# An Assessment Methodology for Trace Reduction Techniques*

**Bas Cornelissen**
*Delft University of Technology*
*The Netherlands*
*s.g.m.cornelissen@tudelft.nl*

**Leon Moonen**
*Simula Research Laboratory*
*Norway*
*Leon.Moonen@computer.org*

**Andy Zaidman**
*Delft University of Technology*
*The Netherlands*
*a.e.zaidman@tudelft.nl*

## Abstract

*Program comprehension is an important concern in software maintenance because these tasks generally require a degree of knowledge of the system at hand. While the use of dynamic analysis in this process has become increasingly popular, the literature indicates that dealing with the huge amounts of dynamic information remains a formidable challenge.*

*Although various trace reduction techniques have been proposed to address these scalability concerns, their applicability in different contexts often remains unclear because extensive comparisons are lacking. This makes it difficult for end-users to determine which reduction types are best suited for a certain analysis task.*

*In this paper, we propose an assessment methodology for the evaluation and comparison of trace reduction techniques. We illustrate the methodology using a selection of four types of reduction methods found in literature, which we evaluate and compare using a test set of seven large execution traces.*

*Our approach enables a systematic assessment of trace reduction techniques, which eases the selection of suitable reductions in different settings, and allows for a more effective use of dynamic analysis tools in software maintenance.*

## 1. Introduction

The use of dynamic analysis has become increasingly popular in various stages of the software development process. Among the areas of interest is program comprehension, which constitutes an essential part of many maintenance tasks [2, 6]: the engineer must sufficiently understand the program at hand before any action can be undertaken. In doing so, a mental map is built that bridges the gap between the program's high-level concepts and its source code [29, 19].

There exist various approaches to gain knowledge of a software system. Static analyses focus on such artifacts as source code and documentation, and potentially cover 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 [1]. One of the main issues with dynamic techniques, however, is the huge amounts of data that need to be analyzed [30].

In recent years, many solutions have been proposed to tackle the scalability issues that are associated with large execution traces. Unfortunately, an effective comparison of such techniques is hampered by three factors. First, the evaluations of the techniques by their authors mostly concern limited numbers of software engineering contexts. Second, the evaluation criteria being used across these evaluations are typically different. Third, different researchers use their own sets of execution traces to evaluate their techniques on, i.e., no two techniques have been tested on one and the same trace. As a consequence, the evaluation results have limited generalizability, which makes it unclear for an engineer which reduction technique best fits a particular context.

In this paper, we propose an assessment methodology for trace reduction techniques. The purpose of this methodology is to enable the community to subject such techniques to a systematic evaluation process, in order to provide end-users with sufficient information to choose the most suitable technique in their respective contexts. We illustrate our methodology by applying it on a selection of trace reduction techniques encountered in literature, which we evaluate and compare using context-specific criteria. We argue how such assessments enable the reasoning about the applicability of a reduction technique in certain analysis contexts, which leads to a more effective use of dynamic analysis tools during maintenance tasks.

The remainder of this paper is organized as follows. Section 2 provides an outline of the problem area and the challenges involved therein. Next, in Section 3, we elaborate on our assessment methodology. We then illustrate the use of this methodology by discussing four existing reduction techniques in Section 4, which we then assess in Section 5. Our findings are discussed in Section 6, after which we present conclusions and future directions in Section 7.

## 2. Background

Our intent to support software engineers in discerning the most effective reduction techniques in specific contexts is motivated by the research community's growing interest in dynamic analysis. These analyses are often characterized by huge amounts of data: Reiss and Renieris, for example, re-

*This work was conducted in the Software Evolution Research Lab at Delft University of Technology as part of the Reconstructor project, sponsored by NWO/Jacquard.

| Summarization | Metrics-based filtering | Language-based filtering | Ad hoc |
|---|---|---|---|
| execution pattern notation [11] | frequency spectrum analysis [1, 32] | package filtering [13, 25] | sampling [4, 12] |
| pattern summarization [21, 27, 15, 24] | utilityhood measure [14] | visibility specifiers [17] | fragment selection [25, 28] |
| object & event clustering [13, 22] | webmining [31] | getters & setters [17, 7] | ... |
| monotone subsequence summ. [18] | stack depth limitation [11, 7] | constructor hiding [17, 7] | |
| ... | ... | ... | |

**Table 1. Categories of automatic trace reduction techniques.**

port [21] on an experiment in which one gigabyte of trace data was generated for every two seconds of executed C/C++ code or every ten seconds of Java code.

Being able to cope with such amounts of run time data is beneficial to many areas in software engineering. These include such tasks as debugging and performance optimization, and tasks related to software understanding, such as feature analysis, trace understanding, and visualization. Unfortunately, in many such tasks, the analyses have upper bounds on the amount of data that can be handled. In earlier work, for example, we reconstructed UML sequence diagrams from event traces [7]. Clearly, from a cognitive point of view, such diagrams in themselves do not scale up to thousands of events. We then proposed novel visualization techniques [10] with a strong focus on scalability; still, the tool's performance deteriorates as the amount of data being visualized exceeds several hundred thousands of events.

The huge amounts of data involved in dynamic analysis necessitate the use of *trace reduction techniques* to render the information suitable for analysis. In this paper we consider (very) large traces, and therefore focus on automatic rather than manual techniques in achieving initial data reductions. Many such techniques have been proposed in literature over the recent years, each targeting different aspects of execution traces. To provide an overview of the approaches in literature, we distinguish four different categories:

(a) **Summarization** techniques attempt to shorten a trace by replacing part of its contents by more concise notations. Typical summarization targets include recurrent patterns.

(b) **Metrics-based filtering** is centered around the use of certain metrics. Examples of such metrics are the stack depth, and degrees of fan-in and fan-out.

(c) **Language-based filtering** techniques are targeted at the omission of such constructs as getters and setters, private methods, and so forth.

(d) **Ad hoc** approaches concern the use of "black-box" techniques that do not consider the trace contents.

Table 1 shows the categories, along with various example techniques and pointers to literature.

# 3. Assessment Methodology

The main issue with the reduction techniques being offered is that they are seldomly compared side-by-side by their respective authors. For lack of a common assessment framework, the different techniques are generally not evaluated

- in the same software engineering contexts;
- by the same evaluation criteria; and
- on the same test set (i.e., execution traces).

The absence of a benchmark hinders technical progress in this field [26], and engineers faced with large amounts of trace data have the difficult task of selecting the most suitable reduction technique(s) in their specific contexts.

To address this issue, we propose an assessment methodology that is aimed at the thorough evaluation and comparison of trace reduction mechanisms. Such assessments are important because they enable a side-by-side comparison of both existing and future techniques. The key aspect of our methodology is the use of a common context, common evaluation criteria, and common test set.

Given a set of trace reduction techniques that are to be assessed, our methodology distinguishes the following steps:

1. **Context:** the establishment of a context, i.e., a certain task, and the role of reduction techniques therein.

2. **Criteria:** the definition of a set of evaluation criteria that are relevant to the context.

3. **Metrics:** the definition of set of metrics that enables the reasoning about the techniques in terms of the aforementioned criteria.

4. **Test set:** the selection of a series of execution traces on which to evaluate the techniques.

5. **Application:** the application of the techniques on the test set while extracting the previously defined metrics.

6. **Interpretation:** the interpretation and comparison of the measurements, in terms of the evaluation criteria.

Our methodology is applicable in any context that involves the need for trace reductions, and to any of the trace reduction techniques in Table 1. Furthermore, the evaluation criteria can be chosen such that the end-user's requirements are met. Note that the first three steps of our methodology correspond, respectively, to the goal, the question, and the metric in the Goal-Question-Metric (GQM) paradigm [3].

The methodology can be used in various cases. Examples are the development of new (or more effective use of existing) analysis tools that require the reduction of certain amounts and types of trace data, and the development of

new reduction techniques that should be compared to existing solutions with respect to certain criteria. The use of our methodology in these cases ensures that the relevant aspects of reduction techniques can be properly compared, which helps end-users to estimate the applicability of those techniques in specific contexts.

# 4. Four Reduction Techniques

We demonstrate our methodology on a selection of four trace reduction techniques. Our choice for these particular techniques is motivated by the categorization in Section 2, in the sense that we select one technique from each of the four categories. The techniques under study are subsequence summarization, stack depth limitation, a combination of language-based filtering techniques, and sampling.

For lack of available implementations of these techniques, we have created versions of our own. These are based on the descriptions in literature; relevant details are provided below to ensure the reproducibility of our experiment. Finally, for reasons of scalability, our implementations process traces on the fly rather than reading them completely into memory.

## 4.1. Subsequence summarization

The first reduction mechanism that we put to the test is a summarization technique by Kuhn and Greevy [18]. It is based on the grouping of trace events according to some criterion, with each group (or "subsequence") being represented in the output trace by that group's first event. The projected result is a trace that generally contains a significantly smaller number of events. The authors of this technique have named it *monotone subsequence summarization*, and while they use it to represent traces as signals in time, the technique is essentially a trace reduction mechanism.

The grouping criterion used by this technique is based on nesting level differences between trace events: the algorithm assigns consecutive events that have equal or increasing nesting levels to the same group. As soon as a level decrease is encountered and the difference exceeds a certain threshold, called the *gap size*, a new group is initiated. Considering the fact that the nesting level typically fluctuates during the execution of a system, the number of resulting events is smaller than the number of original events, and can be controlled by changing the gap size. Our implementation follows an iterative approach: initially setting the gap size to 0, the algorithm repeatedly increments its value until the projected output size meets the requirements.

## 4.2. Stack depth limitation

The second technique is centered around metrics-based filtering and is called *stack depth limitation*. This form of reduction has been used both in static contexts [23] and in our earlier work [7], in which encouraging results were attained in the removal of implementation details from test case executions. The variant discussed here revolves around the definition of a *maximum* depth: events taking place at depths higher than this threshold are removed from the original trace. The maximum depth depends on the maximum size of the output trace and on the stack depth progression in the original trace, i.e., the program's nesting behavior.

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, by use of which the trace is consequently reduced.

## 4.3. Language-based filterings

From the third category of reduction mechanisms we consider a combination of *language-based filtering* techniques. Since initial experiments have pointed out that these techniques by themselves are generally not successful in the significant reduction of large traces, we consider *three* consecutive filtering steps:

(i) Removal of getters/setters and their control flow.
(ii) Removal of private and protected method calls.[1]
(iii) Removal of constructors and their control flow.

Depending on the maximum output size, either of these mechanisms can be applied "on demand" in the given order.

## 4.4. Sampling

The fourth category of reduction techniques is represented in our experiment by *sampling*, an ad hoc reduction method that is used, among others, by Chan et al. [4] in reducing the dynamic information used by their AVID visualizer. The variant that we use in our experiment is simple: given an execution trace, we keep every $n$-th event. We call $n$ the sampling distance, which is automatically determined based on the maximum size of the output trace.

# 5. Experimental Setup

Our demonstration assessment aims at a thorough evaluation of the trace reduction techniques in the previous section. The design of the experiment follows our methodology, with each of the six steps being described in the following sections.

## 5.1. Context

In this experiment, we consider a use case in which an engineer is faced with the task of understanding a system's execution through the *visualization* of its execution traces. We assume his main interest to concern events taking place at high and medium abstraction levels, i.e., low-level details are

[1] Note that we preserve the control flows of private and protected methods since these are generally of interest, e.g., private initialization and processing methods within a main method.

considered less important. To make this example representative of real life situations, we assume the traces at hand to contain several tens of thousands or even millions of events. Furthermore, the intended visualization offers the opportunity to understand the temporal aspects of the trace, and is interactive in the sense that one can dynamically alter the size of the input data if need be.

## 5.2. Evaluation criteria

The context of our experiment entails a set of requirements that must be sufficiently met by a candidate reduction technique. In particular, we distinguish three evaluation criteria: reduction success rate, performance, and information preservation. These criteria are largely representative of actual use cases in the sense that they are often applicable in practice, particularly the first and third criteria.

**Reduction success rate:** the degree to which the techniques attain the desired reductions. We say that a reduction fails if the size of a reduced trace does not satisfy some threshold on the output size. The reduction success rate is relevant, as it depends greatly on the trace aspects exploited by a technique, and the degree to which these aspects occur in the trace.

**Performance:** a measure for the computational effort that is involved in the reduction. This is relevant in our context because the interactive nature of the reference visualization implies that modifications of the trace data should be processed as quickly as possible. For example, if during an interactive session the engineer decides that the trace data should be reduced further, it is not desirable if it takes several minutes for the visualization to refresh its views.

**Information preservation:** the extent to which information from the original trace is kept after reduction. While the application of a reduction generally implies that certain information is lost, it is important to quantify this loss and to study how it relates to the information needed for the context.

We explore two directions for measuring information preservation. The first route involves a generic approach from information theory that does not use background information regarding the data that is compared; the second route concerns a domain-specific analysis of information preservation that is tailored to the comparison of traces, with respect to the context sketched earlier. In the latter case, we distinguish three *event types* in a trace: (1) high-level events, which intuitively correspond to the control routines in a trace; (2) low-level events, which intuitively correspond to implementation details (e.g., utilities); and (3) medium-level events that comprise the remainder and intuitively concern business logic.

## 5.3. Metrics

In order to reason about the relevant aspects of the reduction techniques in terms of the criteria discussed above, we define a set of metrics. The first two metrics below are directly related to the measurement of reduction success rate and performance, respectively. The last two metrics correspond to the two routes to measuring information preservation.

**Actual output size:** the actual size of the output dataset after reduction, in calls. This metric allows for a discussion on the reduction success rate in each run. The measurements reflect the degree to which the reduction was successful (if at all), on the basis of which an *average success rate* can be calculated for each technique. For example, if a trace must be reduced to 1,000 events, the success rate is 90% in case of an output of 900 events, and 0% if the reduction fails.

**Computation time:** the amount of time spent on the reduction, in seconds. This metric allows for a comparison of the techniques in terms of performance. Since the reduction 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 reduction.

**Normalized compression distance (NCD):** a generic similarity metric [5] that uses standard compression algorithms to compute a practical approximation of the non-computable but optimal "normalized information distance" (NID) [20]. This metric has its origins in the field of information theory and is based on the notion of Kolmogorov complexity. NCD has been successfully applied in various areas, ranging from text corpora to handwriting recognition, genome sequences, and pieces of music. The NCD can be used to measure information preservation: a reduced trace that is shown to have a high similarity to the original trace implies that little information has been lost.

**Preservation of events per type:** for each event type, we measure the percentage of events that remains after reduction, relative to the number of events in the original trace for that type. While there are various options for defining such types (e.g., utilityhood [17]), we define the high-, medium-, and low-level types without loss of generality as (1) events with no fan-in, (2) events with no fan-out, and (3) remaining events for our demonstration experiment. As events we consider the method signatures, and fan-in/fan-out rates are determined on the basis of the original trace.[2]

## 5.4. Test set

**Systems under study.** The test set in our example assessment consists of seven different execution traces from six different Java systems. For this test set to be as representative as possible, in our systems selection we have taken into account such characteristics as system size, typical trace size, and multithreading.

[2] Alternatively, one could use the system's *static* call graph, as in [17].

| Trace | System kLOC | # calls | # threads | Description |
|---|---|---|---|---|
| checkstyle-short | 57 | 31,237 | 1 | The processing of a small input file that contains 50 lines of commented Java code. |
| pacman-death | 3 | 139,582 | 1 | The start of a game, several player and monster movements, player death, start of a new game, and quit [8]. |
| jhotdraw-3draw5fig | 73 | 161,087 | 1 | 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 [10]. |
| cromod-assignment | 51 | 266,337 | 11 | The execution of a typical assignment that involves the calculation of greenhouse parameters for two days for one greenhouse section [10]. |
| checkstyle-3c | 57 | 1,173,914 | 1 | The processing of three Java source files that are between 500 and 1000 lines in size each. |
| azureus-newtorrent | 436 | 3,144,785 | 172 | The program's initialization, and invocation of the "new torrent" functionality on a small file before quitting. |
| ant-selfbuild | 99 | 12,135,031 | 1 | The execution of the program, having specified the non-trivial task of building Apache Ant itself [31]. |

**Table 2. Description of the test set.**

JPACMAN is a small application 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 graphical map while eating food and evading monsters.

CROMOD is a medium-size, multithreaded industrial system that regulates the environmental conditions in greenhouses. Given a set of sensor inputs at the command line, 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 events.

CHECKSTYLE[3] is a medium-size source code validation tool. From the command line it takes a set of coding standards to process one or more input files, while systematically looking for violations and reporting these to the user.

JHOTDRAW[4] is a medium-size tool for graphics editing. It was developed as a showcase for design pattern usage and is acknowledged to be well-designed. It provides a GUI that offers various graphical features such as the insertion of figures and drawings.

AZUREUS[5] is a large-size, multithreaded peer-to-peer client that implements the BitTorrent protocol. Its GUI can be used to exchange files by use of so-called torrents, which are files containing metadata on the files being exchanged.

APACHE ANT[6] is a medium-size, Java-based build tool. It is command line based and owes much of its popularity to its ability to work cross-platform. The execution trace for this system was obtained through fellow researchers [31].

**Execution scenarios.** For each system we define a typical execution scenario. We then instrument the systems, run the scenarios, and register the entries and exits of all constructor and (static) method calls on the class level, and the threads in which these events take place. This results in seven execution traces that size from several tens of thousands of events to several millions of events.[7] The descriptions and sizes of the traces are given in Table 2.

### 5.5. Application

Each of the four techniques is applied on all seven traces. The task being performed during each run is the reduction of the input trace while conforming to a certain *threshold*. The threshold is the maximum output size of the trace, and reflects use cases in which a certain degree of reduction is necessary for a certain task. An example of such a use case is a trace analysis method from earlier work [9], which can handle at most 50,000 events because it has complexity $O(n^2)$ with respect to the trace size $n$.

We employ seven different thresholds with values between 1,000 and 1,000,000 calls. This yields a total of 196 runs, which we perform on a Linux system with an Intel Pentium M 1.6 GHz processor and 2 GB of memory.

### 5.6. Interpretation

The final stage of the assessment concerns the interpretation of the results. By focusing on the measurements in each of the 196 runs, we discuss the results of the techniques in terms of our evaluation criteria. Finally, based on our observations, we conclude with a comparison of the techniques.

## 6. Results & Discussion

The results of the experiment are shown in Table 3, which shows the measurements for each of the four techniques across the relevant runs.[8] Reductions that were unsuccessful are denoted by dashes; furthermore, the percentages of preserved events have been rounded upwards so as to distinguish very small fractions from zeroes. Finally, the NCD

[3] Checkstyle 4.3, http://checkstyle.sourceforge.net/
[4] JHotDraw 6.0b, http://www.jhotdraw.org/
[5] Azureus 2.5.0.0, http://azureus.sourceforge.net/
[6] Apache Ant 1.6.1, http://ant.apache.org/

[7] The traces are available online and may be downloaded from http://swerl.tudelft.nl/bin/view/Main/TraceRepos.
[8] Runs with thresholds higher than the input trace sizes were omitted.

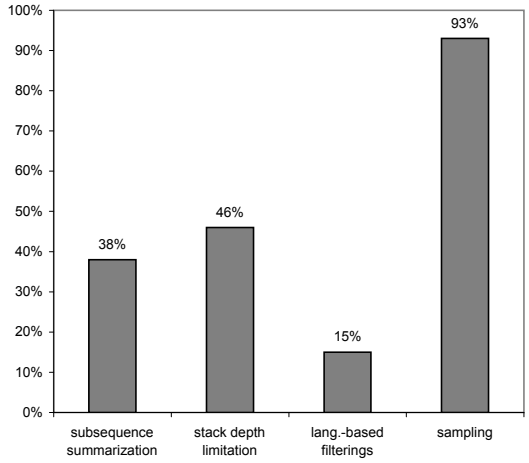


**Figure 1. Average reduction success rates.**

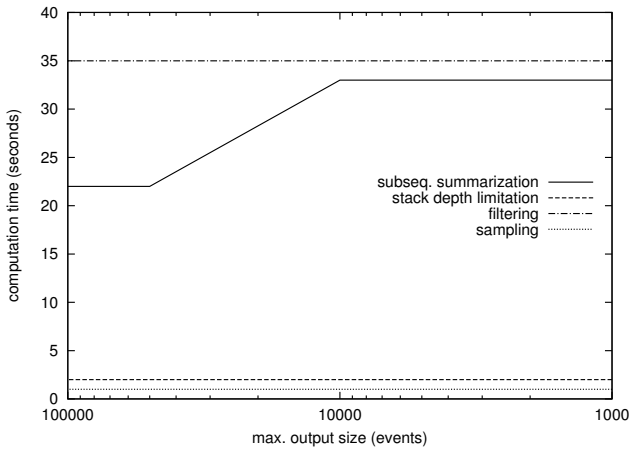


**Figure 2. Performance for the Cromod trace.**

values for information preservation were omitted in this table since they proved unreliable for the trace sizes used in the experiment; this is discussed in Section 6.3.

Figure 1 shows the average reduction success rate of each technique across the entire test set. The percentages are based on the measurements of all relevant runs.

Figures 2 through 4 demonstrate the performance of each technique in terms of computation time. We have selected the cases that exhibit the clearest differences (i.e., the largest traces) and that have high numbers of successful reductions, being `cromod-assignment`, `checkstyle-3c`, and `ant-selfbuild`. Note that the latter two diagrams employ logarithmic scales for the computation time.

Finally, Table 4 summarizes each technique's achievements relative to those of the other techniques.

In the following sections we discuss our findings, which are structured according to the three criteria.

## 6.1. Reduction success rate

In terms of the first evaluation criterion, we observe that the *sampling* technique achieved the best results: it is the only method that yielded successful reductions under all circumstances, with the output sizes mostly being close to the thresholds. This is presumably due to its ad hoc nature, as execution traces can always be sampled such that the maximum output size is satisfied, regardless of the size and composition of the trace.

The *summarization* and *stack depth limitation* techniques are both dependent on stack depth progression and show results that are similar to one another, with both methods mostly having difficulties with the `azureus-newtorrent` trace. The cause is most likely found in the abundance of active threads during this program's execution, in which (1) there occur many thread interactions, which hinders the grouping algorithm used during summarization; and (2) many threads exhibit low nesting levels, which renders depth limitations less effective. Furthermore, when faced with strict trace size limits, the summarization technique occasionally produces very small traces because in such cases the gap size is large out of necessity.

Finally, the combination of *language-based filtering* techniques proves disappointing with nearly half of the reductions having been unsuccessful. A noteworthy exception is the `cromod-assignment` trace, in which 98 out of every 100 events concern constructors, which are all filtered given any of the thresholds in Table 3.

Note that alternative definitions may be considered: e.g., a reduction may not necessarily have "failed" in case the result contains only several events too many.

## 6.2. Performance

With regard to performance, our measurements show that all four techniques were capable of reducing traces smaller than one million events within one minute (Figure 2). When looking at larger traces, however, there exist clear differences: here we observe that *sampling* easily outperforms any of the other techniques (Figures 3 and 4). We assume the principal cause to be that the sampling distance can be determined a priori, after which the trace needs to be processed only once. For the same reason, the computational effort involved in this approach is independent of the thresholds.

The same holds for the *stack depth limitation* technique, but here a trace must be processed twice because the stack depth frequencies must first be collected. Moreover, the interpretation of the stack depth at each event requires additional parsing effort in comparison to the black-box approach used by the sampling technique.

Concerning the *language-based filtering* techniques, there is little timing data available due to the many failed reductions. The data that is available, however, suggests that this approach is significantly slower than the aforementioned techniques. One can think of several reasons for this slowdown. Since the filterings are applied on demand and one by one, low thresholds require that the trace at hand is processed

| | | Subsequence summarization | | | | | Stack depth limitation | | | | | Language-based filterings | | | | | Sampling | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | max.output size | output size | comp.time | high % | medium % | low % | output size | comp.time | high % | medium % | low % | output size | comp.time | high % | medium % | low % | output size | comp.time | high % | medium % | low % |
| **checkstyle-short** | 10,000 | 6,417 | 2 | 59 | 29 | 12 | 4,251 | 0 | 100 | 12 | 16 | - | - | - | - | - | 7,809 | 0 | 12 | 25 | 26 |
| 31,237 calls | 5,000 | 781 | 4 | 36 | 4 | 2 | 4,251 | 0 | 100 | 12 | 16 | - | - | - | - | - | 4,462 | 0 | 0 | 16 | 13 |
| | 1,000 | 781 | 4 | 36 | 4 | 2 | 730 | 0 | 100 | 1 | 5 | - | - | - | - | - | 976 | 0 | 6 | 4 | 4 |
| **pacman-death** | 100,000 | 73,402 | 4 | 100 | 52 | 54 | 44,743 | 1 | 100 | 67 | 1 | 37,343 | 3 | 87 | 54 | 3 | 69,791 | 0 | 47 | 50 | 51 |
| 139,582 calls | 50,000 | 26,374 | 7 | 86 | 34 | 6 | 44,743 | 1 | 100 | 67 | 1 | 37,343 | 3 | 87 | 54 | 3 | 46,527 | 0 | 22 | 33 | 34 |
| | 10,000 | 7,080 | 18 | 34 | 11 | 1 | 172 | 1 | 100 | 1 | 1 | - | - | - | - | - | 9,970 | 0 | 6 | 8 | 8 |
| | 5,000 | 37 | 22 | 12 | 1 | 1 | 172 | 1 | 100 | 1 | 1 | - | - | - | - | - | 4,985 | 0 | 4 | 4 | 4 |
| | 1,000 | 37 | 21 | 12 | 1 | 1 | 172 | 1 | 100 | 1 | 1 | - | - | - | - | - | 997 | 0 | 3 | 1 | 1 |
| **jhotdraw-3draw5fig** | 100,000 | 25,965 | 12 | 45 | 26 | 10 | 83,584 | 1 | 100 | 58 | 48 | 61,906 | 5 | 80 | 59 | 27 | 80,543 | 0 | 51 | 51 | 50 |
| 161,087 calls | 50,000 | 25,965 | 12 | 45 | 26 | 10 | 42,748 | 1 | 100 | 26 | 25 | 44,383 | 17 | 42 | 47 | 17 | 40,271 | 0 | 26 | 26 | 25 |
| | 10,000 | 7,559 | 23 | 27 | 5 | 4 | 5,491 | 1 | 100 | 2 | 2 | - | - | - | - | - | 9,475 | 0 | 6 | 6 | 6 |
| | 5,000 | 2,659 | 29 | 18 | 4 | 1 | - | - | - | - | - | - | - | - | - | - | 4,881 | 0 | 4 | 4 | 4 |
| | 1,000 | 773 | 40 | 17 | 1 | 1 | - | - | - | - | - | - | - | - | - | - | 1,000 | 0 | 1 | 1 | 1 |
| **cromod-assignment** | 100,000 | 10,983 | 22 | 83 | 49 | 3 | 363 | 2 | 99 | 2 | 1 | 443 | 35 | 22 | 1 | 1 | 88,779 | 1 | 28 | 34 | 34 |
| 266,337 calls | 50,000 | 10,983 | 22 | 83 | 49 | 3 | 363 | 2 | 99 | 2 | 1 | 443 | 36 | 22 | 1 | 1 | 44,389 | 1 | 12 | 12 | 17 |
| | 10,000 | 207 | 33 | 74 | 1 | 1 | 363 | 2 | 99 | 2 | 1 | 443 | 35 | 22 | 1 | 1 | 9,864 | 1 | 4 | 4 | 4 |
| | 5,000 | 207 | 33 | 74 | 1 | 1 | 363 | 2 | 99 | 2 | 1 | 443 | 35 | 22 | 1 | 1 | 4,932 | 1 | 2 | 2 | 2 |
| | 1,000 | 207 | 33 | 74 | 1 | 1 | 363 | 2 | 99 | 2 | 1 | 443 | 35 | 22 | 1 | 1 | 997 | 1 | 1 | 1 | 1 |
| **checkstyle-3c** | 1,000,000 | 922,603 | 49 | 100 | 89 | 66 | 990,514 | 9 | 100 | 79 | 92 | 769,277 | 40 | 99 | 88 | 38 | 586,957 | 3 | 52 | 50 | 51 |
| 1,173,914 calls | 500,000 | 209,263 | 92 | 12 | 19 | 17 | 454,657 | 9 | 100 | 37 | 42 | - | - | - | - | - | 391,304 | 3 | 34 | 34 | 34 |
| | 100,000 | 31,233 | 135 | 7 | 3 | 3 | 49,834 | 9 | 100 | 4 | 5 | - | - | - | - | - | 97,826 | 2 | 8 | 9 | 9 |
| | 50,000 | 31,233 | 135 | 7 | 3 | 3 | 49,834 | 9 | 100 | 4 | 5 | - | - | - | - | - | 48,913 | 2 | 4 | 5 | 5 |
| | 10,000 | 8,032 | 177 | 2 | 1 | 1 | 2,061 | 9 | 100 | 1 | 1 | - | - | - | - | - | 9,948 | 2 | 2 | 1 | 1 |
| | 5,000 | 1,650 | 220 | 2 | 1 | 1 | 2,061 | 9 | 100 | 1 | 1 | - | - | - | - | - | 4,995 | 2 | 0 | 1 | 1 |
| | 1,000 | 1 | 304 | 2 | 0 | 0 | 743 | 9 | 100 | 1 | 1 | - | - | - | - | - | 1,000 | 2 | 0 | 1 | 1 |
| **azureus-newtorrent** | 1,000,000 | - | - | - | - | - | 657,657 | 25 | 100 | 20 | 54 | 217,671 | 185 | 69 | 6 | 17 | 786,196 | 7 | 26 | 25 | 25 |
| 3,144,785 calls | 500,000 | - | - | - | - | - | 451,133 | 25 | 100 | 13 | 33 | 217,671 | 183 | 69 | 6 | 17 | 449,255 | 7 | 15 | 15 | 15 |
| | 100,000 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 98,274 | 7 | 4 | 4 | 4 |
| | 50,000 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 49,917 | 7 | 2 | 2 | 2 |
| | 10,000 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 9,983 | 7 | 1 | 1 | 1 |
| | 5,000 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 4,999 | 7 | 1 | 1 | 1 |
| | 1,000 | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | 1,000 | 7 | 1 | 1 | 1 |
| **ant-selfbuild** | 1,000,000 | 921,549 | 1,589 | 100 | 14 | 4 | 768,246 | 86 | 100 | 8 | 6 | 111,650 | 193 | 100 | 1 | 2 | 933,463 | 24 | 0 | 8 | 8 |
| 12,135,031 calls | 500,000 | 34,432 | 2,338 | 100 | 1 | 1 | 434,088 | 86 | 100 | 4 | 4 | 111,650 | 191 | 100 | 1 | 2 | 485,401 | 24 | 0 | 4 | 5 |
| | 100,000 | 34,432 | 2,326 | 100 | 1 | 1 | 94,396 | 86 | 100 | 1 | 1 | 84,588 | 579 | 100 | 1 | 1 | 99,467 | 22 | 0 | 1 | 1 |
| | 50,000 | 34,432 | 2,326 | 100 | 1 | 1 | 47,793 | 87 | 100 | 1 | 1 | - | - | - | - | - | 49,938 | 22 | 0 | 1 | 1 |
| | 10,000 | 4,830 | 3,085 | 100 | 1 | 1 | 4,742 | 87 | 100 | 1 | 1 | - | - | - | - | - | 9,995 | 22 | 0 | 1 | 1 |
| | 5,000 | 4,830 | 3,073 | 100 | 1 | 1 | 4,742 | 87 | 100 | 1 | 1 | - | - | - | - | - | 5,000 | 22 | 0 | 1 | 1 |
| | 1,000 | 534 | 3,867 | 100 | 1 | 1 | 343 | 87 | 100 | 1 | 1 | - | - | - | - | - | 1,000 | 22 | 0 | 1 | 1 |

**Table 3. Measurement results for all four techniques.**

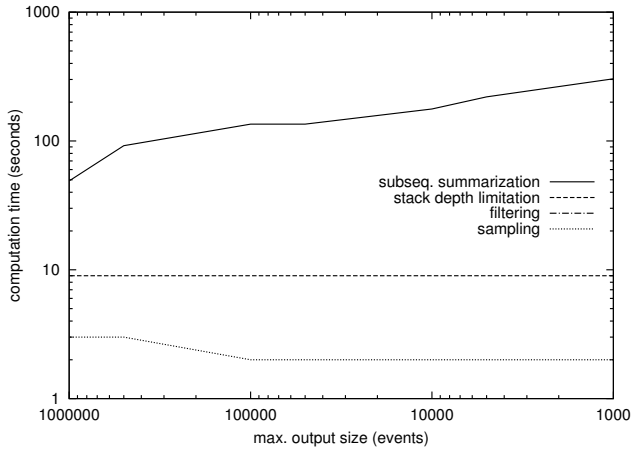


**Figure 3. Performance for the checkstyle-3c trace.**

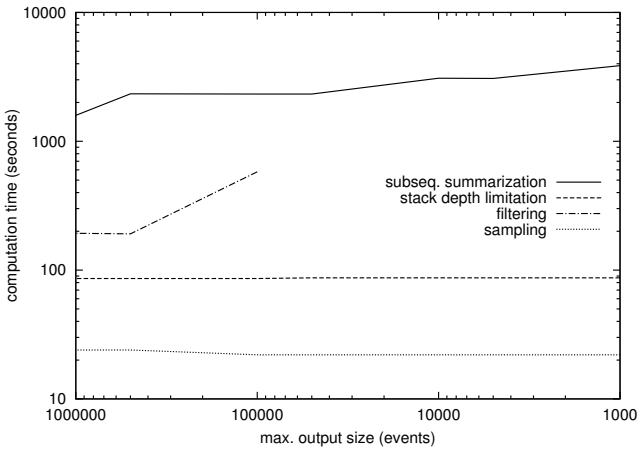


**Figure 4. Performance for the ant-selfbuild trace.**

up to four times, i.e., once for each filter type, and once more to read and write the traces. Moreover, the stack depths and signatures of each event must be parsed in order to acquire the information that is targeted by the filters.

The *subsequence summarization* technique typically requires a trace to be processed multiple times, as the gap size must be repeatedly incremented (starting at 0) until a suitable projected output size has been found. This iterative process yields significant overheads if the threshold is much smaller than the size of the trace, with the effort involved in each iteration being proportional to the trace size. Moreover, the number of necessary iterations also depends on the stack depth progression in the trace. The overall result is that the summarization approach is clearly the slowest technique in our experiment, particularly for large traces.

## 6.3. Information preservation

The assessment of our final criterion yields mixed results. Unfortunately, the values computed by the NCD metric proved unreliable in practice due to the trace sizes that were used in our experiment. To explain the issue, we need to provide some background on this metric. NCD is based on the notion that two objects are close to each other if we can significantly compress one object given the information in the other [5]. In practice, this translates to compressing the concatenation of the original and reduced traces and comparing its size to that of the compressed original trace. However, it turns out that standard compression tools split their input in "compression windows" within which the compression information is shared. As the size of the concatenation of the original and reduced trace exceeds the size of the compression window, that particular compressor can no longer be used to determine the NCD between those traces (since we are no longer compressing one object given the information in the other). Personal communication with R. Cilibrasi, the metric's first author [5], confirmed these issues and suggested their circumvention by writing a dedicated compressor. Since the metric only serves as an example in our experiment, this is left as a direction for future work.

For the domain-specific assessment of information preservation, we focus on high-level *and* medium-level events since those are required by our context (Section 5.1).

*Subsequence summarization* typically attains the best results: the percentages of preserved high-level events are significantly higher than those of the medium-level events which, in turn, are often higher than those of the low-level events. This is because each group is represented by its first event, and using our depth-based grouping criterion this event is likely to reside at relatively high levels.

The *stack depth limitation* and *language-based filtering* techniques show comparable results: the percentages of preserved high-level events are generally higher than those of other event types, with the depth limitation technique attaining the highest percentages in this respect. In several reductions, however, the fractions of preserved medium-level events are not always higher than those of low-level events. Examples are `checkstyle-3c` for depth limitation, and `azureus-new-torrent` for filtering. This implies that the use of these two techniques sometimes causes the preservation of low-level events at the cost of those at the medium-level, which is undesirable in the given context.

The *sampling* technique mostly exhibits similar fractions of preserved events across all three event types, particularly in large traces. This means that all event types are equally represented in the reduced traces. We attribute this to the technique's ad hoc nature, which implies that low-level events are neither identified, nor removed. This makes sampling the least useful technique in preserving high-level and medium-level events in our context.

On an interesting note, the measurements for the `ant-selfbuild` trace suggest that all of its high-level events are often preserved. However, it turns out that our definition of high-level events implies that this trace has only *one* high-level event.

| | Subsequence summarization | Stack depth limitation | Language-based filterings | Sampling |
|---|---|---|---|---|
| reduction success rate | o | o | − | + |
| performance | − | o | o | + |
| information preservation | + | o | o | − |

**Table 4. Assessment summary with respect to the example context.**

### 6.4. Threats to validity

A potential threat to the internal validity concerns the test set in our experiment. As with most evaluations in literature, certain implications were based on the properties of our test set, e.g., systems with multiple threads running the risk that stack depth-based reductions may have limited applicability (Section 6.1). Such observations do not necessarily hold true for any program or trace, as threading and nesting behavior can vary from system to system. We have addressed this issue by using a test set that is above average in terms of size and composition, and that contains systems and traces with different sizes and characteristics.

An additional threat to the internal validity concerns the fact that reduction techniques in literature could be subject to different interpretations. To address this threat, we have described our implementation choices to allow validation by others and to ensure the reproducibility of our results.

Concerning the external validity, we note that the reduction techniques considered in this paper are automatic in nature. The assessment of reduction methods is more difficult if other factors come into play; e.g., when a technique relies heavily on additional information, such as domain knowledge. Furthermore, most reduction techniques can be implemented in different manners: for instance, in terms of performance, the summarization algorithm used in our experiment could benefit from a higher initial gap size in case of large traces or low thresholds.

Finally, alternative contexts may require other evaluation criteria in addition to those used in our example assessment. For example, the evaluation of a memory-intensive technique warrants a discussion on spatial complexity. However, we argue that our example criteria are generic to a great extent: in particular, the notions of reduction success rate and information preservation are applicable in many alternative assessment contexts, which renders our experimental results useful in those cases.

## 7. Concluding Remarks

Program comprehension is an important aspect of the software development process. While the use of dynamic analysis in this process has become increasingly popular, such analyses are often associated with large amounts of trace data, which has lead to the development of numerous trace reduction techniques in recent years. Unfortunately, the different techniques being offered are generally not evaluated (1) in the same software engineering contexts, (2) by the same evaluation criteria, and (3) on the same test sets. As a result, it is often unclear to which extent a certain technique is applicable in a particular context, if at all.

We addressed this challenge by proposing an assessment methodology that uses a common context, common evaluation criteria, and a common test set to ensure that the reduction techniques under study can be properly compared. To illustrate its use in practice, we applied the methodology on a selection of four types of reduction techniques, being subsequence summarization, stack depth limitation, language-based filtering, and sampling. Using a test set of seven large execution traces (made available online), we evaluated and compared these approaches in terms of context-specific criteria, leading to an overview (Table 4) that is valuable for software maintainers in similar contexts.

In summary, the work described in this paper makes the following contributions:

- An assessment methodology for the evaluation and comparison of trace reduction techniques.
- The demonstration of this methodology through the implementation, evaluation, and comparison of four types of trace reduction techniques used in literature.

### 7.1. Future work

As a direction for future work we consider applying our methodology to additional traces and reduction techniques. In particular, we seek to determine the extent to which the assessment results can be generalized, i.e., whether the achievements of a technique are representative for other techniques in the same category (Table 1). This includes the use of larger test sets and the consideration of alternative contexts, which could involve different evaluation criteria (e.g., with more emphasis on *qualitative* aspects).

Another direction for future work concerns adapting the compressor that is used to compute the NCD metric, such that it no longer suffers from the "compression window" limitations that were discussed in Section 6.3. This enables its applicability to realistically-sized traces, and renders it an interesting alternative for measuring information preservation.

Finally, we seek to investigate whether certain trace characteristics (similar to those in [16]) can help in predicting the effectiveness of certain reduction techniques. The CROMOD trace in our experiment is a good example, as its many constructors were the key to the success of the constructor filtering technique in that case.

# References


[1] T. Ball. The concept of dynamic analysis. *ACM SIGSOFT Softw. Eng. Notes*, 24(6):216–234, 1999.

[2] V. R. Basili. Evolving and packaging reading technologies. *J. Syst. Software*, 38(1):3–12, 1997.

[3] V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Softw. Eng.*, pages 528–532. Wiley, 1994.

[4] A. Chan, R. Holmes, G. C. Murphy, and A. T. T. Ying. Scaling an object-oriented system execution visualizer through sampling. In *Proc. 11th Int. Workshop on Program Comprehension (IWPC)*, pages 237–244. IEEE, 2003.

[5] R. Cilibrasi and P. Vitányi. Clustering by compression. *IEEE Trans. on Information Theory*, 51(4):1523–1545, 2005.

[6] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[7] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proc. 11th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE, 2007.

[8] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. 15th Int. Conf. on Program Comprehension (ICPC)*, pages 49–58. IEEE, 2007.

[9] B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. In *Proc. Int. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 6–10. Tech. report TUD-SERG-2007-022, Delft Univ. of Techn., 2007.

[10] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Software*, 2008. To appear.

[11] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS)*, pages 219–234. USENIX, 1998.

[12] P. Dugerdil. Using trace sampling techniques to identify dynamic clusters of classes. In *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 306–314, 2007.

[13] J. Gargiulo and S. Mancoridis. Gadget: A tool for extracting the dynamic structure of Java programs. In *Proc. 13th Int. Conf. on Softw. Eng. & Knowledge Eng. (SEKE)*, pages 244–251. IEEE, 2001.

[14] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. C. Lethbridge. Recovering behavioral design models from execution traces. In *Proc. 9th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 112–121. IEEE, 2005.

[15] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proc. 10th Int. Workshop on Program Comprehension (IWPC)*, pages 159–168. IEEE, 2002.

[16] A. Hamou-Lhadj and T. C. Lethbridge. Measuring various properties of execution traces to help build better trace analysis tools. In *10th Int. Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 559–568. IEEE, 2005.

[17] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, pages 181–190. IEEE, 2006.

[18] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proc. 22nd Int. Conf. on Software Maintenance (ICSM)*, pages 320–329. IEEE, 2006.

[19] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proc. Int. Conf. on Software Eng. (ICSE)*, pages 492–501. ACM, 2006.

[20] M. Li, X. Chen, X. Li, B. Ma, and P. Vitányi. The similarity metric. *IEEE Trans. on Information Theory*, 50(12):3250–3264, 2004.

[21] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. 23rd Int. Conf. on Software Engineering (ICSE)*, pages 221–230. ACM, 2001.

[22] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proc. 6th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 47–55. IEEE, 2002.

[23] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *Proc. Int. Conf. on Software Engineering (ICSE)*, pages 254–263. ACM, 2005.

[24] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. 14th Int. Conf. on Program Comprehension (ICPC)*, pages 84–88. IEEE Computer Society, 2006.

[25] M. Salah and S. Mancoridis. A hierarchy of dynamic software views: From object-interactions to feature-interactions. In *Proc. 20th Int. Conf. on Software Maintenance (ICSM)*, pages 72–81. IEEE, 2004.

[26] S. E. Sim, S. M. Easterbrook, and R. C. Holt. Using benchmarking to advance research: A challenge to software engineering. In *Proc. 25th Int. Conf. on Software Engineering (ICSE)*, pages 74–83. IEEE, 2003.

[27] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software - Practice and Experience*, 31(4):371–394, 2001.

[28] A. Vasconcelos, R. Cepêda, and C. Werner. An approach to program comprehension through reverse engineering of complementary software views. In *Proc. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 58–62. Tech. report 2005-12, University of Antwerp, 2005.

[29] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55, 1995.

[30] A. Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp, 2006.

[31] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proc. 9th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 134–142. IEEE, 2005.

[32] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proc. 8th European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 329–338. IEEE, 2004.