Generating Software Documentation in Use Case Maps from Filtered Execution Traces

Conference Paper · October 2015

DOI: 10.1007/978-3-319-24912-4_13

CITATIONS
7

READS
229

3 authors:

Edna Braun
University of Ottawa
11 PUBLICATIONS **265** CITATIONS

SEE PROFILE

Daniel Amyot
University of Ottawa
265 PUBLICATIONS **5,055** CITATIONS

SEE PROFILE

Timothy Lethbridge
University of Ottawa
208 PUBLICATIONS **5,661** CITATIONS

SEE PROFILE

Generating Software Documentation in Use Case Maps from Filtered Execution Traces

Edna Braun, Daniel Amyot, and Timothy Lethbridge

School of EECS, University of Ottawa, Canada
{ebraun,damyot,tcl}@eecs.uottawa.ca

Abstract. One of the main issues in software maintenance is the time and effort needed to understand software. Software documentation and models are often incomplete, outdated, or non-existent, in part because of the cost and effort involved in creating and continually updating them. In this paper, we describe an innovative technique for automatically extracting and visualizing software behavioral models from execution traces. Lengthy traces are summarized by filtering out low-level software components via algorithms that utilize static and dynamic data. Eight such algorithms are compared in this paper. The traces are visualized using the Use Case Map (UCM) scenario notation. The resulting UCM diagrams depict the behavioral model of software traces and can be used to document the software. The tool-supported technique is customizable through different filtering algorithms and parameters, enabling the generation of documentation and models at different levels of abstraction.

Keywords: Feature location · Software documentation · Trace summarization · Use Case Map · Utility · Visualization

1 Introduction

Understanding software during maintenance activities is often very difficult due to incomplete, outdated, or non-existent documentation. In the absence of up-to-date documentation, programmers frequently need to extract structural and behavioral information directly from the code. A number of techniques and methods have been developed in order to facilitate program comprehension [7,17,18], and several others, especially focusing on the use of dynamic analysis, are compared in a survey from Cornelissen et al. [8]. Feature location and comprehension approaches [11,13,21,22,26,25,28,27] have also been investigated by many researchers, and several others are summarized in a survey by Dit et al. [9].

In this paper, we propose a new approach to extract high-level behavioral models of a software feature by using execution traces. 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. [13]. A feature represents a functionality that is defined by requirements and accessible to developers and users [9]. As execution traces are often too lengthy for comprehension, we simplify them 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 a metric 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 filtering out utilities, we are left with a much smaller trace. In order to visualize reduced traces, we selected the Use Case Map (UCM) notation [6,16], a standard scenario language used to specify requirements of dynamic systems and explain their emergent behavior. The behavioral model is then displayed as UCM diagrams, which combine structure and sequences of activities. The abstracted diagrams can be used as documentation that summarizes feature behavior. The approach offers parameters that enable the selection of how much information to preserve in the traces (Fig. 1).

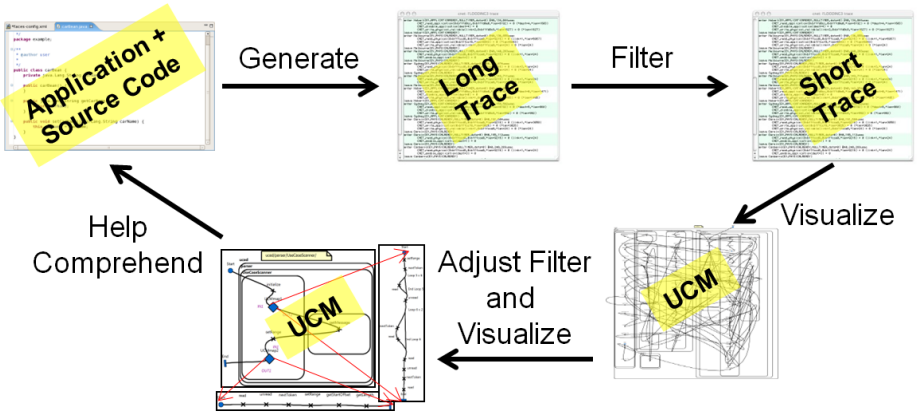


Fig. 1. From source code to traces and behavioral models

Section 2 of this paper gives background on UCM with a mapping from traces, and definitions of static and dynamic metrics used in our approach. Section 3 presents utilityhood functions used by eight algorithms for our comparative analysis. Section 4 highlights our new automated approach (TraceToUCM) for converting execution traces to UCMs. Sections 5 to 7 give an illustrative example, followed by a discussion and conclusions.

2 Mapping and Metrics

This section gives a brief summary of the UCM scenario modeling notation for reverse-engineered behavioral models, with a mapping from execution traces.

Then, it introduces the static and dynamic information used by our utility filtering algorithms.

2.1 Mapping Traces to Use Case Maps

The UCM notation is part of the User Requirements Notation (URN) [2,16], an international standard used for the elicitation, analysis, specification, and validation of requirements. UCM has first been suggested for trace visualization by Amyot et al. in [1].

We chose Use Case Maps 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 Message Sequence Charts and UML sequence diagrams, which are often used for trace visualization [26], UCMs abstract from the inter-component communication to focus on the business/feature logic. Using jUCMNav [20], an Eclipse-based tool for URN modeling, analysis and transformations, we are however able to generate sequence diagrams (with synthetic messages) from UCMs, enabling one to visualize the scenarios represented in two different formats.

As in UML activity diagrams, UCMs can integrate many scenarios with operators for looping and forking/joining alternative or concurrent paths. Complex maps can also be decomposed into sub-maps (through stubs, shown as diamonds). Stubs may contain multiple sub-maps, allowing for flexible integration and exploration of scenarios that have overlapping parts. The various scenario elements can be bound to components (shown as rectangles). The latter can have sub-components, enabling structures to be visualized in two dimensions, in a compact way.

Each item in a filtered execution trace is visually communicated using UCM. To translate execution traces to UCM maps, Table 1 provides a mapping from trace elements (in a formalized trace format [4]) to UCM notation elements. This is an extension of a preliminary informal mapping we studied in [14].

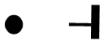
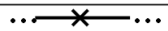
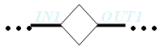
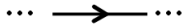
2.2 Static Data Metrics

Static analysis examines a program's code to derive properties that hold for all executions [3]. The static analysis is performed without actually executing programs. The relevant static data metrics collected here are the following:

- *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 code blocks.
- *McCabe Cyclomatic Complexity (McCabe)*: Number of flows through a piece of code [19].

This data is collected via Metrics 1.3.6 [23], an Eclipse plug-in used to gather static data on software (in Java) at the package, class, and method levels.

Table 1. Mapping of trace elements to UCM elements

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)	method {2} 
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.	

2.3 Dynamic Data Metrics

Dynamic analysis derives properties that hold true for one or more executions by examination of the running program through instrumentation. This can provide useful information about the behavior of programs for the specific input parameters that are entered. We use the Eclipse TPTP tracer and Java profiler [12] to collect execution traces and CPU usage. An execution trace contains the list of each method that was called, in order of calls, and the depth of the call tree. We process the execution trace and derive more detailed information for each of the method calls:

- *Fan-in*: Number of methods that called this method.
- *Fan-out*: Number of methods this method called.
- *UniqueFanin* (array): Unique set of methods that called this method. The length of the array is the fan-in.
- *NumberOfTimesCalledBy* (array): List of all methods that this method has been called by (contains duplicates).
- *UniqueFanout* (array): Unique set of methods that this method called. The length of the array is the fan-out.
- *NumberOfTimesMethodsCalled* (array): List of all methods that this method called (contains duplicates).
- *TotalSegmentPresence*: Total number of trace segments this method was found in, based on Dugerdil and Jossi's approach [10].
- *PercentageOfSegmentsMethodPresentIn*: Percentage of trace segments this method was found in.

– *Depth*: Depth of this method in the call hierarchy.
– *BaseTime*: Time taken to execute the invocation of a method.
– *CpuUsageAverage*: Base time divided by the number of calls.
– *CumulativeCpuTime*: Amount of CPU time spent in a method accumulated from all invocations.

3 Utilityhood Algorithms

We use different combinations of the static and dynamic data we have collected to develop a number of algorithms to calculate utilityhood functions ($U(r)$) for each method r in the execution traces. This selection of functions is based on preliminary experiments involving 18 Java open source systems used to determine the most promising metrics (based on correlations between metrics) from which utilityhood should be computed [4]. Table 2 gives a summary of these utilityhood functions, where N is the number of unique methods in the execution trace, $RelativeMethodSize(r)$ is the size of a method r in relation to other methods in the system, and $TotalNumberOfSegments$ is the total number of segments the execution trace is broken into [4].

Table 2. Summary of utilityhood function algorithms

Name	**Utilityhood Function**
Algo. 1 [14]	$U(r) = \frac{UniqueFanin(r)}{N-1}$
Algo. 2 [15]	$U(r) = \frac{UniqueFanin(r)}{N} \times \frac{log(\frac{N}{UniqueFanout(r)+1})}{log(N)}$
Algo. 3	$U(r) = \frac{1}{Nbd(r) \times McCabe(r) \times RelativeMethodSize(r)}$
Algo. 4	$U(r) = \frac{fanout(r)}{Nbd(r) \times McCabe(r) \times RelativeMethodSize(r)}$
Algo. 5 [10]	$U(r) = \frac{SegmentPresence(r)}{TotalNumberOfSegments}$
Algo. 6 [27]	$U(r) = \frac{1}{(UniqueFanin(r) \times UniqueFanout(r))^2}$
Algo. 7	$U(r) = \frac{UniqueFanin(r)}{(N-1) \times AverageCallHierarchy \times RelativeMethodSize(r)}$
Algo. 8	$U(r) = \frac{1}{CpuUsageAverage(r) \times McCabe(r) \times RelativeMethodSize(r)}$

- *Algorithm 1*: based on earlier work done in [14]. The higher the number of callers a method has, the more likely that it will be eliminated as a low-level utility class.
- *Algorithm 2*: based on work done by Hamou-Lhadj [15]. In this case, a fan-out is also taken into consideration. If two methods `A()` and `B()` have the same number of callers but `B()` also calls out to other methods, it will rank lower on the utilityhood index.
- *Algorithm 3*: we use the McCabe value, also known as cyclomatic complexity [19], to measure the number of linearly independent paths through a method. One of the hypotheses we are testing is that methods that are simple are more likely to be utilities than methods that are more complex.
- *Algorithm 4*: we test our hypothesis 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 styles, and the size of the whole system. A method considered large in one software system might be the smallest in another.
- *Algorithm 5*: based on the work of Dugerdil and Jossi [10], who developed trace segmentation and clustering techniques to reverse-engineer software systems. While segmenting the execution trace, they remove classes present in most of the segments of the execution trace. They observe those classes "perform some utility work, not specific to any functional component". We take the trace segmentation part of their methodology and test it against the other seven algorithms.
- *Algorithm 6*: based on an algorithm developed by Wang et al. [27]. They claim that the static (syntactical) situation of a software program reflects only inaccurately the situation of the dynamic behavior of the system, using factors like actual number and type of procedure calls, as well as size of the actual transferred information. 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 subject system's functionality.
- *Algorithm 7*: we look at each method in the execution trace and calculate 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 through `Z()` $\rightarrow$ `Y()` $\rightarrow$ `A()`, which is of depth 3, and also called through `Z()` $\rightarrow$ `X()` $\rightarrow$ `W()` $\rightarrow$ `A()`, which is of depth 4, then the $AverageCallHierarchy$ would be $(3 + 4)/2 = 3.5$.
- *Algorithm 8*: tests the hypothesis that most complex methods with the highest average processor (CPU) time, and with the most lines of code, are the most important methods and should be used to include in the behavioral model of a feature.

4 Automated Approach

We developed a tool called *TraceToUCM* that takes three required inputs (an execution trace, CPU profiling information, and the metrics discussed in Section 2), with optional lists of Java methods to be explicitly included or excluded in the resulting traces, and filtering parameter information (e.g., thresholds for each algorithm). The tool generates automatically a UCM diagram using the mapping shown in Table 1 for each of our filtering algorithms, in addition to other files with intermediate or by-product information (e.g., data tables, statistics, rankings, etc.). We also created an automatic layout function to draw all the UCM diagrams, which helps keep the layout variable consistent between the diagrams obtained from different utility detection algorithms.

The following is a brief summary of the main steps taken by the automatic trace summarization tool, illustrated in Fig. 2.

1. Run the target program and collect the execution trace and CPU information (using TPTP). Use Metrics 1.3.6 for producing the various static metrics.
2. Remove methods explicitly excluded.
3. Remove duplicate loops from the trace (but preserve loop counts).
4. Remove data access methods (`.get*()` or `.set*()` with less than 2 lines of code and a complexity index less than 2).
5. Remove small methods (with fewer than 5 lines of code) and simple methods (that score 2 or less on the complexity index).
6. Integrate the static data and CPU data for the methods.
7. Calculate utilityhood for all the methods using each of the 8 algorithms.
8. Filter the traces according to each utilityhood result and generate UCMs from the resulting traces.
9. If required, re-iterate to raise or lower the level of abstraction (through the filtering information file).
10. If required, a sequence diagram can also be generated from each UCM diagram via jUCMNav.

5 Illustrative Example

We use an example to demonstrate the approach proposed in this paper. We run one feature of the software, collect the trace, reduce the execution trace and produce UCM behavioral models. We selected a Java application called *Use Case Editor (UCEd)*, version 1.6.2 [24], which is an environment for use-case-based requirements engineering. UCEd contains tools for editing use cases, such as use-case integration and use-case simulation. UCEd has 700 classes, 3284 methods, and 33,004 lines of code. We run the feature "Open a project".

We collected the execution trace and the profiling information while a file was being opened. We also collected static information on the code using Metrics [23]. We then filtered the trace and generated UCM diagrams. Table 3 highlights the size of the trace after each step (in a cumulative way) or after each algorithm

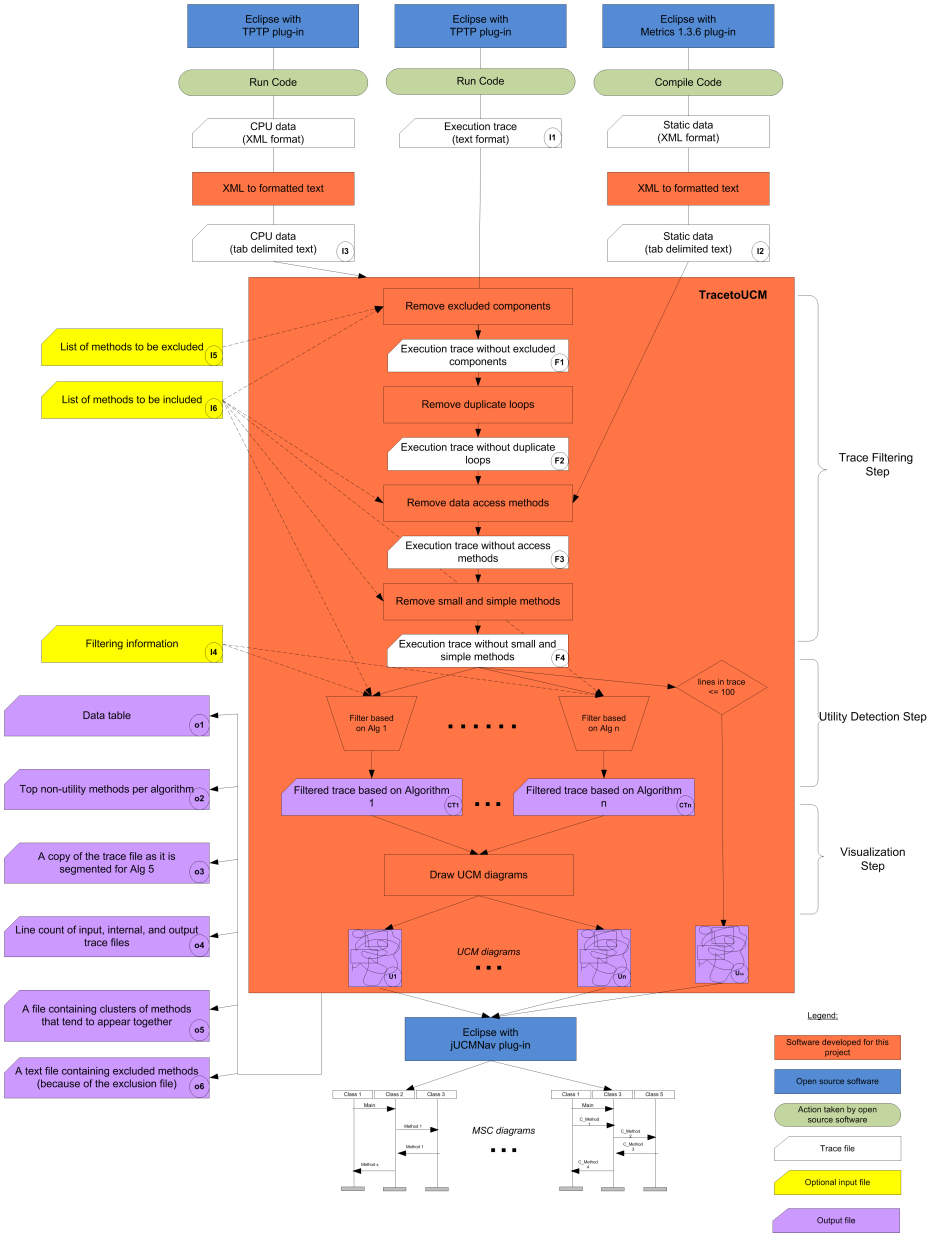


Fig. 2. Overview of the TraceToUCM tool

(independently from one another). We can see that before the algorithms are applied to the trace, just by removing repeated loops, access methods, as well

as small and simple methods, we have reduced the execution trace from 19,543 to 1987 lines, which already represents a 90% reduction in size.

Table 3. UCEd execution trace line count

Trace name	Trace size	Reduction
Uced_Original	19543	0.00%
Uced_Original_LoopsRemoved	7698	60.61%
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%

The eight UCM diagrams are then generated, one per algorithm. In the first iteration, we actually have set the filter at 40, i.e., the 40 methods ranked as non-utility would be included in the trace and the UCM diagram. Fig. 3 shows the generated UCM diagram based on the top 40 methods ranked according to Algorithm 4. The diagram is cluttered and is hard to follow, but it does show many UCM nested components capturing classes within their packages.

We then raised the level of abstraction a bit higher 1) by using the top 20 methods instead of 40 and 2) by using UCM stubs to hide some of the details in sub-diagrams. The result is shown in Fig. 4. This new UCM diagram is much easier to follow than the UCM diagram in Fig. 3, but may miss important information. One can play easily with such levels of abstraction in order to at least generate a diagram that stands a chance of being understandable.

In order to assess which of the algorithms lead to the most understandable and intuitive results for this trace, we showed the corresponding UCM diagrams to two UCEd designers and asked them whether the diagrams contained the main methods used for understanding the "Open a project" feature. Both designers chose the UCM diagrams based on Algorithms 1 and 4, for 20 methods (e.g., the one in Fig. 4).

The UCM models produced by the TraceToUCM tool also contain scenario definitions, which allow one to simulate the UCM model and to transform each execution sequence into a sequence diagram. This hence provides a different visualization of the same information. Each UCM component becomes a lifeline in the sequence diagram, but containment relationships are more difficult to understand. Sequence diagrams are also general less compact than UCM diagrams

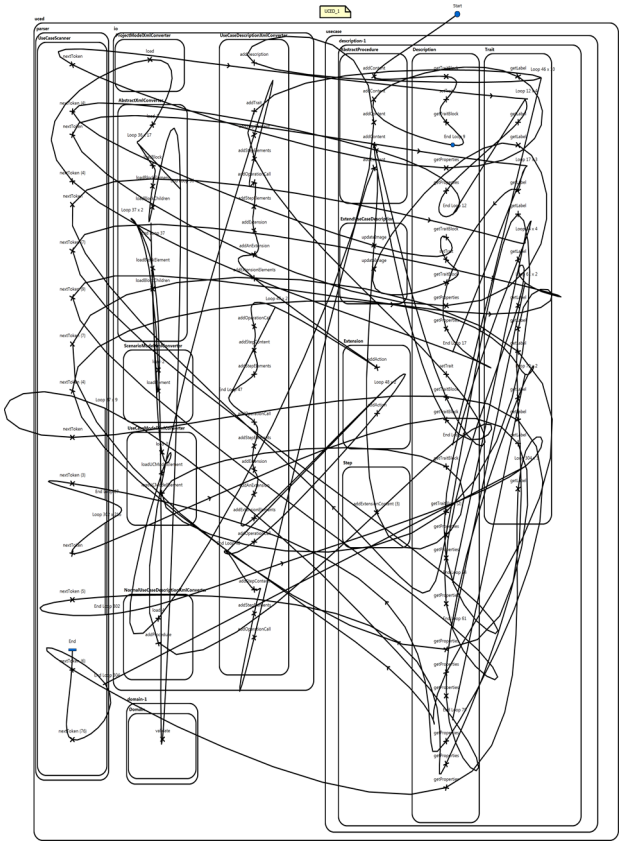


Fig. 3. Use Case Map model derived for the UCEd trace using the top 40 methods (not meant to be read)

(especially with synthetic messages), but on the other hand UCM diagrams may have more complicated layout issues, possibly with paths crossing each other.

6 Discussion and Evaluation

6.1 Additional Experiments

In [4], we also report on the additional evaluation of traces obtained from two other Java projects: jUCMNav (with 94 KLOC, and 3 developers for validation) and Umple (with 79 KLOC, and 4 developers). Again, the developers were asked to determine which UCM diagrams were suitable for understanding specific features. Looking at the scores from the three groups of designers (including UCEd's), Algorithms 2 and 4 generally come ahead of the other algorithms. The algorithms that did better than the others used method size, method complexity and nested block depth metrics. More work needs to be done to fine-tune the

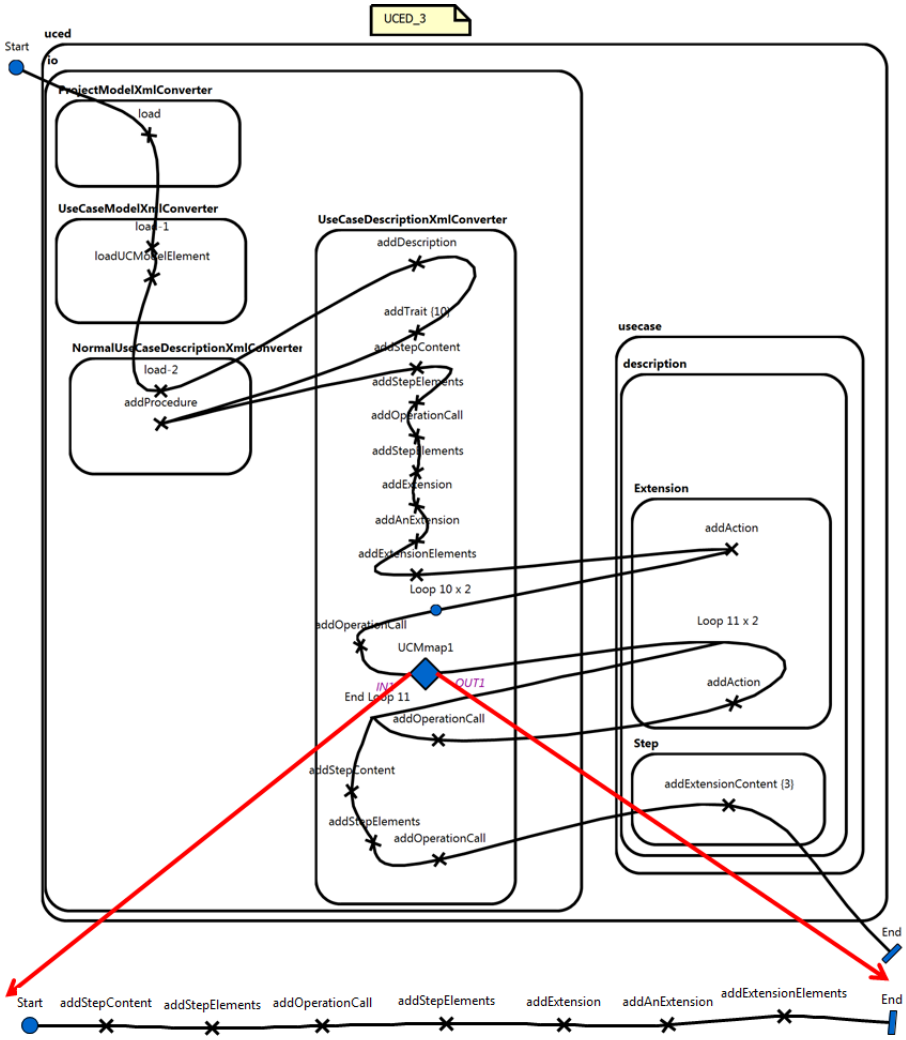


Fig. 4. Use Case Map model derived for the UCEd trace using the top 20 methods, with stubs

combinations of those parameters in the algorithms. Based on answers to a questionnaire, there was agreement among the designers that UCM diagrams were suitable to represent execution traces. 7 out of 8 designers agreed (including one who strongly agreed) and one was neutral.

We have also reverse-engineered our own TraceToUCM tool (3.4 KLOC) by producing UCM diagrams for its main feature. These diagrams were presented to 16 students and software engineers who were asked to pick the top three diagrams/algorithms that they felt gave enough information to describe what the program did. The results showed that reduced traces, visualized by using

the UCM notation, can be helpful in documenting software. We also learned that it may not be desirable to pick one "best" algorithm since a few of the participants expressed they had multiple favorites.

While designing the TraceToUCM method and tool, we considered the following criteria, which were used to assess 11 other existing program comprehension approaches explored in [4]. We look at each criterion and discuss how TraceToUCM measures up.

- *Scalability*: TraceToUCM is scalable because regardless of the target software, the user can focus the behavioral model extraction on one feature at a time.
- *Scalability for trace size*: TraceToUCM can be used regardless of the input file size. UCM supports recursive levels of nested stubs, and by using stubs we can visualize large traces.
- *Visualization*: The auto-layout program that is used by jUCMNav was not always able to provide readable layouts of the UCM diagrams. Manual intervention was necessary to move overlapping component names and intertwined paths.
- *Level of prior knowledge required*: No prior knowledge of the code is required to use this technique. The user only needs to know how to run the selected feature.
- *Validation*: Validation of the UCM diagrams was done with many participants already familiar with the UCM notation, which adds some bias to our results.
- *Usability*: Even though all that the user needs to do is exercise the feature of interest, collect data, and run TraceToUCM, a process that is hence mostly automated, manual intervention is required to untangle larger UCM diagrams. More work needs to be done to make the approach more user-friendly.
- *Intrusive data collection*: TraceToUCM, which builds on TPTP, has an observer effect from the collection of CPU data and is therefore not suitable for time-critical systems.
- *Target programming language*: The technique of comparing execution traces is likely generalizable to all systems written with an object-oriented programming language. However, the tool support itself is currently limited to Java programs.

In general, TraceToUCM does better on average than other program comprehension techniques, with some weaknesses in terms of visualization, validation and utility, where other approaches have at times scored better.

6.2 Threats to Validity

There are some threats to the validity of our approach. We look at the threats using a general framework proposed by Wohlin et al. [29], 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.

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.

Internal validity refers to the ability to correctly infer connections between dependent and independent variables. 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 first 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), 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 TraceToUCM 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 are. In addition, the inclusion input file (labeled I6 in Fig. 2) can be used to ensure important methods are not filtered out even if they score low by the filters and algorithms.

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 behavioral 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 [14]), 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 [11,15,27]. In that 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 jUCMNav's auto-layout feature 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. Therefore, our dependent variable, understandability of the UCM diagram, may have been hampered by the layout tool.

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 (as designers with a general and deep understanding of their systems are difficult to access), which prevents statistical significance from being reached. Work needs to be done using larger collections of software systems, where dozens of developers are available.

7 Conclusion and Future Work

This paper contributes TraceToUCM, an environment where different algorithms targeting the filtering of utilities (exploiting dynamic, static, and CPU information) can be implemented for comparison and evaluation. TraceToUCM also supports the visualization of reduced traces with Use Case Maps and sequence diagrams for software feature documentation and comprehension. A comparison of eight filtering algorithms was performed with three real software projects and their designers (who know the systems), and with one additional project where people who did not know the software had to explain and rank generated UCM diagrams. A comparison with other related approaches along eight criteria highlighted the benefits, limitations, and overall good potential of TraceToUCM.

Note that this work focused on programs written in Java. Yet, 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.

Further characterization of utilities could be done as another future work item. One could investigate how context-dependent utilities are used in different contexts. For example, 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)?

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 parallel forks) 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. [5] on the reverse engineering of behavior from distributed Java applications.

Acknowledgements. This work was sponsored in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Amyot, D., Mansurov, N., Mussbacher, G.: Understanding existing software with use case map scenarios. In: Sherratt, E. (ed.) Telecommunications and beyond: The Broader Applicability of SDL and MSC, Lecture Notes in Computer Science, vol. 2599, pp. 124–140. Springer Berlin Heidelberg (2003)
2. Amyot, D., Mussbacher, G.: User Requirements Notation: The first ten years, the next ten years. JSW 6(5), 747–768 (2011)
3. Ball, T.: The concept of dynamic analysis. In: Nierstrasz, O., Lemoine, M. (eds.) Software Engineering – ESEC/FSE'99, Lecture Notes in Computer Science, vol. 1687, pp. 216–234. Springer Berlin Heidelberg (1999)
4. Braun, E.: Reverse Engineering Behavioural Models by Filtering out Utilities from Execution Traces. Ph.D. thesis, University of Ottawa, Canada (2013), `http://hdl.handle.net/10393/26093`
5. Briand, L., Labiche, Y., Leduc, J.: Toward the reverse engineering of UML sequence diagrams for distributed Java software. Software Engineering, IEEE Transactions on 32(9), 642–663 (Sept 2006)
6. Buhr, R.: Use case maps as architectural entities for complex systems. Software Engineering, IEEE Transactions on 24(12), 1131–1155 (Dec 1998)
7. Burtscher, M., Ganusov, I., Jackson, S., Ke, J., Ratanaworabhan, P., Sam, N.: The VPC trace-compression algorithms. Computers, IEEE Transactions on 54(11), 1329–1344 (Nov 2005)
8. Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R.: A systematic survey of program comprehension through dynamic analysis. Software Engineering, IEEE Transactions on 35(5), 684–702 (Sept 2009)
9. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process 25(1), 53–95 (2013)
10. Dugerdil, P., Jossi, S.: Empirical assessment of execution trace segmentation in reverse-engineering. In: ICSOFT 2008 - Proceedings of the Third International Conference on Software and Data Technologies, Volume SE/MUSE/GSDCA. pp. 20–27. INSTICC Press (2008)
11. Dugerdil, P., Repond, J.: Automatic generation of abstract views for legacy software comprehension. In: Proceedings of the 3rd India Software Engineering Conference. pp. 23–32. ISEC '10, ACM, New York, NY, USA (2010)
12. Eclipse Foundation: Eclipse test and performance tools platform project (2011), `http://www.eclipse.org/tptp/`

13. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. Software Engineering, IEEE Transactions on 29(3), 210–224 (March 2003)
14. Hamou-Lhadj, A., Braun, E., Amyot, D., Lethbridge, T.: Recovering behavioral design models from execution traces. In: 9th Software Maintenance and Reengineering (CSMR). pp. 112–121 (March 2005)
15. Hamou-Lhadj, A.: Techniques to Simplify the Analysis of Execution Traces for Program Comprehension. Ph.D. thesis, University of Ottawa, Canada (2006), http://hdl.handle.net/10393/29296
16. ITU-T: Recommendation Z.151 (10/12), User Requirements Notation (URN) language definition, Geneva, Switzerland. http://www.itu.int/rec/T-REC-Z.151/en (2012), http://www.itu.int/rec/T-REC-Z.151/en
17. Lee, H.B., Zorn, B.G.: BIT: A tool for instrumenting Java bytecodes. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems. pp. 7–7. USITS'97, USENIX Association, Berkeley, CA, USA (1997)
18. von Mayrhauser, A., Vans, A.: Program understanding behavior during adaptation of large scale software. In: Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on. pp. 164–172 (Jun 1998)
19. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. 2(4), 308–320 (Jul 1976)
20. Mussbacher, G., Amyot, D.: Goal and scenario modeling, analysis, and transformation with jUCMNav. In: Software Engineering - Companion Volume. pp. 431–432 (May 2009)
21. Richner, T., Ducasse, S.: Recovering high-level views of object-oriented applications from static and dynamic information. In: IEEE International Conference on Software Maintenance. pp. 13–22 (1999)
22. Rilling, J., Seffah, A., Bouthlier, C.: The CONCEPT project - applying source code analysis to reduce information complexity of static and dynamic visualization techniques. In: Visualizing Software for Understanding and Analysis. pp. 90–99 (2002)
23. Sauer, F.: Metrics 1.3.6 (2013), http://metrics.sourceforge.net/
24. Somé, S.S.: Use case editor (UCEd) (2007), http://www.site.uottawa.ca/~ssome/Use_Case_Editor_UCEd.html
25. Systä, T.: Understanding the behaviour of Java programs. In: Seventh Working Conference on Reverse Engineering (WCRE'00). pp. 214–223. IEEE CS (2000)
26. Systä, T., Yu, P., Muller, H.: Analyzing Java software by combining metrics and program visualization. In: 4th Software Maintenance and Reengineering Conference. pp. 199–208 (Feb 2000)
27. Wang, Y.y., Li, Q.s., Chen, P., Ren, C.d.: Dynamic fan-in and fan-out metrics for program comprehension. Journal of Shanghai University (English Edition) 11(5), 474–479 (2007)
28. Wilde, N., Scully, M.C.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice 7(1), 49–62 (1995)
29. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA (2000)