Delft University of Technology
Software Engineering Research Group
Technical Report Series

On large execution traces and trace abstraction techniques

Bas Cornelissen and Leon Moonen

Report TUD-SERG-2008-005

TUD-SERG-2008-005

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Submitted to the Journal on Software Maintenance and Evolution (JSME), 2008, Wiley & Sons.

JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **00**:1–7 *Prepared using* smrauth.cls *[Version: 2003/05/07 v1.1]*

Research

On Large Execution Traces and Trace Abstraction Techniques

Bas Cornelissen[1,*], Leon Moonen[1]

[1] *Delft University of Technology, The Netherlands*

SUMMARY

Program comprehension is an important concern in the context of software maintenance tasks because these activities generally require a certain degree of knowledge of the system at hand. Although the use of dynamic analysis for information gathering has become increasingly popular, the literature indicates that dealing with the excessive amounts of data resulting from dynamic analysis remains a formidable challenge.

Although various trace abstraction techniques have been proposed to address these scalability concerns, such techniques are typically not discussed in terms of properties such as complexity and information preservation, and lack thorough evaluation of technique-specific parameters. Moreover, the absence of a common execution trace repository makes matters even worse, as most researchers test their techniques only on their own particular traces. Consequently, it is very difficult to make a fair comparison between the abstraction techniques known from literature.

In this paper, we present a characterization of large execution traces in which a set of key properties is extracted from a series of seven representative traces from six different object-oriented systems. Having highlighted the key issues in this context, we propose an assessment methodology for the quantitative evaluation and comparison of trace abstraction techniques. We apply this methodology on a selection of three light-weight abstraction methods, assessing them on the basis of metrics that are relevant in their evaluation. We discuss the results, compare the techniques, and relate the measurements to the trace characteristics found earlier.

Our findings provide a valuable insight into factors that play an important role in coping with large execution traces, and we consider our work a significant step towards a systematic assessment of (new) trace abstraction techniques.

KEY WORDS: program comprehension, dynamic analysis, execution traces

*Correspondence to: s.g.m.cornelissen@tudelft.nl

1. Introduction

Program comprehension is an important aspect of the software development process [1]. For example, when performing software maintenance tasks, the engineer must first familiarize himself with the program at hand before any action is to be undertaken. In doing so, a mental map is built that bridges the gap between the system's high-level concepts and its source code (e.g., [2, 3]).

There exist various types of methods to gain knowledge of a software system. Static analysis focuses on such artifacts as source code and documentation, and potentially covers all of the program's execution paths. Dynamic analysis, on the other hand, concerns the examination of the program's behavior at runtime, which offers the ability to reveal object identities and occurrences of late binding [4]. One of the main issues with dynamic techniques, especially if performed in a post mortem fashion, is the amount of data that needs to be analyzed: the execution traces that are produced by instrumented software are generally too large to comprehend [5].

The majority of events in execution traces stem from repetitions during the program's execution behavior [6]. While such constructs as loops offer relatively little information, they result in large traces for even the simplest scenarios, thus rendering common visualizations useless without the necessary abstraction steps [5]. This poses a serious problem not only for straightforward visualizations such as UML sequence diagrams [7, 8], but also for more elaborate techniques (e.g., [9]) as they typically do not scale up to traces of millions of events.

In recent years, many potential solutions have been proposed to tackle the scalability issues that are associated with large execution traces. Unfortunately, a thorough comparison of such techniques is hampered by three factors. First, trace abstraction techniques have certain characteristics such as computational complexity and information loss, which are seldomly explicitly quantified by the authors. Second, the results of such evaluations depend largely on certain technique-specific parameters of which the significance and effects are often non-trivial. Third, we have observed that different researchers generally use distinct execution traces to evaluate their techniques on. A major consequence of these three factors is that abstraction techniques are rarely subjected to extensive assessments in which the key aspects are thoroughly discussed and, by extension, fairly compared to existing solutions side-by-side.

Goal of the paper. The goals of this paper are to characterize key properties of large execution traces, and to use this knowledge in defining a systematic assessment methodology for trace abstraction techniques, focusing on the techniques' quantitative aspects.

We achieve these goals through the characterization of seven large execution traces that we consider to be representative in terms of both size and application type. Next, we motivate and discuss a selection of three trace abstraction techniques encountered in literature. We then describe an assessment methodology that enables the evaluation and comparison of such techniques, and apply our methodology on example techniques. In this assessment, they are compared using a set of evaluation criteria that is based on the trace properties that were found earlier. The results enable us to reason about the benefits and drawbacks of each technique and allow for a fair comparison of the approaches, which in turn enables developers to make a balanced choice between the techniques being offered in literature.

The main contributions of this paper are the following:

- The characterization of key properties of a series of large execution traces.
- The definition of an assessment methodology for trace abstraction techniques, focusing on the quantitative aspects.
- The application of this methodology through the assessment of three trace abstraction techniques found in literature.

Structure of the paper. This paper is organized as follows. In Section 2 we perform an analysis of seven large execution traces. We then describe three trace abstraction techniques in Section 3. In Section 4 we elaborate on our assessment methodology. An example assessment is described in Section 5, after which we discuss the results in Section 6. Related work is outlined in Section 7, after which we present conclusions and future directions in Section 8.

2. A Characterization of Large Execution Traces

Dynamic analysis is often praised as a useful means to extract precise information from software. Indeed, particularly in the context of object-oriented systems, dynamic techniques are powerful as they are capable of revealing occurrences of polymorphism and late binding. On the other hand, the major challenge of dynamic analysis is equally well-known: the massive amounts of data that are collected at runtime make it difficult for the user to extract the required information [5].

While there is a great deal of ongoing research concerning this challenge, reports on extensive traces analyses that involve reasonably large sets of sizable traces are rare (e.g., [6]). For this reason, it is our opinion that the *key properties* of large execution traces have not been convincingly identified to the present day.

In this section, we present a characterization of a seven large execution traces from six object-oriented systems. Such a characterization serves two purposes:

- The establishment of a context, i.e., an outline of the problem area of large execution traces and the metrics involved therein.
- The identification of trace characteristics that can have a significant impact on the results of certain abstraction techniques.

We first describe the systems under study in Section 2.1 and the execution scenarios in Section 2.2. We then propose a set of trace metrics in Section 2.3 and discuss the resulting traces in Section 2.4.

2.1. Systems under study

The focus in this paper is on six different Java systems. One of these systems concerns an industrial case; four are open source systems. The next section provides brief descriptions of the cases.

Table I. Properties of the systems under study.

Case	Type	LOC	# classes	# inst.classes	# inst.sigs
JPacman 1.8.1	gui	3 K	22	19	318
Cromod 1.0	batch	51 K	145	109	5,721
Checkstyle 4.3	batch	57 K	275	241	6,515
JHotDraw 6.0b	gui	73 K	398	330	6,797
Apache Ant 1.6.1	batch	99 K	520	-	-
Azureus 2.5.0.0	gui	436 K	4,864	4,122	54,801

2.1.1. System descriptions

JPacman is a small application that is used for educational purposes at Delft University of Technology. The program is an implementation of the well-known Pacman game in which the player can move around on a map while eating food and evading monsters. While the application is not large in terms of source code, previous experiences showed the amount of interactions during play to be significant [8, 9].

Cromod is an industrial system that regulates the environmental conditions in greenhouses. Given a set of sensor inputs, it calculates for a series of discrete points in time the optimal values for such parameters as heating, windows, and shutters. Since these calculations are performed for a great number of points in time, a typical scenario involves massive amounts of interactions.

Checkstyle[†] is a source code validation tool for Java. It utilizes a set of coding standards to process one or more input files, while systematically looking for irregularities and reporting these to the user. The number of events induced in an execution scenario depends on the size of the input file(s).

JHotDraw[‡] is a well-known tool for graphics editing. It was developed as a showcase for design pattern usage and is generally considered to be well-designed. It presents the user with a GUI that offers various graphical features such as the insertion of figures.

Apache Ant[§] is a Java-based build tool. It owes much of its popularity to its ability to work cross-platform. The execution trace for this system was obtained through fellow researchers [10].

Azureus[¶] is an open source P2P client that implements the BitTorrent protocol. Users can use the GUI to exchange files by use of so-called torrents, which are files containing metadata on the files being exchanged. With its nearly 5,000 classes totaling over 400,000 lines of code, this is the largest system in our assessment.

[†]Checkstyle 4.3, `http://checkstyle.sourceforge.net/`
[‡]JHotDraw 6.0b, `http://www.jhotdraw.org/`
[§]Apache Ant 1.6.1, `http://ant.apache.org/`
[¶]Azureus 2.5.0.0, `http://azureus.sourceforge.net/`

2.1.2. System properties

For each of the systems under study, we have identified five properties that we feel may influence the characteristics of their execution traces. These properties are described below.

Type. In terms of usability, we roughly distinguish between two execution types: *batch* and *GUI*. A program of the former type takes a set of command line parameters, fulfills its purpose, and terminates. The latter category concerns systems that involve graphical user interaction during their execution.

Lines of code. The total amount of lines of code (LOC) provides a rough indication of a system's size and complexity. Since we do not consider the exact number to be relevant in a trace analysis context, we choose to simply count all lines.

No. of classes. The total number of classes in the system.

No. of instrumented classes. For each system we specify the number of instrumented classes. Only the *core* classes are subject to instrumentation; this excludes such functionalities as XML parsing, mouse movements, and extensive logging [10, 9].

No. of instrumented signatures. We specify each system's number of instrumented method and constructor signatures since this figure is potentially useful in determining the degree of repetition in a trace.

Table I summarizes these properties for each of the six systems[‖]. The discrepancies between the total numbers of classes and the instrumented portions are explained by the omission of classes pertaining to non-core functionalities, and the fact that interfaces are not instrumented.

2.2. Scenario descriptions

With the exception of APACHE ANT which was instrumented using the Java Virtual Machine Profiler Interface (JVMPI), the systems described in the previous section have been instrumented with AspectJ [11]. This was done using an aspect that captures all constructor and (static) method calls between the systems' classes [7, 8]. Next, for each system we have defined and executed a typical scenario. Exceptions are APACHE ANT, as its scenario and execution trace were created by a third party, and CHECKSTYLE, for which we have defined *two* distinct scenarios as this allows for the comparison of the two resulting traces. We provide brief descriptions of the execution scenarios in Table II.

2.3. Trace metrics

In our characterization we distinguish seven different metrics that we consider to be important in a trace analysis context. While some can be considered as an addition to existing work (e.g.,

[‖]Note that since APACHE ANT was not instrumented by ourselves, unfortunately no accurate information on its instrumentation phase is available.

Table II. Description of the execution scenarios.

Scenario	Description
`checkstyle-simple`	The processing of a small input file containing 50 lines of Java code.
`pacman-death`	The start of a game, several movements, player death, start of a new game, and quit [9].
`jhotdraw-3draw5fig`	The creation of a new drawing in which five different figures are inserted, after which the drawing is closed. This process is repeated two times [9].
`cromod-assignment`	The execution of a typical assignment that involves the calculation of greenhouse parameters for two days for one greenhouse section.
`checkstyle-3checks`	The processing of three Java source files that are between 500 and 1000 lines in size each.
`azureus-newtorrent`	The initialization and invocation of the program's "new torrent" functionality on a small file before quitting.
`ant-selfbuild`	The building of Apache Ant itself [10].

Hamou-Lhadj and Lethbridge [6]), we do not take *recurrent patterns* into account. One reason behind this choice is that we have experienced that pattern recognition is too expensive when dealing with traces that contain millions of events. Another reason is the involvement of many different matching criteria in pattern recognition algorithms, which warrant for an extensive assessment and comparison in themselves.

The metrics used in our experiment include:

No. of calls. This metric describes the total number of constructor and method calls in the trace. A call in our definition consists of a calling class, a callee class, and a signature.

No. of unique calls. The number of unique calls in the trace.

Percentage of constructors. A measure for the percentage of calls that concerns constructors.

Repetitiveness. The repetitiveness is a simple measure for the degree of repetition in a trace, with a higher value corresponding to more recurrences. We define this metric as follows:

$$Repetitiveness \;=\; \left(\; 1 \;-\; \frac{\#\ calls - \#\ unique\ calls}{\#\ calls} \;\right) \;*\; 100$$

While this definition by no means covers all aspects of repetitive behavior in traces (e.g., recurring patterns [12, 13]), it does provide an indication as to the ratio between the total number of calls and the number of unique calls.

Average stack depth. The average stack depth during execution.

Maximum stack depth. The maximum stack depth during execution.

No. of threads. The total number of concurrent threads.

Table III. Trace measurement results.

Trace	# calls	# unique calls	% constructors	repetitiveness	average depth	maximum depth	# threads
checkstyle-simple	31,238	1,920	7.66	93.85	8.53	46	1
pacman-death	139,582	156	0.79	99.89	2.82	8	1
jhotdraw-3draw5fig	161,087	1,943	3.53	98.79	6.26	16	1
cromod-assignment	266,343	353	97.77	99.87	5.27	11	11
checkstyle-3checks	1,173,968	2,079	1.71	99.82	17.09	104	1
azureus-newtorrent	3,713,026	34,660	1.00	99.07	400.89	466	172
ant-selfbuild	12,135,031	1,404	0.17	99.99	38.00	53	1

2.4. Results

The seven scenarios were executed on the instrumented systems, after which the aforementioned metrics were extracted from each of the seven traces. Table III shows the results, with Figure 1 providing additional information on the stack depth progressions during the executions. The results are discussed in the next sections and we conclude with a summary of our findings in Section 2.4.5.

2.4.1. Trace size

`checkstyle-simple` is both the shortest *and* least repetitive trace: its 31,238 calls yield a repetitiveness of 93.85. This suggests that in comparison to the other six traces, a relatively large amount of the program's functionality is covered by this scenario. On an interesting note, the `ant-selfbuild` trace is both the longest *and* most repetitive trace.

Furthermore, the `checkstyle-3checks` trace is almost 38 times as long as `checkstyle-simple`, yet offers a mere 23 additional (unique) calls.

2.4.2. Constructors and unique calls

`pacman-death` has the smallest number of *unique calls*. This corresponds to its limited size in terms of lines of code, and results in a degree of repetitiveness that is surpassed only by `ant-selfbuild`, which features 87 times as many calls.

In comparison to `checkstyle-3checks`, a relatively large share of the calls in `checkstyle-simple` is made up from *constructors* (7.66% versus 1.71%). This suggests that CHECKSTYLE's initialization phase is responsible for a significant portion of this trace. This

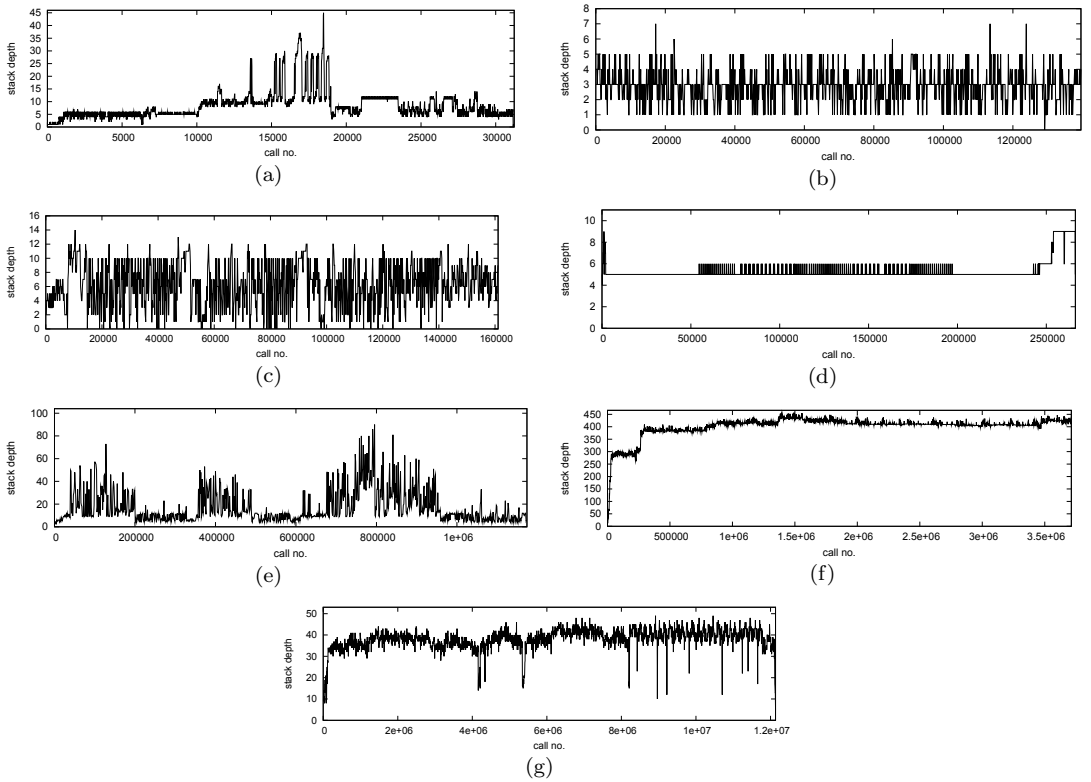

Figure 1. Progression of the stack depth throughout each of the seven traces.
(a) checkstyle-simple; (b) pacman-death; (c) jhotdraw-3draw5fig; (d) cromod-assignment; (e) checkstyle-3checks; (f) azureus-newtorrent; (g) ant-selfbuild.

also means that abstractions such as constructor hiding [8] are potentially suitable in short scenarios.

`cromod-assignment` has an extraordinarily high share of constructors; as it turns out, this is due to the creation of numerous discrete time objects during the model calculation.

2.4.3. Repetitiveness and stack depth metrics

The repetitiveness and average stack depth of the `jhotdraw-3draw5fig` trace are in par with those of the CHECKSTYLE traces, yet its *maximum* depth (16) is much lower than the values of 46 found in `checkstyle-simple` and 104 in `checkstyle-3checks`.

`cromod-assignment` and `pacman-death` have remarkably low stack depths, rendering these traces sensitive to depth-based abstractions.

The three largest traces feature the highest average stack depths. The `azureus-newtorrent` trace is of particular interest: as shown in Figure 1(f), the stack depth rapidly increases during

the first 250,000 calls and never drops significantly during the remainder. This can be attributed to the fact that upon termination of the AZUREUS execution, most of the 172 "open" threads (accounting for a high stack depth) are not cleanly shut down. The same issue exists with 3 out of CROMOD's 11 threads. The consequence is that stack depth-related abstractions must be cautiously applied when dealing with multithreaded software.

2.4.4. Stack depth progressions

Figure 1(d) indicates that the progression of the stack depth in `cromod-assignment` is exceptionally stable, with the majority of calls occurring at depths 5 and 6.

Another important observation is the fact that `pacman-death` and `jhotdraw-3draw5fig` are both traces from GUI-controlled programs, and feature stack depth progressions (Figure 1(b) and (c)) that have common "baselines", i.e., the stack depths regularly return to low values. In the context of such baselines, depth-limiting abstraction techniques might prove useful, as they enable a distinction between high-level control events and low-level details. While there is no such baseline in the GUI-based AZUREUS, it must be noted that the stack depths in this trace are difficult to interpret due to the aforementioned thread issue.

2.4.5. Summary

The table below summarizes the results of our trace characterization.

Characteristic	**Implication**
High degree of repetitiveness	Opportunities for abstractions related to pattern detection
Low degree of repetitiveness	Opportunities for abstractions related to constructor hiding
Common stack depth baseline	Stack depth-related abstractions can separate high and low level events
Substantial amount of constructors	Opportunities for abstractions based on constructor hiding
Multiple threads	Limited applicability of stack depth-related abstractions

3. Light-weight Trace Abstraction Techniques

The example assessment in this paper concerns a selection of three light-weight abstraction techniques. Our choice for these particular techniques is motivated by three criteria that we feel are key properties in covering as many real-life use cases as possible, i.e., that render the techniques useful in a broad range of program comprehension contexts.

- **High degree of scalability.** For a scalability technique to be applicable in practice, it must be powerful enough to cope with traces containing millions of events.
- **Preservation of temporal information.** We consider the chronological order of the events to be a key feature in many dynamic analysis contexts, and therefore advocate the preservation of this information.

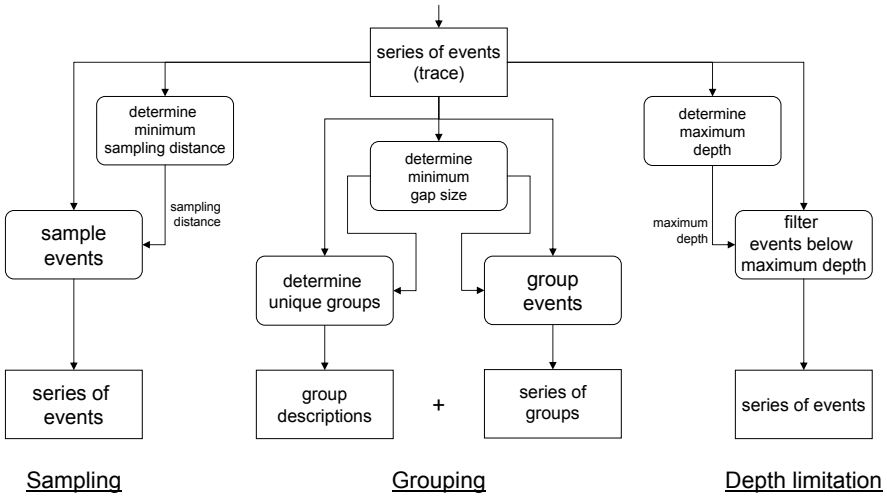

Figure 2. Overview of the three abstraction techniques in the assessment.

- **Independence of domain knowledge.** In order for a scalability technique to be quickly and broadly applicable in practice, no expert knowledge regarding the technique or the system under study should be required.

Based on these criteria, we have selected three abstraction techniques used in literature: *sampling*, *stack depth limitation*, and *grouping*. These methods are outlined in Figure 2 and are described in the following sections.

3.1. First technique: Sampling

The first technique that we investigate is *sampling*, an effective abstraction method that is broadly used in software research and by Chan et al. [14] in the context of dynamic analysis. The variant that we use in our experiment is simple: rather than processing all events in an execution trace, we consider every n-th event. We call n the sampling distance: its value is dependent on the trace size N on the one hand, and the maximum size of the output data M on the other. The latter parameter must be specified by the user whereas the sampling distance can be automatically determined in advance.

As an example, let us consider a trace that contains 100,000 events, i.e., $N = 100{,}000$, and some visualization tool that requires its input data to contain at most 40,000 events, i.e., $M = 40{,}000$. In order for the resulting dataset's size to respect this limit, the sampling technique must process every third event, i.e., $n = 3$. This results in the processing of 33,333 points out of the original 100,000.

Table IV. Stack depths during an example scenario, with the number of events at each depth.

Stack depth	0	1	2	3	4	5	6	7	8	9	10	11
Frequency	607	674	890	1,584	2,115	2,142	2,930	1,052	471	74	16	16

3.2. Second technique: Stack depth limitation

The second technique that is incorporated in this experiment is based on *stack depth limitation*. This form of abstraction has been used both in static contexts [15] and in our earlier work [8], where it proved to be quite effective in filtering implementation details from test case executions. The variant discussed here revolves around the definition of a *maximum* depth d; events that take place at depths higher than this threshold are filtered from the original trace. The value of this threshold depends on two factors: the stack depth "distribution" in the trace, and the maximum size of the resulting dataset.

For this technique to obtain the necessary stack depth information, the algorithm first collects the amount of events at each depth. Next, given the maximum output size, the value of the maximum depth can be automatically determined. The example stack depth distribution in Table IV, for instance, implies that for $M = 5{,}000$, the maximum depth must be 4. The user-defined value of M is this technique's only parameter.

The use of stack depth limitation effectively filters (deeply) nested events, resulting in a dataset that mostly contains high-level events. The number of events that are subject to filtering is largely dependent on the stack depth progression throughout a trace.

3.3. Third technique: Grouping

The final abstraction technique that we put to the test is concerned with the *grouping* of events. Rather than processing an execution trace on an event-by-event basis, this method groups the events according to some criterion. The result is an abstracted dataset that is comprised of a significantly smaller number of elements that can be processed much faster. The "contents" of each group are stored separately and accessed when needed.

The grouping criterion that we utilize is based on stack depth differences between the events, and was proposed by Kuhn and Greevy [16]. Using this method, which they call *monotone subsequence summarization*, the algorithm assigns consecutive events that have equal or increasing stack depths to the same group. As soon as a stack depth decrease is encountered and the difference is larger than a certain user-defined threshold, called the *gap size*, a new group is initiated. Considering the fact that the stack depth constantly fluctuates during an execution scenario, the number of resulting groups is typically much smaller than the number of original events.

While the aforementioned paper does not report on an elaborate assessment of the effect of the gap size, it is stated that a gap size of 3 allows for the reduction of a typical trace to 10% of its original size [16]. This motivates the technique's inclusion in our experiment: a thorough evaluation using different gap sizes provides a deeper insight into its effectiveness.

4. Assessment Methodology

To address the aforementioned issues concerning the comparison of abstraction techniques, we propose an assessment methodology that is aimed at the thorough evaluation of trace abstraction mechanisms. Such assessments are important because they enable a side-by-side comparison of both existing and future approaches, e.g., in terms of computational complexity.

The methodology that we propose focuses specifically on a technique's quantitative aspects. While we acknowledge the importance of a qualitative component during any assessment, e.g., a technique's effect in terms of semantics, this is left to future work for reasons of space.

Our methodology distinguishes the following steps:

1. The definition of a set of evaluation criteria by which to compare the abstraction techniques.

2. The definition of set of metrics that enables the reasoning about the techniques in terms of the aforementioned criteria.

3. The application of the techniques on a representative test set while extracting the previously defined metrics, given a series of thresholds.

4. The interpretation of the results of each particular technique.

5. The comparison of the techniques, based on the evaluation criteria defined earlier.

The application of this methodology in the assessing a set of abstraction techniques provides valuable information that can help software developers to understand the applicability of those techniques in specific contexts, thus enabling them to make a balanced choice between the abstraction techniques being offered.

5. Experimental Setup

We illustrate our assessment methodology by performing an experiment in which the three trace abstraction techniques described in Section 3 are quantitatively evaluated. As a test set we use the seven execution traces presented in Section 2.

The following sections describe the five steps that are to be undertaken in this experiment.

5.1. Evaluation criteria

The comparison between the techniques is to be performed on the basis of three criteria: computational complexity, information preservation, and reliability.

Computational complexity is a criterion for the computational effort that is involved in the application of a technique. In order to enable a fair comparison, this criterion takes into account the results for all traces and thresholds involved in the experiment. As such, this criterion determines the technique's scalability potential.

Information preservation concerns the degree to which the information from the original trace is conveyed. Clearly, the application of an abstraction mechanism always involves the filtering of information, but what is important is the amount and the nature of the data being lost. This observation is vital in determining the suitability of the technique in particular contexts.

Reliability is a measure for the degree to which the results are predictable. The significance of this criterion is the fact that for an abstraction technique to be useful in practice, the user must know what to expect from the technique, i.e., to determine whether it can prove useful for certain traces or contexts.

5.2. Metrics to be extracted

We have selected a set of metrics that we feel are the most relevant in assessing the abstraction techniques discussed in this paper. Also of importance are the parameters that are specific to certain techniques which are determined automatically during each run.

Given a certain technique and an execution trace, we distinguish two input parameters:

- **Input size.** The total number of constructor and method calls in the trace at hand, as defined earlier in Section 2.3.
- **Maximum output size.** A threshold that defines an upper bound for the size of the output dataset, in terms of calls.

Furthermore, in each test run we measure the following properties:

- **Actual output size.** The actual size of the output dataset after filtering, in calls.
- **Compression ratio.** The degree of compression achieved in this run, as defined by:

$$Compression\ ratio \quad = \quad \frac{input\ size}{output\ size}$$

  This metric enables a performance comparison in terms of compression.
- **Computation time.** The amount of time elapsed during the run, in seconds. Since the abstraction techniques represent different approaches, in each run we measure the total time spent on *all* subtasks. These include such tasks as reading the trace (multiple times if need be), determining the appropriate value for the technique's parameter, and the actual event filtering. The resulting measurements allow for a comparison of the techniques in terms of speed.

Finally, we mention the three parameters that are specific to their respective techniques:

- **Sampling distance.** The distance between every sampled call, in terms of calls. Used in the sampling technique.
- **Gap size.** The maximum stack depth difference between the groups. Used in the grouping technique.
- **Maximum depth.** The maximum stack depth of the resulting calls. Used in the stack depth limitation technique.

5.3. Application of the techniques

Each of the three techniques is applied on all seven traces. As we mentioned in our definition of computation time, the task being performed during each run is the filtering of events from the input trace while conforming to a certain threshold. We utilize such thresholds because in practice, there is typically a need for abstractions whenever a trace visualization tool has an upper bound on the input size. In these cases it is up to the abstraction technique to reduce the trace to a size that the visualization can cope with, e.g., 500,000 events in the case of our Extravis tool [9].

The runs are performed using four different values for the maximum output size: we employ thresholds of 1,000,000, 100,000, 10,000, and 1,000 calls. This choice of thresholds allows for a discussion on each technique's scalability potential.

Additionally, to illustrate how the effect of a technique's parameter(s) can be measured, we perform a separate experiment that focuses strictly on the grouping technique's main parameter, i.e., the gap size. We achieve this by using a set of different gap sizes and subjecting our traces to this technique with no constraints on the output size. Six different gap sizes are used for all seven traces and the resulting compression ratios of all 42 runs are measured.

5.4. Interpretation

The final stage of the experiment concerns the interpretation of the results. We discuss the techniques by themselves by focusing on the collected metrics in each of the 72 runs. Special attention is given to the effect of the grouping technique's parameter as this involves a separate experiment. We then conclude with a comparison of the techniques on the basis of the evaluation criteria described earlier.

6. Results & Discussion

The results for the three techniques are shown in Tables V, VI, and VII. In each Table, the fifth column yields the parameter that is specific to the sampling, depth limitation, and grouping technique, being the sampling distance, maximum depth, and gap size respectively. Figure 3 demonstrates the performance of each technique in terms of computation time, for which we have selected the three largest traces. Results of the separate experiment concerning the effect of the gap size are shown in Figure 4. Finally, Table VIII shows a schematic comparison of the three techniques using our evaluation criteria. We discuss our findings in the following sections.

6.1. Sampling results

Computational complexity. The computational complexity of the sampling technique is favorable: the sampling distance can be determined instantly, after which the trace needs to be processed only once. This is reflected by Table V, which shows the computational effort to

Table V. Sampling results.

Trace	input size	max.size	output size	samp.dist.	comp.ratio	time
checkstyle-simple	31,238	1,000,000	31,238	1	1.00	1
		100,000	31,238	1	1.00	1
		10,000	7,809	4	4.00	1
		1,000	976	32	32.00	1
pacman-death	139,582	1,000,000	139,582	1	1.00	2
		100,000	69,791	2	2.00	2
		10,000	9,970	14	14.00	2
		1,000	997	140	140.00	2
jhotdraw-3draw5fig	161,087	1,000,000	161,087	1	1.00	3
		100,000	80,543	2	2.00	3
		10,000	9,475	17	17.00	3
		1,000	1,000	161	161.00	3
cromod-assignment	266,343	1,000,000	266,343	1	1.00	5
		100,000	88,781	3	3.00	5
		10,000	9,864	27	27.00	5
		1,000	997	267	267.00	5
checkstyle-3checks	1,173,968	1,000,000	586,984	2	2.00	16
		100,000	97,830	12	12.00	16
		10,000	9,948	118	118.00	16
		1,000	1,000	1,173	1,173.00	16
azureus-torrent	3,713,026	1,000,000	928,256	4	4.00	48
		100,000	97,711	38	38.00	47
		10,000	9,981	372	372.00	47
		1,000	1,000	3,710	3,713.00	48
ant-selfbuild	12,135,031	1,000,000	933,463	13	13.00	118
		100,000	99,467	122	122.00	121
		10,000	9,995	1,214	1,214.00	118
		1,000	1,000	12,123	12,135.00	118

be (1) rather constant within all trace contexts, and (2) independent of the maximum output size.

Information preservation. This aspect is the sampling technique's main weak spot. Since from a semantics points of view the events are filtered completely arbitrarily, a great deal of the original trace's information value is potentially lost. Whether or not this poses a problem is dependent on the application area, i.e., the program comprehension task being supported by this abstraction.

Reliability. Using the sampling technique, the size of the output data is predictable by definition: given a maximum output size, the smallest possible sampling distance is determined, which by definition results in an output dataset of which the size is close to the threshold.

6.2. Stack depth limitation results

Computational complexity. Using the depth limitation technique, the trace at hand needs to be processed twice: the first run determines the stack depth distribution and the maximum depth, whereupon in the second run the events above the threshold are filtered. While Table VI

Table VI. Stack depth limitation results.

Trace	input size	max.size	output size	max.depth	comp.ratio	time
checkstyle-simple	31,238	1,000,000	31,238	46	1.00	1
		100,000	31,238	46	1.00	1
		10,000	4,251	4	7.35	1
		1,000	730	1	42.79	1
pacman-death	139,582	1,000,000	139,582	8	1.00	2
		100,000	44,743	2	3.12	2
		10,000	172	0	811.52	1
		1,000	172	0	811.52	1
jhotdraw-3draw5fig	161,087	1,000,000	161,087	16	1.00	4
		100,000	83,584	6	1.93	3
		10,000	5,491	0	29.34	2
		1,000	0	< 0	-	-
cromod-assignment	266,343	1,000,000	266,343	11	1.00	6
		100,000	173	4	1,539.55	4
		10,000	173	4	1,539.55	4
		1,000	173	4	1,539.55	4
checkstyle-3checks	1,173,968	1,000,000	990,561	32	1.19	19
		100,000	49,288	4	23.82	13
		10,000	1,844	3	636.64	13
		1,000	740	1	1,586.44	13
azureus-torrent	3,713,026	1,000,000	982,124	404	3.78	40
		100,000	99,719	285	37.23	34
		10,000	9,406	64	394.75	36
		1,000	673	10	5,517.13	35
ant-selfbuild	12,135,031	1,000,000	768,246	32	15.80	85
		100,000	94,396	14	128.55	84
		10,000	4,742	7	2,559.05	85
		1,000	343	6	35,379.10	81

gives the impression that this technique is faster than sampling, it must be noted that in the latter case the output sizes are closer to the thresholds. This means that the depth limitation technique generally filters more events, which in turn requires less I/O.

Information preservation. The information that is lost using this technique presumably concerns the low level events in the trace**. While this is useful when a global view of the scenario is needed (e.g., in sequence diagrams [8]), tasks that concern low-level details do not benefit from filtering based on a maximum stack depth. Using a minimum stack depth could prove more useful in these cases.

Reliability. As it turns out, the results of the depth limitation technique are very unpredictable. The size of the output dataset is largely dependent on the trace's stack depth distribution, which is difficult to convey to the user. The result is that in certain runs, there exist large differences between the maximum and actual output sizes. This occurred in several

**Note that questions regarding the nature of the events filtered by this technique can only be decisively answered through a *qualitative* evaluation.

Table VII. Grouping results.

Trace	input size	max.size	output size	gap size	comp.ratio	time
checkstyle-simple	31,238	1,000,000	23,709	0	1.32	1
		100,000	23,709	0	1.32	1
		10,000	6,417	1	4.87	1
		1,000	781	2	40.00	2
pacman-death	139,582	1,000,000	73,402	0	1.90	2
		100,000	73,402	0	1.90	2
		10,000	7,080	4	19.71	10
		1,000	37	5	3,772.49	12
jhotdraw-3draw5fig	161,087	1,000,000	109,657	0	1.47	3
		100,000	25,945	1	6.21	6
		10,000	7,559	3	21.31	11
		1,000	773	6	208.39	20
cromod-assignment	266,343	1,000,000	255,271	0	1.04	6
		100,000	10,940	1	24.35	10
		10,000	115	2	2,316.03	16
		1,000	115	2	2,316.03	16
checkstyle-3checks	1,173,968	1,000,000	922,654	0	1.27	18
		100,000	31,234	2	37.59	48
		10,000	8,033	3	146.14	65
		1,000	1	6	1173,968.00	111
azureus-torrent	3,713,026	1,000,000	256,520	1	14.47	97
		100,000	89,438	2	41.52	143
		10,000	5,829	5	636.99	287
		1,000	141	18	26,333.52	898
ant-selfbuild	12,135,031	1,000,000	921,549	3	13.17	476
		100,000	34,432	5	352.43	694
		10,000	4,830	7	2,512.43	933
		1,000	534	9	22,724.78	1,154

cases: three out of four CROMOD runs resulted in outputs containing a mere 173 events, while JHOTDRAW shows the worst-case scenario, as a threshold of 1,000 events means not a single event remains after filtering.

6.3. Grouping results

Computational complexity. The use of grouping typically requires a trace to be processed multiple times, as the gap size must be repeatedly incremented until a suitable projected output size has been found. As shown in Table VII and Figure 3, this iterative process yields significant overheads if (1) the trace at hand is very large, or (2) the maximum output size is relatively small.

Information preservation. The preservation of information is this technique's main advantage. While the output dataset is typically manageable in terms of size due to the group representation, the contents of the groups can be looked up at all times, resulting in virtually no loss of information.

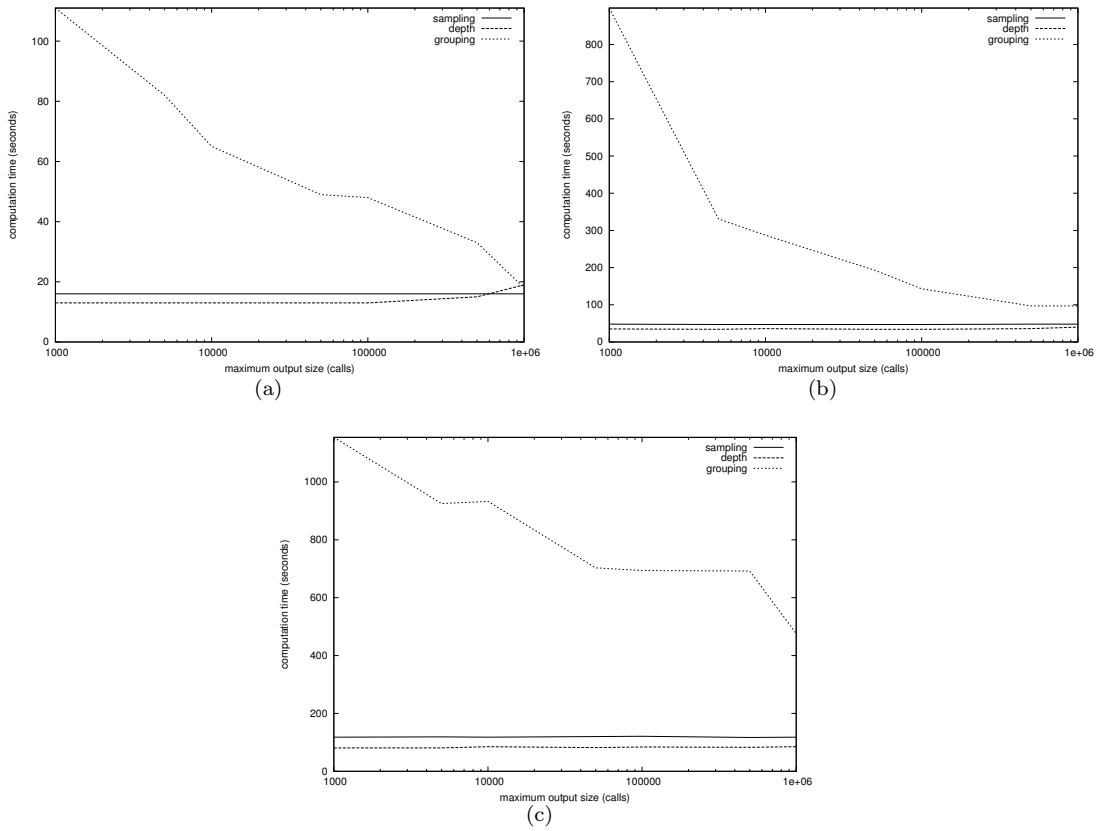

Figure 3. Computation time spent during each run, using the different abstraction techniques. (a) checkstyle-3checks; (b) azureus-newtorrent; (c) ant-selfbuild.

Reliability. As with the depth limitation technique, it is difficult to predict the effectiveness of the grouping technique since it is dependent on the stack depth distributions in traces. Several traces discussed in this paper were reduced to small numbers of large groups. The effect of the gap size is discussed in more detail in the next section.

6.3.1. Effect of the gap size

Increasing the gap size proves extremely effective: using a value of 2, five out of seven traces are reduced to less than 10% of their original size. Using a gap size of 4, the same holds for the remaining two traces, with four traces even being reduced by a factor 100 or more. Setting the gap size to 5 shows a rather sudden increase in compression ratio for the JPacman and

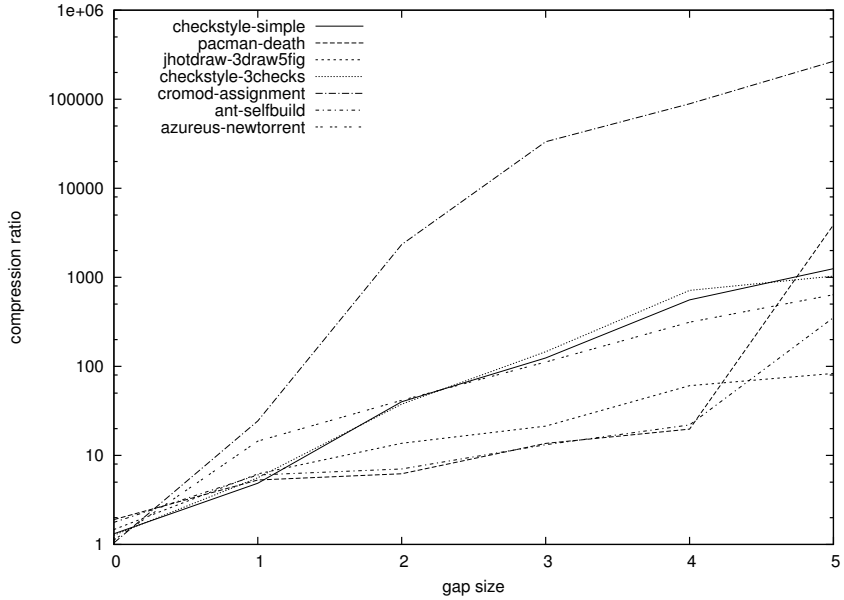

Figure 4. Effect of the grouping technique's gap size on the compression ratio.

Apache Ant traces: this suggests that these traces contain many similar, contiguous groups of calls that are intermitted by stack depth differences of size 5.

What is particularly noteworthy is the result for Cromod: the compression ratio of this trace turns out to be extremely sensitive to increased gap sizes, with a ratio of almost 100,000 at a gap size of 4. Since this trace comprises several hundred thousands of calls, the only logical conclusion to be drawn here is that a large segment of the trace has ended up in one group due to small depth differences within this segment. This is confirmed by Figure 1(d), as this graph indicates that the depth differences are generally low throughout the entire Cromod execution.

6.4. Threats to validity

Our selection of three light-weight techniques was based on three criteria that we considered relevant in most trace comprehension contexts. However, as we mentioned earlier, the criteria being used are generally dependent on the software development task that is to benefit from the abstraction technique. The fact that other tasks require different selection criteria can be considered a threat to external validity.

In addition, a potential threat to the internal validity concerns the execution traces that were used in this paper. We have drawn several conclusions based on the properties of the traces at hand, e.g., systems with multiple threads running the risk that stack depth-related abstractions have limited applicability. Such implications may not hold true for all programs

Table VIII. Comparison of the techniques using three distinct evaluation criteria.

	Sampling	**Depth limitation**	**Grouping**
computational complexity	++	++	−
information preservation	−−	+	++
reliability	++	−−	−

and traces, as thread behavior can vary from system to system, and be registered using different extraction methods. Another example concerns the common baselines that we found in the GUI-based applications: more systems need to be tested to know for certain whether there exists a correlation.

Finally, a threat to the construct validity might be the fact that the techniques in our example assessment are not very difficult to quantify and compare in terms of such metrics as compression ratio, as they view traces as call series that need to be reduced. Things get more complicated if other factors come into play, for example, when a technique relies heavily on domain knowledge. Other contexts may also require different evaluation criteria than the ones used in our example: the assessment of a memory-intensive technique, for example, warrants the notion of spatial complexity.

We see no threats to the conclusion validity as the measurements are reliable due to automation and objective definition via metrics. Furthermore, the conclusions do not depend on assumptions nor power of statistical tests.

7. Related Work

There is a great deal of ongoing research that incorporates the use of dynamic analysis in various software development tasks. In particular, in the field of program comprehension many abstraction and visualization techniques have been proposed to facilitate such tasks as maintenance issues and change requests. This section outlines the work that is the most closely related to this paper's topics.

7.1. Trace characterization

Hamou-Lhadj and Lethbridge extract certain properties from a set of execution traces [6]. There are three main differences with our trace characterization. First, they discuss multiple *scenarios* from three systems. Rather than analyzing multiple traces for each system, we chose to maintain a broader perspective by selecting more different systems (six in total) in which a distinction is made between GUI-based programs and batch execution systems. Second, the traces that they analyze do not take private methods into account and are generally smaller than ours, averaging 140,000 events. Third, we have emphasized the use of stack depths during executions, whereas their analysis exhibits a strong focus on recurrent patterns. We have not treated such patterns because we feel that (1) the detection of patterns (i.e., isomorphic

subgraphs) is too expensive for traces containing millions of events, and (2) the choice between the various matching criteria involved therein deserves an assessment of its own.

Additionally, since in our paper the focus is not on making traces more easily navigable but rather on reducing the amounts of data, we employ the notion of *compression* ratio rather than the *collapse* ratio used in [6].

7.2. Trace abstraction

Kuhn and Greevy compare feature traces by representing them as "signals in time" [16]. They propose the notion of monotone subsequence summarization that concerns the grouping of calls. Since their work only briefly mentions the effects of the gap size, in our paper we have elaborated on the quantitative aspects of this technique. Similar work by Zaidman and Demeyer [20] uses the relative frequency of method executions to compare regions in traces, as opposed to using stack depths.

In earlier work we have reconstructed UML sequence diagrams from test case executions [8]. To cope with the issue of large amounts of data we proposed a series of abstraction techniques, among which is the stack depth limitation technique discussed in Section 3.2. While the preliminary results for medium-sized traces were promising, our quantification of this technique in Table VI sheds more light on its effectiveness in the context of large traces.

Hamou-Lhadj et al. have published several papers that concern the summarization of execution traces. They report on a technique to recover behavioral design models from execution traces [17], in which they determine which classes are utility classes, and identify the classes that have a high fan-in and a low (or non-existent) fan-out. Once these classes are removed from the trace, the resulting trace is visualized in the Use Case Map (UCM) notation. They also applied this "utilityhood" measure in another paper [18] in which a large execution trace is automatically summarized.

The same authors report on an algorithm for the detection of recurrent patterns in traces [19]. A set of criteria is proposed to group the patterns that were found. To the same purpose, Systä et al. utilize the Boyer-Moore string matching algorithm [24].

7.3. Trace visualization

There have been many trace visualization techniques and tools in the past decades. These include Jinsight by De Pauw et al. [21], who propose the "execution pattern notation" as an abstraction measure. Lange and Nakamura present Program Explorer [22], a tool that offers multiple trace views while employing merging, pruning, and slicing techniques to render traces navigable. Another well-known tool is ISVis by Jerding et al. [23], who introduced the "information mural" to facilitate the navigation through traces. Systä et al. present Shimba [24], which summarizes recurrent patterns in terms of a sequence diagram visualization. Other techniques include Reiss's Jive tool for online dynamic analysis [25], and the polymetric views by Ducasse et al. [26].

More recently, Greevy et al. [27] proposed a 3D visualization of a software system's execution that displays the large amounts of dynamic information as growing towers, which become taller as more instances of a type are created. Furthermore, in earlier work we presented Extravis [9],

which uses a circular view to depict a system's structural elements and a bundling technique to prevent cluttering among the call relations between these elements.

The use of abstraction mechanisms such as the ones discussed in this paper can significantly reduce the input data for the visualization tools listed above, thus rendering thorough analyses of such techniques extremely useful.

8. Conclusions & Future Work

The use of dynamic analysis in program comprehension tasks has become increasingly popular. Many approaches and tools have been developed in an attempt to tackle the scalability issues that arise during the analysis of execution traces. Unfortunately, reports on the techniques being offered (1) typically involve limited sets of representative systems and traces, (2) often do not elaborate on the precise effects of the parameters, and (3) seldomly involve extensive assessments and comparisons of those techniques in terms of such properties as compression ratio and information loss.

We have addressed these issues by characterizing a set of seven large execution traces from six different systems, with the intent of identifying the key properties that play an important role in the application of abstraction techniques. We then proposed an assessment methodology aimed at the quantitative evaluation and comparison of trace abstraction techniques. Using the seven large traces, we applied our methodology on a selection of three abstraction techniques, being sampling, stack depth limitation, and grouping.

The results of our assessment indicate that the sampling technique is both fast and reliable, but filters information in an arbitrary fashion. Furthermore, our measurements suggest that the depth limitation and grouping techniques largely depend on the composition of a trace in terms of its nesting behavior, with the grouping technique preserving the most information at the cost of scalability.

Our findings by no means imply that other trace characteristics or properties of the software systems under study are of no importance: rather, we conclude that trace abstraction techniques are not universally applicable in practice, and that program and trace properties must be studied if a sensible choice of abstraction techniques is to be made. Furthermore, in order to gain a complete picture of a technique's applicability in a particular context, we advocate the extension of our methodology with a qualitative aspect.

The work described in this paper makes the following contributions:

- The characterization of a set of seven large execution traces that were extracted from six different object-oriented systems.
- The proposition of an assessment methodology for the evaluation and comparison of the quantitative aspects of trace abstraction techniques.
- The application of this methodology through the implementation, evaluation, and comparison of three light-weight trace abstraction techniques used in literature, being sampling, stack depth limitation, and grouping.

8.1. Future work

As a direction for future work we consider extending our assessment methodology with a *qualitative* aspect. While certain quantitative properties of an abstraction technique are crucial and warrant an assessment of their own, the practical applicability of such a technique also depends greatly on the program comprehension task at hand and on the nature of the information that is being lost during filtering. For example, the filtering of events through sampling is sensible if the goal is the visualization of a program's high-level phases, but is not acceptable in the context of bug detection because of the loss of arbitrary information.

Furthermore, our experiments have shown that the presence of multiple threads in a system severely complicates the analysis of its traces. This makes it difficult to determine which abstractions are potentially suitable in such cases. Future work includes a thorough investigation of this issue: for example, one could consider to filter each thread's "startup" event, so as to prevent the stack depth from unnecessarily increasing in case of many threads (e.g., Figure 1(f)). This restores the possibility of using such techniques as stack depth limitation.

REFERENCES

1. Corbi TA. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 1989; **28**(2):294–306.
2. Mayrhauser Av, Vans AM. Program comprehension during software maintenance and evolution. *IEEE Computer* 1995; **28**(8):44–55.
3. LaToza TD, Venolia G, DeLine R. Maintaining mental models: a study of developer work habits. *Proc. 28th Int. Conf. on Software Engineering (ICSE)*, ACM, 2006; 492–501.
4. Ball T. The concept of dynamic analysis. *ACM SIGSOFT Software Eng. Notes* 1999; **24**(6):216–234.
5. Zaidman A. Scalability solutions for program comprehension through dynamic analysis. PhD Thesis, University of Antwerp 2006.
6. Hamou-Lhadj A, Lethbridge TC. Measuring various properties of execution traces to help build better trace analysis tools. *10th Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 2005; 559–568.
7. Briand LC, Labiche Y, Leduc J. Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Trans. Software Eng.* 2006; **32**(9):642–663.
8. Cornelissen B, Deursen Av, Moonen L, Zaidman A. Visualizing testsuites to aid in software understanding. *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2007; 213–222.
9. Cornelissen B, Holten D, Zaidman A, Moonen L, Wijk JJv, Deursen Av. Understanding execution traces using massive sequence and circular bundle views. *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, IEEE, 2007; 49–58.
10. Zaidman A, Calders T, Demeyer S, Paredaens J. Applying webmining techniques to execution traces to support the program comprehension process. *Proc. 9th European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2005; 134–142.
11. AspectJ: The AspectJ project at Eclipse.org, `http://www.eclipse.org/aspectj/`.
12. De Pauw W, Lorenz D, Vlissides J, Wegman M. Execution patterns in object-oriented visualization. *Proc. 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, USENIX, 1998; 219–234.
13. Hamou-Lhadj A, Lethbridge TC. A metamodel for dynamic information generated from object-oriented systems. *Electr. Notes Theor. Comput. Sci.* 2004; **94**:59–69.
14. Chan A, Holmes R, Murphy GC, Ying ATT. Scaling an object-oriented system execution visualizer through sampling. *Proc. 11th Int. Workshop on Program Comprehension (IWPC)*, IEEE, 2003; 237–244.
15. Rountev A, Connell BH. Object naming analysis for reverse-engineered sequence diagrams. *Proc. 27th Int. Conf. on Software Engineering (ICSE)*, ACM, 2005; 254–263.
16. Kuhn A, Greevy O. Exploiting the analogy between traces and signal processing. *Proc. 22nd Int. Conf. on Software Maintenance (ICSM)*, IEEE, 2006; 320–329.

17. Hamou-Lhadj A, Braun E, Amyot D, Lethbridge TC. Recovering behavioral design models from execution traces. *Proc. 9th European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2005; 112–121.
18. Hamou-Lhadj A, Lethbridge TC. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, IEEE, 2006; 181–190.
19. Hamou-Lhadj A, Lethbridge TC. An efficient algorithm for detecting patterns in traces of procedure calls. *Proc. 1st ICSE Int. Workshop on Dynamic analysis (WODA)*, IEEE, 2003; 1–6.
20. Zaidman A, Demeyer S. Managing trace data volume through a heuristical clustering process based on event execution frequency. *Proc. 8th European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2004; 329–338.
21. De Pauw W, Helm R, Kimelman D, Vlissides JM. Visualizing the behavior of object-oriented systems. *Proc. 8th Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, 1993; 326–337.
22. Lange DB, Nakamura Y. Object-oriented program tracing and visualization. *IEEE Computer* 1997; **30**(5):63–70.
23. Jerding DF, Stasko JT, Ball T. Visualizing interactions in program executions. *Proc. 19th Int. Conf. on Software Engineering (ICSE)*, ACM, 1997; 360–370.
24. Systä T, Koskimies K, Müller H. Shimba - an environment for reverse engineering Java software systems. *Software - Practice and Experience* 2001; **31**(4):371–394.
25. Reiss SP. Visualizing Java in action. *Proc. Symp. on Software Visualization (SOFTVIS)*, ACM, 2003; 57–65.
26. Ducasse S, Lanza M, Bertuli R. High-level polymetric views of condensed run-time information. *Proc. 8th European Conf. on Software Maintenance and Reengineering (CSMR)*, IEEE, 2004; 309–318.
27. Greevy O, Lanza M, Wysseier C. Visualizing live software systems in 3D. *Proc. Symp. on Software Visualization (SOFTVIS)*, ACM, 2006; 47–56.

TUD-SERG-2008-005
ISSN 1872-5392