Class TraceItem()

int numOfLineInOriginalFile = 0;
String TraceItemName = new String();      // method name
Integer FanIn = new Integer(1);           // number of methods that called this method
Integer FanOut = new Integer(1);          // number of called methods
Vector<String> calledBy = new Vector();     // Unique List of callers
HashMap<String, Integer> numberOfTimesCalledBy = new HashMap();

// the total number of calls including multiple from same method
Integer totalNumberCalledBy = new Integer(0);

// number of times a method called by, key is the method name
HashMap<String, Integer> calledByDistance = new HashMap();

// Initialize to negative if it is not called by anyone
Float averageCalledByDistance = new Float(-1.0);

// Initialize to negative if it is not called by anyone
Float averageCalledByDistanceBetweenMethodsCalled = new Float(-1.0);

// Unique List of methods Called by this method
Vector<String> methodsCalled = new Vector();

// # of times a method called, key is the method name
HashMap<String, Integer> numberOfTimesMethodsCalled = new HashMap();

// the total number of calls including multiple from same method
Integer totalNumberMethodsCalled = new Integer(0);

// list of depth of calls in the call hierarchy (used for average)
Vector<Integer> depths = new Vector();

float MaxMLoc = 1;
double AverageCallHierarchy;
Boolean Leaf = new Boolean(false);          // This method is leaf if number of methodsCalled = 0;
Boolean DataAccessMethod = new Boolean(false);

// Static Data
String JavaFile = new String(); // JavaFile is the name of the class the method is in.
Float MLOC = new Float (0); // Method Line of Code
Float NORM = new Float (0);      Float McCabe = new Float (1);
Float NOF = new Float (0);      Float NBD = new Float (0);
Float NSF = new Float (0);      Float NSC = new Float (0);
Float NOC = new Float (999);    Float DIT = new Float (999);
Float NOI = new Float (999);    Float NOM = new Float (999);
Float VG = new Float (999);     Float RMI = new Float (999);
Float PAR = new Float (999);    Float CA = new Float (999);
Float LCOM = new Float (999);   Float CE = new Float (999);
Float NSM = new Float (999);    Float RMD = new Float (999);
Float RMA = new Float (999);    Float SIX = new Float (999);
Float WMC = new Float (999);

// CPU Data
Float CpuUsage = new Float (0);
Float CpuUsageCumulative = new Float(-999);
Float CpuUsageAverage = new Float(-999);
Float CpuUsageBase = new Float(-999);

// Segmentation
// This value stores whether or not this trace item is present in a given segment. Length is number of Segments.
Vector<Integer> segmentPresence = new Vector();
int TotalSegmentPresence = 1;
int NumberOfSegments = 1;
Float PercentageOfSegmentsMethodPresentIn = new Float(1);

```

Figure 43. Class description for TraceItem.

Appendix G: jUCMNav – Open UCM Principal Diagram

The following list containing principal methods for the “open a UCM diagram” feature of jUCMNav is provided by jUCMNav creator. All are in the seg/jUCMNav/ package.

```
model/util/URNReferencerChecker/sanitizeReferences
model/util/URNNamingHelper/sanitizeURNspec
editors/UCMNavMultiPageEditor/setTarget
editors/UCMNavMultiPageEditor/setPageText
editors/UCMNavMultiPageEditor/setModel
editors/UCMNavMultiPageEditor/setInput
editors/UCMNavMultiPageEditor/setDirty
editors/UCMNavMultiPageEditor/setActivePage
editors/UCMNavMultiPageEditor/selectionChanged
editors/UCMNavMultiPageEditor/removePageChangeListener
editors/UCMNavMultiPageEditor/pageChange
editors/UCMNavMultiPageEditor/notifyChanged
editors/UCMNavMultiPageEditor/isSaveAsAllowed
editors/UCMNavMultiPageEditor/isDirty
editors/UCMNavMultiPageEditor/initializePageSwitching
editors/UCMNavMultiPageEditor/getZoomManager
editors/UCMNavMultiPageEditor/getTarget
editors/UCMNavMultiPageEditor/getSelectionSynchronizer
editors/UCMNavMultiPageEditor/getSelectionListener
editors/UCMNavMultiPageEditor/getResourceTracker
editors/UCMNavMultiPageEditor/getPageCount
editors/UCMNavMultiPageEditor/getMultiPageTabManager
editors/UCMNavMultiPageEditor/getMultiPageCommandStackListener
editors/UCMNavMultiPageEditor/getModel
editors/UCMNavMultiPageEditor/getFileManager
editors/UCMNavMultiPageEditor/createPages
editors/resourceManagement/UrnModelManager/getModel
editors/resourceManagement/MultiPageFileManager/create
editors/resourceManagement/EmfModelManager/load
editors/actionContributors/UCMActionBarContributor/-init-
editors/actionContributors/UCMActionBarContributor/declareGlobalActionKeys
editors/actionContributors/UCMActionBarContributor/contributeToToolBar
editors/actionContributors/UCMActionBarContributor/buildActions
```

Appendix H: Umple – Principal Methods for the Compile Feature

The following list containing principal methods for the compile feature of Umple is provided by Umple creator. All are located in the `cruise/umple/` package.

`compiler/ParseResult/getWasSuccess`
`compiler/ParseResult/setPosition`
`compiler/ParseResult/setWasSuccess`
`compiler/UmpleInternalParser/addExtendsTo`
`compiler/UmpleInternalParser/addNecessaryFiles`
`compiler/UmpleInternalParser/addUnlinkedAssociations`
`compiler/UmpleInternalParser/addUnlinkedAssociationVariables`
`compiler/UmpleInternalParser/addUnlinkedExtends`
`compiler/UmpleInternalParser/addUnlinkedInterfaceExtends`
`compiler/UmpleInternalParser/analyze`
`compiler/UmpleInternalParser/analyzeAllTokens`
`compiler/UmpleInternalParser/analyzeAssociation`
`compiler/UmpleInternalParser/analyzeAttribute`
`compiler/UmpleInternalParser/analyzeClass`
`compiler/UmpleInternalParser/analyzeClassToken`
`compiler/UmpleInternalParser/analyzeComment`
`compiler/UmpleInternalParser/analyzeCoreToken`
`compiler/UmpleInternalParser/analyzeinlineAssociation`
`compiler/UmpleInternalParser/analyzeLayoutToken`
`compiler/UmpleInternalParser/analyzeMultilineComment`
`compiler/UmpleInternalParser/analyzeState`
`compiler/UmpleInternalParser/analyzeStateMachine`
`compiler/UmpleInternalParser/analyzeStateMachineToken`
`compiler/UmpleInternalParser/analyzeToken`
`compiler/UmpleInternalParser/analyzeTraceToken`
`compiler/UmpleInternalParser/checkDuplicateAssociationNames`
`compiler/UmpleInternalParser/checkExtendsForCycles`
`compiler/UmpleInternalParser/checkSingletonAssociations`
`compiler/UmpleInternalParser/createAssociation`
`compiler/UmpleInternalParser/createStateFromDefinition`
`compiler/UmpleInternalParser/layoutNewElements`
`compiler/UmpleInternalParser/parse`
`compiler/UmpleInternalParser/parseAllFiles`
`compiler/UmpleInternalParser/populateStateMachine`

compiler/UimpleInternalParser/postTokenAnalysis
compiler/UimpleInternalParser/postTokenClassAnalysis
compiler/UimpleInternalParser/postTokenCoreAnalysis
compiler/UimpleInternalParser/postTokenInterfaceAnalysis
compiler/UimpleInternalParser/postTokenLayoutAnalysis
compiler/UimpleInternalParser/postTokenStateMachineAnalysis
compiler/UimpleInternalParser/postTokenTraceAnalysis
compiler/UimpleInternalParser/setStrictness
compiler/UimpleInternalParser/updateAssociationEnds
compiler/UimpleInternalParser/verifyClassesInUse
compiler/UimpleModel/addAssociation
compiler/UimpleModel/addGenerate
compiler/UimpleModel/addUimpleClass
compiler/UimpleModel/clearGenerates
compiler/UimpleModel/generate
compiler/UimpleModel/getAssociation
compiler/UimpleModel/getAssociations
compiler/UimpleModel/getDefaultGenerate
compiler/UimpleModel/getGeneratedCode
compiler/UimpleModel/getGenerates
compiler/UimpleModel/getGlossary
compiler/UimpleModel/getShouldGenerate
compiler/UimpleModel/getTraceType
compiler/UimpleModel/getUimpleClass
compiler/UimpleModel/getUimpleClasses
compiler/UimpleModel/getUimpleElements
compiler/UimpleModel/getUimpleFile
compiler/UimpleModel/getUimpleInterfaces
compiler/UimpleModel/numberOfAssociations
compiler/UimpleModel/numberOfGenerates
compiler/UimpleModel/numberOfUimpleClasses
compiler/UimpleModel/numberOfUimpleInterfaces
compiler/UimpleModel/run
UimpleConsoleMain/main
UimpleConsoleMain/optParse
UimpleConsoleMain/preModelOptionProcess

Appendix I: Data Correlation

The following tables contain additional correlations between dynamic/static variables. Some of the tables have already appeared in the thesis in a shorter form. For example, Table 52 is a longer version of the MLOC correlation information found in Table 10, and Table 53 is a longer version of the McCabe correlation information in Table 9.

System name	Number of methods	Dynamic														Static			
		Fan In	Fan Out	Unique Fan In	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	EncounteredinTrace	AverageCallHierarchy	DataAccessMethod	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR	
HelloWorld	2	0.256	-0.130	0.139	-0.217	-0.141	0.091	0.165	-0.124	-0.180	0.262	-0.232	-0.232	-0.096	-0.168	0.146	-0.003	0.131	
HoverHelp	6	-0.107	-0.469	0.000	-0.497	0.200	-0.200	-0.325	0.346	1.000	0.062	-0.632	-0.632	-0.632	-0.442	-0.586	-0.598	1.000	
Clipboard	9	0.125	-0.318	0.125	-0.279	0.125	1.000	-0.305	0.125	-0.125	-0.395	-0.193	-0.134	0.125	-0.493	-0.316	-0.534	0.316	
DND	14	0.730	-0.321	0.366	-0.286	0.113	0.548	-0.171	0.829	0.679	0.136	-0.167	-0.115	0.750	-0.502	-0.187	-0.442	0.000	
TextEdit	14	0.471	-0.398	0.207	-0.719	-0.090	-0.207	0.002	0.491	0.548	0.286	-0.216	-0.211	0.468	-0.076	0.670	0.545	0.436	
Address Book	16	0.365	-0.641	0.200	-0.550	0.630	0.333	-0.622	0.333	0.488	-0.618	-0.683	-0.683	-0.683	-0.582	0.011	-0.405	0.067	
BrowserExample	16	0.499	-0.471	0.557	-0.622	0.447	0.447	-0.464	0.447	0.447	0.218	-0.448	-0.448	0.447	-0.445	-0.370	0.186	-0.346	
ImageViewer	20	0.277	-0.366	-0.186	-0.645	0.076	-0.333	-0.130	0.621	0.420	0.241	-0.253	-0.600	-0.600	-0.235	-0.177	-0.124	0.169	
ImageAnalyzer	26	0.095	-0.321	0.175	-0.531	0.185	0.022	-0.141	0.455	0.312	-0.095	-0.188	-0.187	0.101	-0.280	0.116	-0.211	0.038	
LayoutExample	35	-0.174	-0.555	-0.221	-0.742	0.322	-0.331	-0.327	-0.368	0.500	-0.360	0.199	0.200	0.195	-0.438	0.025	-0.461	0.000	
TraceToUCL on jhotdraw.TF	66	0.198	-0.394	0.334	-0.359	0.529	0.046	-0.340	0.371	0.046	0.113	-0.399	-0.682	0.229	-0.691	-0.737	-0.773	-0.155	
PaintExample	104	0.171	-0.209	0.103	-0.377	0.046	0.294	-0.033	0.119	0.219	0.265	0.055	0.056	-0.035	-0.199	-0.222	-0.222	-0.036	
Graphics	213	0.256	-0.130	0.139	-0.217	-0.141	0.091	0.165	-0.124	-0.180	0.262	-0.232	-0.232	-0.096	-0.168	0.146	-0.003	0.131	
Control	294	-0.046	-0.246	-0.022	-0.253	0.073	0.079	-0.156	0.186	0.804	-0.105	-0.034	-0.034	-0.031	-0.415	-0.241	-0.337	0.037	
Lucimnav.TP.TP Nov26AfterE	305	0.104	-0.176	0.030	-0.263	-0.337	0.105	-0.076	0.060	0.274	0.085	0.013	0.014	0.053	-0.066	0.046	0.018	0.250	
Uimple compiling BCMS.TP1	351	0.052	-0.073	0.030	-0.331	0.165	0.094	-0.024	0.072	0.579	-0.009	0.004	0.006	-0.004	-0.078	-0.293	-0.360	-0.271	
Used.TP.TP open file Nov12	458	-0.049	-0.119	0.062	-0.583	0.192	-0.018	-0.088	0.054	0.204	-0.006	0.044	0.044	0.040	-0.099	-0.112	-0.114	-0.247	
JhotDraw	566	0.057	-0.174	-0.015	-0.493	0.150	0.049	-0.067	0.139	0.249	0.057	0.086	0.086	0.111	-0.252	-0.200	-0.226	-0.207	
		0.066	-0.178	0.044	-0.405	0.078	0.067	-0.075	0.102	0.324	0.039	-0.015	-0.024	0.034	-0.206	-0.147	-0.203	-0.083	

Table 44 Correlation of static and dynamic variables with leaf.

System name	Number of methods	Dynamic													Static			
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	EncounteredInTrace	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	0.034	-0.039	0.157	-0.165	0.001	0.312	0.011	0.422	-0.124	0.081	-0.039	-0.039	0.005	-0.299	-0.126	-0.332	-0.045
HoverHelp	6	0.795	-0.068	0.739	-0.701	0.416	0.730	0.526	0.346	0.346	0.927	-0.414	-0.414	-0.385	-0.378	-0.658	-0.549	0.346
Clipboard	9	-0.125	0.238	-0.125	0.122	-0.125	0.125	0.227	0.125	0.125	0.395	0.192	0.133	1.000	0.308	0.316	-0.205	0.395
BrowserExample	14	0.988	-0.211	0.914	-0.278	0.200	1.000	-0.192	-0.200	0.447	0.878	-0.201	-0.201	1.000	-0.231	-0.205	-0.166	-0.155
DND	14	0.931	-0.382	0.684	-0.310	0.144	0.698	-0.190	0.663	0.829	0.445	-0.213	-0.147	0.956	-0.350	0.226	-0.133	0.477
TextEditor	14	0.527	-0.195	0.101	-0.353	-0.044	-0.101	0.218	0.613	0.491	0.316	-0.105	-0.104	0.494	-0.055	0.177	0.204	0.651
Address Book	16	0.730	-0.214	0.067	-0.183	-0.095	-0.067	-0.158	-0.098	0.333	0.228	-0.228	-0.228	-0.228	-0.101	0.308	0.194	-0.333
JavaViewer	20	0.702	0.165	-0.026	-0.374	-0.050	-0.206	0.488	0.296	0.621	0.809	-0.222	-0.622	-0.621	-0.015	0.241	0.052	0.160
ImageAnalyser	26	0.659	0.225	0.519	-0.317	0.183	0.376	0.556	0.010	0.455	0.581	-0.186	-0.185	0.664	-0.268	0.008	-0.222	0.478
LayoutExample	35	0.510	0.756	0.643	0.546	-0.280	0.693	0.652	-0.222	-0.368	0.705	-0.612	-0.612	-0.599	-0.003	0.040	0.176	0.055
TraceToUCM on jhdraw_tf	66	0.267	-0.146	0.124	-0.133	0.153	-0.124	-0.078	-0.124	0.371	0.168	-0.173	-0.259	0.739	-0.422	-0.472	-0.576	0.311
PaintExample	104	0.550	0.327	0.351	-0.074	0.213	0.453	0.521	-0.261	0.119	0.537	0.111	0.112	0.164	-0.240	-0.239	-0.269	0.293
Graphics	213	0.034	-0.039	0.157	-0.165	0.001	0.312	0.011	0.422	-0.124	0.081	-0.039	-0.039	0.005	-0.299	-0.126	-0.332	-0.045
Control	292	0.230	0.287	0.298	0.055	-0.155	0.662	0.328	0.156	0.186	0.439	-0.157	-0.157	-0.120	-0.049	-0.007	0.094	0.070
ucmnav_TP1P_Nov26AfterEx	305	0.272	0.045	0.232	-0.045	0.193	0.343	0.177	0.099	0.060	0.302	-0.002	-0.002	0.116	0.047	0.220	0.124	-0.292
Umlle compiling BCMS_TP1	351	0.132	-0.081	0.187	-0.164	0.048	0.161	0.019	0.129	0.072	0.623	-0.003	0.000	0.130	-0.043	0.041	-0.002	0.016
Uced_TP1P_open_file_Nov12	458	0.287	0.254	0.207	-0.080	0.147	0.347	0.284	0.064	0.054	0.704	-0.015	-0.015	0.192	-0.080	0.087	0.102	0.015
JHdDraw	566	0.323	0.251	0.291	-0.143	-0.006	0.364	0.346	0.230	0.139	0.366	-0.018	-0.018	0.113	-0.132	-0.017	-0.051	-0.109
		0.275	0.146	0.249	-0.096	0.050	0.351	0.232	0.139	0.102	0.456	-0.046	-0.050	0.114	-0.104	0.016	-0.023	-0.018

Table 45 Correlation of static and dynamic variables with AverageCallHierarchy.

System name	Number of methods	Dynamic										Static						
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	EncounteredinTrace	AverageCallHierarchy	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCummulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	0.074	-0.148	-0.071	-0.204	0.337	0.074	-0.004	0.422	-0.180	0.100	-0.033	-0.033	-0.019	-0.536	-0.411	-0.679	-0.596
HoverHelp	6	-0.107	-0.469	0.000	-0.497	0.200	-0.200	-0.325	0.346	1.000	0.062	-0.632	-0.632	-0.632	-0.442	-0.586	-0.598	1.000
Clipboard	9	-1.000	-0.159	-1.000	0.035	-1.000	-0.125	-0.227	0.125	-0.125	-0.395	0.673	-0.103	0.125	-0.466	-0.316	-0.534	-0.395
DND	14	0.496	-0.218	0.248	-0.194	0.077	0.372	-0.116	0.563	0.679	-0.069	-0.114	-0.078	0.509	-0.341	-0.127	-0.301	0.000
TextEditor	14	0.073	-0.218	0.113	-0.394	-0.049	-0.113	-0.135	0.613	0.548	-0.129	-0.118	-0.116	0.041	-0.308	-0.166	-0.196	0.239
BrowserExample	15	-0.159	-0.211	-0.083	-0.278	0.200	-0.200	-0.215	-0.200	0.447	-0.293	-0.201	-0.201	-0.200	-0.272	-0.321	-0.415	0.031
Address Book	16	0.000	-0.313	0.098	-0.268	0.754	-0.098	-0.318	-0.098	0.488	-0.429	-0.333	-0.333	-0.333	-0.341	-0.150	-0.335	0.293
ImageViewer	20	-0.103	-0.154	-0.078	-0.271	-0.096	-0.140	-0.168	0.296	0.420	-0.045	-0.106	-0.059	-0.420	-0.262	-0.195	-0.261	-0.118
ImageAnalyzer	26	-0.085	-0.100	-0.047	-0.166	0.058	-0.083	-0.116	0.010	0.312	-0.106	-0.059	-0.058	-0.081	-0.108	0.093	-0.134	-0.143
LayoutExample	35	-0.096	-0.278	-0.110	-0.371	0.228	-0.166	-0.170	-0.222	0.500	-0.190	0.099	0.100	0.097	-0.298	-0.143	-0.315	-0.138
TraceToUCM on JhotDraw TPTP Sept6After	66	-0.066	-0.018	0.015	-0.017	0.075	1.000	-0.034	-0.124	0.046	-0.073	-0.021	-0.034	-0.076	-0.039	-0.045	-0.015	-0.138
PaintExample	104	-0.143	-0.186	0.023	-0.266	-0.055	0.141	-0.199	-0.261	0.219	-0.052	0.058	0.059	-0.071	-0.261	-0.355	-0.309	-0.134
Graphics	213	0.074	-0.148	-0.071	-0.204	0.337	0.074	-0.004	0.422	-0.180	0.100	-0.033	-0.033	0.034	-0.536	-0.411	-0.679	-0.596
Control	293	-0.060	-0.212	-0.092	-0.216	0.132	0.019	-0.152	0.156	0.804	-0.144	0.033	0.034	0.029	-0.422	-0.289	-0.415	-0.156
Jucmanav TPTP Nov26AfterExclusion Proce	305	0.192	-0.005	0.135	-0.125	-0.326	0.269	0.097	0.099	0.274	0.195	-0.040	-0.039	0.065	-0.183	-0.197	-0.316	-0.094
Uimple compiling BCMIS TPTP Sept6After	351	0.074	-0.051	0.209	-0.218	0.150	0.238	0.006	0.129	0.579	0.026	-0.052	-0.051	0.004	-0.059	-0.271	-0.408	-0.277
Used TPTP open file Nov12AfterExclusion	458	-0.062	-0.049	0.022	-0.106	0.186	0.050	-0.058	0.064	0.204	0.054	0.032	0.032	-0.012	-0.251	-0.209	-0.253	-0.254
JhotDraw	566	0.118	-0.006	0.118	-0.260	0.103	0.105	0.069	0.230	0.249	0.047	0.080	0.080	0.116	-0.268	-0.256	-0.307	-0.238
		0.036	-0.077	0.056	-0.194	0.088	0.134	-0.021	0.139	0.324	0.024	0.011	0.006	0.025	-0.261	-0.252	-0.349	-0.228

Table 46 Correlation of static and dynamic variables with DataAccessMethod.

System name	Number of methods	Dynamic													Static			
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistanceBetweenMethodsCalled	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCummulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	-0.053	-0.307	0.056	-0.287	0.166	-0.193	0.001	0.337	-0.141	-0.003	0.085	0.086	0.095	-0.076	-0.131	-0.342	-0.241
HoverHelp	6	0.249	0.067	0.775	-0.879	0.200	0.212	0.416	0.200	0.200	0.372	-0.334	-0.333	-0.308	0.424	-0.293	0.239	0.200
Clipboard	9	1.000	0.159	1.000	-0.035	0.125	0.227	-0.125	-1.000	0.125	0.395	-0.673	0.103	-0.125	0.466	0.316	0.534	0.395
BrowserExample	11	0.350	-0.999	0.581	-0.974	0.200	-0.999	0.200	0.200	0.447	0.293	-1.000	-1.000	0.200	0.110	0.146	0.166	0.155
DND	14	0.372	0.123	0.524	0.000	0.207	0.218	0.144	0.077	0.113	0.392	-0.683	0.066	0.181	0.355	0.190	0.492	0.000
TextEditor	14	-0.023	-0.304	0.501	-0.297	-0.501	-0.279	-0.044	-0.049	-0.090	0.029	-0.502	-0.503	-0.042	0.115	-0.150	-0.139	0.070
Address Book	16	0.209	-0.406	0.401	-0.392	-0.095	-0.395	-0.095	0.754	0.630	-0.568	-0.476	-0.476	-0.475	-0.357	-0.213	-0.449	0.286
ImageViewer	20	0.042	0.392	0.667	0.181	0.433	0.315	-0.050	-0.096	0.076	0.055	-0.410	-0.078	-0.075	0.555	0.334	0.806	0.194
ImageAnalyser	26	0.064	-0.343	0.454	-0.518	0.058	-0.174	0.183	0.058	0.185	0.073	-1.000	-1.000	0.064	0.104	0.107	0.164	-0.062
LayoutExample	35	-0.010	-0.249	0.104	-0.533	0.047	-0.096	-0.280	0.228	0.322	-0.160	-0.144	-0.142	-0.140	0.157	0.012	-0.108	-0.065
TraceToUCM on Jhotdraw TPT	66	0.072	-0.360	0.328	-0.339	0.075	-0.338	0.153	0.075	0.529	0.208	-0.376	-0.413	0.166	-0.568	-0.522	-0.595	-0.011
PaintExample	104	0.153	0.124	0.048	-0.178	0.135	0.166	0.213	-0.055	0.046	0.010	0.142	0.144	0.236	-0.291	-0.257	-0.354	0.121
Graphics	213	-0.053	-0.307	0.056	-0.287	0.166	-0.193	0.001	0.337	-0.141	-0.003	0.085	0.086	0.095	-0.076	-0.131	-0.342	-0.241
Control	289	-0.009	-0.186	0.054	-0.090	0.081	-0.097	-0.155	0.132	0.073	-0.113	0.075	0.075	0.062	-0.037	-0.129	-0.343	-0.216
Ucmnav TPTP Nov26AfterExcl	305	-0.090	-0.074	0.027	-0.001	-0.026	-0.102	0.193	-0.326	-0.337	-0.060	0.106	0.106	0.049	-0.046	-0.020	0.104	-0.423
Ucmle compiling BCMS TPTP	351	0.015	0.027	-0.006	-0.041	0.118	0.030	0.048	0.150	0.165	-0.048	0.056	0.056	0.064	0.065	-0.011	-0.118	-0.221
Uced TPTP open file Nov12At	458	0.069	0.112	0.280	-0.137	0.342	0.095	0.147	0.186	0.192	0.166	0.065	0.065	0.134	-0.140	-0.055	-0.084	-0.121
JhotDraw	566	0.003	0.010	0.141	-0.119	0.171	0.008	-0.006	0.103	0.150	-0.085	-0.037	-0.037	-0.039	-0.160	-0.138	-0.148	0.140
		0.017	-0.046	0.135	-0.131	0.150	-0.025	0.050	0.088	0.077	-0.009	0.003	0.012	0.056	-0.085	-0.092	-0.143	-0.105

Table 47 Correlation of static and dynamic variables with AverageCalledByDistance.

System name	Number of methods	Dynamic														Static			
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCummulative	MLOC	McCabe	NBD	PAR	
HelloWorld	2	0.307	-0.061	0.437	-0.052	0.166	0.242	0.312	0.074	0.091	0.304	-0.043	-0.043	0.050	-0.143	0.138	-0.067	-0.181	
HoverHelp	6	0.994	0.335	0.775	-0.267	0.200	0.882	0.730	-0.200	-0.200	0.930	0.310	0.310	0.341	-0.405	0.000	-0.239	-0.200	
Clipboard	9	0.125	-0.318	0.125	-0.279	0.125	-0.305	0.125	-0.125	1.000	-0.395	-0.193	-0.134	0.125	-0.493	-0.316	-0.534	0.316	
BrowserExample	12	0.988	-0.211	0.914	-0.278	0.200	-0.192	1.000	-0.200	0.447	0.878	-0.201	-0.201	1.000	-0.231	-0.205	-0.166	-0.155	
DND	14	0.867	-0.458	0.875	-0.366	0.207	-0.286	0.698	0.372	0.548	0.681	-0.304	-0.209	0.874	0.222	0.647	0.323	0.000	
TextEditor	14	-0.133	0.958	-1.000	0.565	-0.501	0.727	-0.101	-0.113	-0.207	-0.139	1.000	1.000	-0.097	-0.253	0.027	-0.320	0.162	
Address Book	16	0.000	-0.214	0.067	-0.183	-0.095	-0.217	-0.067	-0.098	0.333	-0.293	-0.228	-0.228	-0.228	-0.228	-0.148	-0.370	-0.333	
JavaViewer	20	0.006	0.447	0.867	0.860	0.433	0.337	-0.206	-0.140	-0.333	0.027	-0.045	0.333	0.334	0.180	0.129	0.310	0.282	
ImageAnalyzer	26	0.613	-0.059	0.866	0.422	0.058	0.350	0.376	-0.083	0.022	0.355	-0.057	-0.058	0.615	0.053	0.124	-0.032	0.322	
LayoutExample	35	0.422	0.822	0.655	0.252	0.047	0.610	0.693	-0.166	-0.331	0.599	-0.662	-0.662	-0.651	0.215	0.249	0.334	0.033	
TraceToUCM on Jhotdraw TPT	66	-0.066	-0.018	0.015	-0.017	0.075	-0.034	-0.124	1.000	0.046	-0.073	-0.021	-0.034	-0.076	-0.039	-0.045	-0.015	-0.138	
PaintExample	104	0.242	0.057	0.414	-0.027	0.135	0.174	0.453	0.141	0.294	0.440	0.077	0.077	0.158	-0.224	-0.275	-0.253	-0.023	
Graphics	213	0.307	-0.061	0.437	-0.052	0.166	0.242	0.312	0.074	0.091	0.304	-0.043	-0.043	0.050	-0.143	0.138	-0.067	-0.181	
Control	290	0.237	0.304	0.428	0.201	0.081	0.343	0.662	0.019	0.079	0.411	-0.279	-0.279	-0.258	0.010	0.014	0.077	0.011	
JUnitnav_TPTP_Nov26AfterExcl	305	0.339	0.142	0.449	0.005	-0.026	0.283	0.343	0.269	0.105	0.396	-0.007	-0.007	0.139	-0.051	-0.028	-0.054	-0.120	
Umlle compiling BCMS TPTP	351	-0.033	-0.032	0.543	-0.094	0.118	-0.046	0.161	0.238	0.094	0.003	0.059	0.060	0.187	-0.041	-0.098	-0.121	-0.263	
Used TPTP open file Nov12Af	458	0.259	0.298	0.514	0.038	0.342	0.292	0.347	0.050	-0.018	0.447	0.004	0.004	0.179	-0.084	-0.039	-0.007	-0.045	
JHotDraw	566	0.177	0.206	0.457	-0.012	0.171	0.230	0.364	0.105	0.049	0.267	-0.041	-0.041	0.048	-0.140	-0.101	-0.130	-0.041	
		0.210	0.160	0.464	0.021	0.150	0.216	0.351	0.134	0.066	0.299	-0.043	-0.039	0.074	-0.081	-0.040	-0.061	-0.083	

Table 48 Correlation of static and dynamic variables with averageCalledByDistanceBetweenMethodsCalled.

System name	Number of methods	Dynamic														Static			
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR	
HelloWorld	2	0.880	0.467	0.253	0.062	-0.193	0.242	0.011	-0.004	0.165	0.790	0.048	0.047	0.208	-0.012	0.042	0.031	0.026	
HoverHelp	6	0.860	0.735	0.706	-0.112	0.212	0.882	0.526	-0.325	-0.325	0.764	0.427	0.427	0.454	-0.140	0.092	-0.238	-0.325	
Clipboard	9	0.227	0.998	0.227	0.863	0.227	-0.305	0.227	-0.227	-0.305	-0.072	0.484	0.879	0.227	0.575	0.419	-0.040	-0.468	
BrowserExample	13	-0.343	1.000	-0.574	0.980	-0.999	-0.192	-0.192	-0.215	-0.464	-0.268	0.999	0.999	-0.192	-0.075	-0.112	-0.159	-0.120	
DND	14	-0.183	0.976	-0.136	0.929	0.218	-0.286	-0.190	-0.116	-0.171	-0.364	0.537	0.952	-0.241	0.300	0.060	-0.220	0.040	
TextEdit	14	0.543	0.699	-0.727	0.298	-0.279	0.727	0.218	-0.135	0.002	0.551	0.725	0.725	0.571	-0.013	0.373	-0.060	0.321	
Address Book	16	-0.161	0.997	-0.077	0.830	-0.395	-0.217	-0.158	-0.318	-0.622	0.627	0.666	0.666	0.666	0.525	-0.141	0.413	0.063	
JavaViewer	20	0.685	0.863	0.451	0.346	0.315	0.337	0.488	-0.168	-0.130	0.832	-0.115	-0.156	-0.154	0.647	0.773	0.640	-0.006	
ImageAnalyser	26	0.798	0.794	0.341	0.057	-0.174	0.350	0.556	-0.116	-0.141	0.930	0.174	0.173	0.795	-0.128	-0.090	-0.101	0.674	
LayoutExample	35	0.957	0.782	0.949	0.167	-0.096	0.610	0.652	-0.170	-0.327	0.979	-0.133	-0.133	-0.114	0.015	0.107	0.175	0.312	
TraceToUCM on Jhotdraw TPT	66	0.170	0.969	-0.953	0.954	-0.338	-0.034	-0.078	-0.034	-0.340	0.100	0.932	0.702	-0.102	0.716	0.466	0.312	0.002	
PaintExample	104	0.816	0.849	0.368	0.115	0.166	0.174	0.521	-0.199	-0.033	0.500	0.044	0.044	0.089	-0.059	-0.024	0.010	0.168	
Graphics	213	0.880	0.467	0.253	0.062	-0.193	0.242	0.011	-0.004	0.165	0.790	0.048	0.047	0.208	-0.012	0.042	0.031	0.026	
Control	291	0.878	0.565	0.807	0.296	-0.097	0.343	0.328	-0.152	-0.156	0.958	-0.148	-0.149	-0.121	0.154	0.085	0.239	0.131	
Jucmnav TPTP Nov26AfterExcl	305	0.691	0.859	0.400	0.550	-0.102	0.283	0.177	0.097	-0.076	0.700	0.011	0.011	0.299	0.252	0.191	-0.014	0.015	
Uimple compiling BCM5 TPTP	351	0.624	0.780	-0.026	0.286	0.030	-0.046	0.019	0.006	-0.024	0.117	0.022	0.016	-0.008	-0.001	-0.025	-0.045	0.047	
Uced TPTP open file Nov12Aft	458	0.953	0.952	0.330	0.061	0.095	0.292	0.284	-0.058	-0.088	0.477	0.026	0.026	0.085	-0.019	0.040	0.039	0.017	
JHotDraw	566	0.839	0.821	0.309	0.041	0.008	0.230	0.346	0.069	-0.067	0.523	-0.087	-0.087	0.030	0.013	0.008	-0.017	0.002	
		0.779	0.791	0.295	0.223	-0.026	0.216	0.232	-0.021	-0.074	0.544	0.021	0.017	0.075	0.074	0.063	0.040	0.047	

Table 49 Correlation of static and dynamic variables with EncounteredInTrace.

System name	Number of methods	Dynamic													Static			
		FanIn	FanOut	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetween MethodsCalled	EncounteredinTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCummulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	0.302	-0.031	-0.033	0.056	0.437	0.253	0.157	-0.071	0.139	0.229	-0.018	-0.018	0.014	-0.019	0.093	0.052	-0.011
HoverHelp	6	0.802	0.259	-0.740	0.775	0.775	0.706	0.739	0.000	0.000	0.841	-0.015	-0.015	0.021	0.012	-0.189	0.000	0.000
BrowserExample	7	0.966	-0.590	-0.635	0.581	0.914	-0.574	0.914	-0.083	0.557	0.851	-0.582	-0.582	0.914	-0.146	-0.109	-0.069	-0.064
Clipboard	9	1.000	0.159	-0.035	1.000	0.125	0.227	-0.125	-1.000	0.125	0.395	-0.673	0.103	-0.125	0.466	0.316	0.534	0.395
DND	14	0.890	-0.323	-0.279	0.524	0.875	-0.136	0.684	0.248	0.366	0.802	-0.489	-0.144	0.831	0.338	0.750	0.559	0.304
TextEditor	14	0.133	-0.958	-0.565	0.501	-1.000	-0.727	0.101	0.113	0.207	0.139	-1.000	-1.000	0.097	0.253	-0.027	0.320	-0.162
Address Book	16	0.730	-0.134	0.268	0.401	0.067	-0.077	0.067	0.098	0.200	-0.228	-0.298	-0.296	-0.293	0.179	-0.034	0.088	-0.200
JavaViewer	20	0.103	0.531	0.579	0.667	0.867	0.451	-0.026	-0.078	-0.186	0.149	-0.453	0.183	0.187	0.276	0.205	0.461	0.157
ImageAnalyser	26	0.740	-0.200	-0.020	0.454	0.866	0.341	0.519	-0.047	0.175	0.457	-0.454	-0.454	0.742	0.025	0.035	-0.018	0.311
LayoutExample	35	0.943	0.667	0.054	0.104	0.655	0.949	0.643	-0.110	-0.221	0.930	-0.284	-0.284	-0.264	-0.022	0.039	0.130	0.260
TraceToUCM on jhotdraw TPTF	66	0.066	-0.981	-1.000	0.328	0.015	-0.953	0.124	0.015	0.334	0.073	-0.982	-0.734	0.076	-0.753	-0.480	-0.297	0.011
PaintExample	104	0.435	0.189	-0.005	0.048	0.414	0.368	0.351	0.023	0.103	0.604	0.036	0.037	0.033	-0.089	-0.066	-0.035	0.017
Graphics	213	0.302	-0.031	-0.033	0.056	0.437	0.253	0.157	-0.071	0.139	0.229	-0.018	-0.018	0.014	-0.019	0.093	0.052	-0.011
Control	285	0.901	0.123	0.068	0.054	0.428	0.807	0.298	-0.092	-0.022	0.847	-0.235	-0.235	-0.219	-0.012	0.017	0.051	0.124
lucmnav TPTP Nov26AfterExcl	305	0.432	0.234	-0.008	0.027	0.449	0.400	0.232	0.135	0.030	0.501	0.017	0.017	0.188	0.028	0.145	0.052	-0.056
Uimple compiling BCMS TPTP	351	-0.014	-0.023	-0.062	-0.006	0.543	-0.026	0.187	0.209	0.030	-0.012	0.010	0.011	0.176	-0.031	-0.091	-0.041	-0.184
Uced TPTP open file Nov12Af	458	0.269	0.360	-0.019	0.280	0.514	0.330	0.207	0.022	0.062	0.344	0.033	0.033	0.137	-0.075	-0.039	-0.008	-0.029
JHotDraw	566	0.259	0.254	0.012	0.141	0.457	0.309	0.291	0.118	-0.015	0.285	-0.030	-0.030	0.040	-0.078	-0.055	-0.063	-0.012
		0.348	0.145	-0.037	0.134	0.463	0.296	0.247	0.057	0.042	0.359	-0.077	-0.060	0.072	-0.051	-0.013	-0.001	-0.018

Table 50 Correlation of static and dynamic variables with fan-in.

System name	Number of methods	Dynamic													Static			
		FanIn	FanOut	UniqueFanIn	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	MLOC	McCabe	NBD	PAR
HelloWorld	2	-0.060	0.243	-0.033	-0.287	-0.052	0.062	-0.165	-0.204	-0.217	-0.038	0.051	0.051	0.016	0.261	0.177	0.383	0.043
HoverHelp	6	-0.349	0.254	-0.740	-0.879	-0.267	-0.112	-0.701	-0.497	-0.497	-0.545	0.617	0.617	0.591	-0.088	0.615	0.091	-0.497
BrowserExample	8	-0.421	0.981	-0.635	-0.974	-0.278	0.980	-0.278	-0.278	-0.622	-0.272	0.974	0.974	-0.278	0.081	0.034	-0.144	0.022
Clipboard	9	-0.035	0.877	-0.035	-0.035	-0.279	0.863	0.122	0.035	-0.279	-0.387	0.757	0.982	0.122	0.195	0.387	-0.161	-0.486
DND	14	-0.333	0.943	-0.279	0.000	-0.366	0.929	-0.310	-0.194	-0.286	-0.466	0.712	0.973	-0.353	0.198	0.006	-0.276	0.000
TextEditor	14	-0.354	0.647	-0.565	-0.297	0.565	0.298	-0.353	-0.394	-0.719	-0.226	0.583	0.570	-0.337	-0.325	-0.410	-0.609	-0.050
Address Book	16	-0.309	0.841	-0.268	-0.392	-0.183	0.830	-0.183	-0.268	-0.550	0.461	0.641	0.640	0.640	0.198	-0.012	0.175	0.099
JavaViewer	20	-0.092	0.526	0.579	0.181	0.860	0.346	-0.374	-0.271	-0.645	-0.043	0.375	0.485	0.484	0.262	0.239	0.283	0.227
ImageAnalyser	26	-0.124	0.216	-0.020	-0.518	0.422	0.057	-0.317	-0.166	-0.531	-0.095	0.521	0.519	-0.126	0.137	0.088	0.024	-0.046
LayoutExample	35	-0.022	0.520	0.054	-0.533	0.252	0.167	0.546	-0.371	-0.742	0.203	-0.401	-0.403	-0.404	0.209	-0.103	0.395	0.000
TraceToUCM on Jhotdraw TPT	66	-0.071	0.982	-1.000	-0.339	-0.017	0.954	-0.133	-0.017	-0.359	-0.075	0.984	0.746	-0.082	0.765	0.496	0.316	-0.007
PaintExample	104	-0.074	0.252	-0.005	-0.178	-0.027	0.115	-0.074	-0.266	-0.377	0.062	0.053	0.050	0.109	0.306	0.180	0.166	0.035
Graphics	213	-0.060	0.243	-0.033	-0.287	-0.052	0.062	-0.165	-0.204	-0.217	-0.038	0.051	0.051	0.016	0.261	0.177	0.383	0.043
Control	286	0.050	0.528	0.068	-0.090	0.201	0.296	0.055	-0.216	-0.253	0.215	-0.135	-0.135	-0.119	0.165	0.077	0.220	0.100
Jucmnav TPTP Nov26AfterExcl	305	-0.047	0.775	-0.008	-0.001	0.005	0.550	-0.045	-0.125	-0.263	-0.027	0.024	0.024	0.000	0.107	-0.006	0.004	-0.037
Uimple compiling BCWS TPTP	351	-0.021	0.383	-0.062	-0.041	-0.094	0.286	-0.164	-0.218	-0.331	-0.072	0.013	-0.018	-0.037	0.695	0.525	0.337	0.186
Used TPTP open file Nov12Af	458	0.024	0.093	-0.019	-0.137	0.038	0.061	-0.080	-0.106	-0.583	0.008	-0.075	-0.076	-0.077	0.151	0.088	0.076	0.126
JHotDraw	566	-0.071	0.144	0.012	-0.119	-0.012	0.041	-0.143	-0.260	-0.493	-0.082	-0.017	-0.018	-0.052	0.350	0.151	0.170	0.280
		-0.035	0.351	-0.037	-0.130	0.021	0.221	-0.096	-0.194	-0.404	-0.012	0.022	0.018	-0.043	0.300	0.170	0.177	0.123

Table 51 Correlation of static and dynamic variables with fan-out.

System name	Number of methods	Dynamic														Static		
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetween MethodsCalled	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCummulative	MLOC	NBD	PAR
HelloWorld	2	-0.015	0.116	0.093	0.177	-0.131	0.138	0.042	-0.126	-0.411	0.146	0.000	-0.294	-0.294	-0.242	0.483	0.496	0.296
HoverHelp	6	-0.091	0.294	-0.189	0.615	-0.293	0.000	0.092	-0.658	-0.586	-0.586	-0.348	0.926	0.926	0.917	0.022	0.612	-0.586
Clipboard	9	0.316	0.402	0.316	0.387	0.316	-0.316	0.419	0.316	-0.316	-0.316	0.350	0.050	0.386	0.316	0.663	0.637	0.350
BrowseExample	10	-0.172	-0.108	-0.109	0.034	0.146	-0.205	-0.112	-0.205	-0.321	-0.370	0.271	-0.144	-0.144	-0.205	0.991	0.558	0.890
DND	14	0.460	-0.045	0.750	0.006	0.190	0.647	0.060	0.226	-0.127	-0.187	0.569	-0.094	0.061	0.447	0.733	0.709	0.245
TextEditor	14	0.645	-0.116	-0.027	-0.410	-0.150	0.027	0.373	0.177	-0.166	0.670	0.594	0.020	0.024	0.664	0.165	0.723	0.375
Address Book	16	0.188	-0.153	-0.034	-0.012	-0.213	-0.148	0.141	0.308	-0.150	0.011	-0.206	0.162	0.163	0.162	0.254	0.625	-0.331
JavaViewer	20	0.243	0.865	0.205	0.239	0.334	0.129	0.773	0.241	-0.195	-0.177	0.528	-0.057	-0.177	-0.176	0.907	0.759	0.072
ImageAnalyzer	26	-0.192	0.050	0.035	0.088	0.107	0.124	-0.090	0.008	0.093	0.116	-0.145	-0.105	-0.105	-0.191	0.291	0.647	0.073
LayoutExample	35	0.027	0.246	0.039	-0.103	0.012	0.249	0.107	0.040	-0.143	0.025	0.056	0.013	0.013	0.015	0.422	0.399	0.627
TraceToUCM on Jhotdraw TPT	66	-0.137	0.505	-0.480	0.496	-0.522	-0.045	0.466	-0.472	-0.045	-0.737	-0.183	0.544	0.798	-0.306	0.888	0.844	-0.077
PaintExample	104	-0.091	0.045	-0.066	0.180	-0.257	-0.275	-0.024	-0.239	-0.355	-0.222	-0.122	0.006	0.006	-0.052	0.574	0.736	0.055
Graphics	213	-0.015	0.116	0.093	0.177	-0.131	0.138	0.042	-0.126	-0.411	0.146	0.000	-0.294	-0.294	-0.242	0.483	0.496	0.296
Control	288	0.027	0.132	0.017	0.077	-0.129	0.014	0.085	-0.007	-0.289	-0.241	0.077	0.001	0.001	0.005	0.523	0.578	0.044
ucmnav TPT Nov26AfterExcl	305	0.255	0.077	0.145	-0.006	-0.020	-0.028	0.191	0.220	-0.197	0.046	0.328	0.027	0.027	0.118	0.643	0.487	0.011
Uimple compiling BCMS TPT	351	-0.026	-0.011	-0.091	0.525	-0.011	-0.098	-0.025	0.041	-0.271	-0.293	-0.067	0.012	0.010	0.027	0.605	0.627	0.267
Used TPT open file Nov12Af	458	0.014	0.063	-0.039	0.088	-0.055	-0.039	0.040	0.087	-0.209	-0.112	0.026	-0.025	-0.025	0.051	0.701	0.701	0.281
JhotDraw	566	-0.021	0.037	-0.055	0.151	-0.138	-0.101	0.008	-0.017	-0.256	-0.200	0.076	-0.036	-0.036	-0.039	0.699	0.766	0.090
		0.027	0.079	-0.013	0.170	-0.092	-0.039	0.063	0.016	-0.252	-0.146	0.062	-0.017	-0.010	-0.005	0.628	0.647	0.156

Table 52 Correlation of static and dynamic variables with McCabe.

System name	Number of methods	Dynamic														Static		
		FanIn	FanOut	UniqueFanIn	UniqueFanOut	averageCalledByDistance	averageCalledByDistanceBetweenMethodsCalled	EncounteredInTrace	AverageCallHierarchy	DataAccessMethod	Leaf	TotalSegmentPresence	CpuUsageAverage	CpuUsageBase	CpuUsageCumulative	McCabe	NBD	PAR
HelloWorld	2	-0.064	0.094	-0.019	0.261	-0.076	-0.143	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	-0.040	-0.054	0.483	0.570	0.473
HoverHelp	6	-0.427	0.305	0.012	-0.088	0.424	-0.405	-0.140	-0.378	-0.442	-0.442	-0.381	-0.127	-0.127	-0.130	0.022	0.595	-0.442
BrowseExample	9	-0.203	-0.071	-0.146	0.081	0.110	-0.231	-0.075	-0.231	-0.272	-0.445	0.256	-0.108	-0.108	-0.231	0.991	0.446	0.932
Clipboard	9	0.466	0.550	0.466	0.195	0.466	-0.493	0.575	0.308	-0.466	-0.493	0.678	-0.201	0.200	0.308	0.663	0.593	0.066
DND	14	-0.050	0.292	0.338	0.198	0.355	0.222	0.300	-0.350	-0.341	-0.502	0.225	-0.110	0.204	-0.129	0.733	0.733	-0.225
TextEditor	14	0.152	-0.145	0.253	-0.325	0.115	-0.253	-0.013	-0.055	-0.308	-0.076	0.228	-0.260	-0.254	0.148	0.165	0.437	-0.213
Address Book	16	0.053	0.512	0.179	0.198	-0.357	-0.228	0.525	-0.101	-0.341	-0.582	0.346	0.695	0.696	0.697	0.254	0.720	-0.077
ImageViewer	20	0.102	0.794	0.276	0.262	0.555	0.180	0.647	-0.015	-0.262	-0.235	0.318	-0.099	-0.077	-0.076	0.907	0.879	0.009
ImageAnalyser	26	-0.148	-0.055	0.025	0.137	0.104	0.053	-0.128	-0.268	-0.108	-0.280	-0.143	-0.102	-0.102	-0.150	0.291	0.504	-0.250
LayoutExample	35	-0.093	0.242	-0.022	0.209	0.157	0.215	0.015	-0.003	-0.298	-0.438	-0.039	-0.155	-0.155	-0.156	0.422	0.702	0.046
TraceToUCM on Jhotdraw TPTT	66	-0.166	0.766	-0.753	0.765	-0.568	-0.039	0.716	-0.422	-0.039	-0.691	-0.185	0.787	0.881	-0.271	0.888	0.743	-0.106
PaintExample	104	-0.090	-0.011	-0.089	0.306	-0.291	-0.224	-0.059	-0.240	-0.261	-0.199	-0.176	-0.057	-0.057	-0.096	0.574	0.711	0.016
Graphics	213	-0.064	0.094	-0.019	0.261	-0.076	-0.143	-0.012	-0.299	-0.536	-0.168	-0.072	-0.040	-0.040	-0.054	0.483	0.570	0.473
Control	287	-0.003	0.325	-0.012	0.165	-0.037	0.010	0.154	-0.049	-0.422	-0.415	0.095	-0.018	-0.018	-0.016	0.523	0.553	0.056
Jucmnav_TPTT_Nov26AfterExcl	305	0.094	0.274	0.028	0.107	-0.046	-0.051	0.252	0.047	-0.183	-0.066	0.134	0.028	0.029	0.057	0.643	0.335	0.146
Uimple_compiling_BCMS_TPTT	351	-0.006	0.004	-0.031	0.695	0.065	-0.041	-0.001	-0.043	-0.059	-0.078	-0.029	0.005	0.005	-0.009	0.605	0.270	0.113
Used TPTT open file_Nov12Af	458	-0.028	-0.009	-0.075	0.151	-0.140	-0.084	-0.019	-0.080	-0.251	-0.099	-0.083	-0.039	-0.039	-0.029	0.701	0.640	0.232
JHotDraw	566	-0.030	0.054	-0.078	0.350	-0.160	-0.140	0.013	-0.132	-0.268	-0.252	0.005	-0.008	-0.008	-0.028	0.699	0.591	0.103
		-0.016	0.125	-0.051	0.299	-0.085	-0.081	0.074	-0.104	-0.261	-0.205	-0.002	0.005	0.011	-0.023	0.627	0.530	0.142

Table 53 Correlation of static and dynamic variables with MLOC.