Summarizing Traces as Signals in Time

Adrian Kuhn and Orla Greevy
Software Composition Group
University of Bern, Switzerland
{akuhn, greevy}@iam.unibe.ch

Abstract

One of the key challenges of dynamic analysis approaches is that they imply a huge volume of data, thus making it difficult to extract high level views. In this paper we describe a novel approach to trace summarization by visually representing entire traces as signals in time. Our technique produces a visualization of the complete feature space of a system that fits on one page. The focus of our work is to visually represent individual traces feature behavior. We assume a one-to-one mapping between features and traces. We apply the approach on a case study, and discuss how our visualization supports the reverse engineer to identify patterns in traces of features. Moreover, we show how the visual analysis of our trace signals reveals that assumed one-to-one mappings between features and traces may be flawed.

Keywords: reverse engineering, dynamic analysis, trace summarization, features, visualization.

1. Introduction

Reverse engineering usually implies the abstraction of high level views that represent different aspects of a software system. Object-oriented systems are difficult to understand by browsing the source code due to language features such as inheritance, dynamic binding and polymorphism. The behavior of the system can only be completely determined at runtime. The dynamics of the program in terms of object interactions, associations and collaborations enhance system comprehension [11]. Typically dynamic analysis involves instrumenting a program under investigation to record its runtime events. The context of our dynamic analysis is feature-centric reverse engineering (i.e. we exercise a system's features on an instrumented software system and capture traces of their runtime behavior).

Interpretation of execution traces is difficult due to their sheer size, thus filtering or compressing the data is a crucial step in the construction of high level views. The main challenge of trace summarization is to reduce the volume of data without loss of information that is relevant for a particular analysis goal [9]. The results of dynamic analysis are often presented as visualizations for better understanding of programs [7]. Dynamic analysis together with program visualization may be used in debugging, evaluating and improving program performance and in understanding program behavior.

Because of the accuracy and speed with which the human visual system works, graphic representations make it possible for large amounts of information to be displayed in a small space. By making a visual representation for the millions of events that make up the feature traces of an application, quickly discernible relationships and patterns can be obtained.

In this paper, we describe a novel visualization approach for dynamic analysis that draws an analogy between execution traces and signals in time. We use the nesting level to visualize traces as time plots, and provide a visualization that allows up to two a dozen feature traces to be displayed simultaneously on a single screen. We show evidence of the visualization's usefulness and how it supports the reverse engineer to interpret and reason about the dynamic information. In particular, we address the following reverse engineering questions:

- *How do we fit a visualization of many traces on one screen?*
- *Can we detect patterns of activity in the traces?*
- *Do our traces reveal flaws in our definition of features?*

We use SmallWiki [5] as an example case study. The same SmallWiki case study has been analyzed in a previous work by Greevy *et al.* with a metrics-based approach [8]. For this paper we traced a total of 18 feature traces.

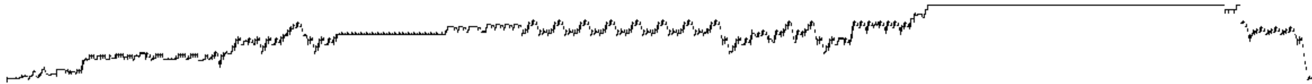

Figure 1. An execution trace as signal, showing on the y-axis the nesting level and along the x-axis the sequence of execution; summarized with Monotone Subsequence Summarization using gap size 0.

Structure of the paper. In Section 2 we draw the analogy between traces and times series, and visualize trace signals as time plots. Based on that in Section 3 we introduce a new trace summarization technique. Section 4 dives into the details of a case study, while Section 5 discusses the pros and cons of the time series analogy and the findings of our analysis. In Section 6 we provide a brief overview of related work in the fields of dynamic analysis, visualization and summarization of execution traces and time series related work. Finally we outline our conclusions in Section 7.

2. Traces are Signals in Time

As any chronological sequence of partially ordered data points qualifies as a *signal*, we can draw analogy between traces and signal processing. This done, we treat traces as if they were signals in time, which in turn provides us with a rich toolkit of well-established and ready-to-use algorithms from the field of signal processing and time series.

A key benefit of treating traces as signals is that we get time plots for free, as shown in Figure 1. Time plots are well-known from a broad range of applications, such as from the field of meteorology or stock markets. They show the change of a signal over time.

A *feature trace* is a record of the steps a program takes during the execution of a feature. We adopt the definition of a feature as a user-triggerable functionality of a software system [6]. In the case of object-oriented applications, a trace records method calls, whereas for systems implemented in procedural programming languages, it records function calls. In this paper we adopt the object-oriented terminology: we consider each execution step as a message sent from the sender to the receiver, whereupon the receiver executes the method selected by the message.

As a formal definition of a *trace*, we use a chronological sequence T of one or more execution events. The call hierarchy imposes a tree structure on the sequence, each event e_n has zero or more child events, with e_{n+1} as its first child, if any. With this definition, the execution sequence is equivalent to a depth-first traversal of the call hierarchy. Further, to equip the events with a partial order, we define the *nesting level* $\mathrm{Level}(e_n)$ of an event e_n as its depth in the call hierarchy.

Typically, an execution event includes information about execution time. However, in this paper we omit execution times and retain only the order of events. As the rise and fall of the signal is preserved even if all events are spaced equally apart in time, we can ignore execution time of events without loss of generality. The outline of the signal remains the same independent of the interval between its data points. Therefore the summarization technique we present in Section 3 retains its results.

3. How to Summarize Traces

In this section we introduce a trace summarization technique, which is based on the representation of traces as time signals. We introduce a technique called *Monotone Subsequence Summarization*, which makes use of the fact that a trace signal is composed of monotone subsequences separated by pointwise discontinuities.

The structure of a trace signal as defined in the previous section is plain simple: beginning at the starting node the nesting level either

- increases step by step as each event calls its first child or
- stays constant as subsequent children of the same event are called, until
- we reach an event without children, in which case the nesting level suddenly drops as execution continues with the latest sibling of the previous events.

The first two cases are monotonally increasing subsequences, whereas the latter is a pointwise continuity.

To summarize a trace signal, we cut the signal at its pointwise discontinuities into monotone subsequences, and compress each such subsequence into a summarized event chain. Thus the summarization is considerably shorter than the raw trace signal, see Figure 2, and consists of method-call-chains instead of single method calls.

The *Monotone Subsequence Summarization* cuts a trace signal between each two consecutive events where the nesting level does not increase

$$\mathrm{Level}(e_n) \leqslant \mathrm{Level}(e_{n+1})$$

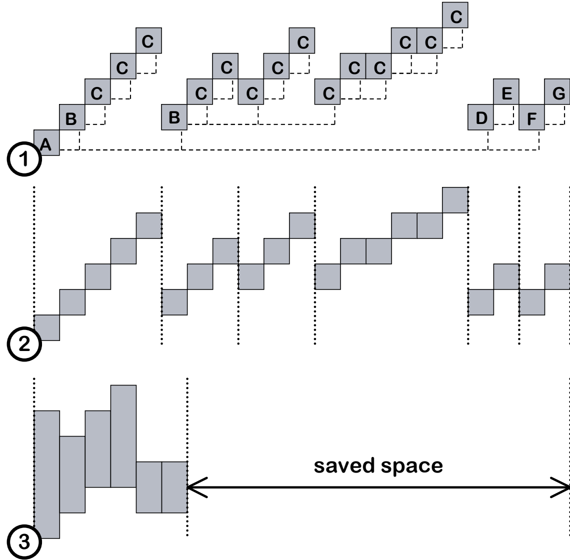

Figure 2. From top to bottom: 1) The outline of a trace, rotated by one quadrant such that time is on the X-axis. 2) We remove method names and retain the nesting level only, each discontinuity is marked with a dotted line. 3) Each monotone subsequence is compressed into one method-call-chain, saving a considerable amount of space.

into pieces $c_1 \dots c_m$ and these pieces become the events of the summarized trace. Furthermore, we define $\text{Level}(c_n)$ as the minimal nesting level within the chain c_n.

Using this technique, it is possible to reduce the length of a trace signal by about 50%. However, we can further improve this by taking into account that not each discontinuity has the same gap size: often the signal drops only by one or two nesting levels, that is the execution stays close to the current chain of method calls, whereas other discontinuities span dozens of nesting levels and thus mark a real break in the execution. Therefore, we allow small gaps within a chain of method calls and refine the above expression as

$$\text{Level}(e_n) - \text{Level}(e_{n+1}) \leqslant \text{gap_size}$$

Using gap_size = 3 it is possible to save up to 90% of the length of a trace signal. In other words even a trace with ten thousand events will fit on one screen.

4. Case study: SmallWiki

For our experiments with the techniques described in the previous sections, we chose *SmallWiki* [13], an open-source, fully object-oriented and extensible Wiki framework. A Wiki is a collaborative web application that allows users to add content, but also allows anyone to edit content. Thus SmallWiki provides features to create, edit and manage hypertext pages on the web.

We identify features of SmalWiki by associating features with the links and entry forms of the SmallWiki pages. We make the assumption that each link or button on a page triggers a distinct feature. We selected 18 distinct user interactions with the SmallWiki application and exercised them on an instrumented system to capture 18 distinct execution traces. The features we chose represent typical user interactions with the application such as login, editing a page or searching a web site. Then we apply trace summarization as described in Section 3, and we represent each feature trace visually as a time plot on one screen.

4.1. Analyzing the time plots of feature traces

In the following paragraphs we describe how we analyzed the time plots of the SmallWiki feature traces and reasoned about these views of feature behavior. As we have access to the developers of SmallWiki, we are able to check the findings of our signal processing analysis techniques with them.

Considering Figure 4, we observe a couple of phenomena exhibited by most or all trace signals of this feature spaces. We recognize that further research in the form of more case studies is required before we can conclude that these phenomena are common to any feature space, but we assume that these observations hold true for most trace signals.

All features share a common introduction. We observe that all features start with the same introduction, see Figure 3 annotation 1. This introduction corresponds to the time plot of the resolveURL feature. This makes sense due to the nature of SmallWiki as a web-based application. Resolving a given URL is the first step to be performed in order to execute a user-initiated feature. This phenomenon is most probably common to any feature space, as most architecture includes some top layer which does some preprocessing before executing the actual feature. This observation reveals that the traces could be further summarized by removing or *factoring out* the common introduction part of the trace.

Shared parts may include variations. Although the same introduction is shared by all trace signals, our analysis reveals variation points. In Figure 4 we see that the introduction sequences of the features copychild, addfolderchild, addpagechild and removechild include a variation point, that is they contain a distinct sequence which is not present in the other traces, see Figure 3 annotation 2.

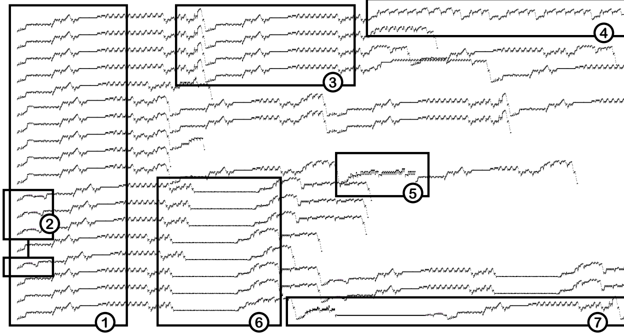

Figure 3. Trace patterns of Figure 4, there is a shared introduction (1) with slight variations (2); recurring patterns (3,6) as well as unique sequences (4,5,7).

Specific behavior is restricted to small hot-spots. Our analysis reveals that only a small amount of the overall behavior of a feature is specific to one sole feature (that is characterized as *single-feature*[8]). This might be due to the generic nature of the SmallWiki application. However we expect to observe this in other case studies as well, since most applications include sequences of common set-up and common tear-down enclosing the actual functionality of a feature. For example, as we exercised user-triggerable actions of SmallWiki, all involve exercising common functionality to handle the http request-response dialog of a web application. Recent work of de Pauw *et al.* in detection in patterns in traces reveals that the actual number of distinct patterns in execution traces was small. The results of their work revealed only 10 distinct patterns in a 40MB trace [3].

Some features are very similar. Our time series representation of feature traces as shown in Figure 4 reveals which features are closely related (that is they exhibit common patterns of behavior). The features copychild, addfolderchild, addpagechild and removechild are all invoked from the same page in SmallWiki, see Figure 3 annotation 6. We verified our findings with the developers and they confirmed that these features actually exercise the same code.

The features comps, props, stylesheets and edittemplate reveal similar time plots on Figure 4. Once again the developers confirmed our findings, as all these features are concerned with *look-and-feel* aspects of the system.

Not all features are equally similar. Figure 4 reveals that the features addFolder, addPage are similar. The feature *editPage* appears to be similar to the previous two features but then exhibits a strong variation, see Figure 3 annotation 7. The similar parts of these features indicate to the reverse engineer that these features may be conceptually related. The developers confirmed our findings.

5. Discussion

In this section we discuss some open issues and limitations of the applied techniques and the signal analogy itself.

On the choice of the nesting level as Y-axis. To represent traces as signal in time, we plot the nesting level of the execution events on the Y-axis. Even though this yields natural looking time plots, such as those familiar from meteorology or stock markets, we have not yet investigated if the nesting level is the most useful property to discriminate the differences between feature traces. Other information such as method names or arguments may prove to be more useful or better discriminators.

On the one-to-one mapping between features and traces. For this analysis, we assume a one-to-one mapping between feature-traces and features. However, the visualization of the feature space revealed that there is no a one-to-one mapping between features and traces. We need to consider a feature, not an execution trace, as the smallest unit of behavior: traces such as, for example the set of copychild, addfolderchild, addpagechild and removechild seem to implement variations of the same feature, while traces such as for example editpage seem to implement multiple features in sequence. It is an open question, how to best model this graph of relations between and among features and traces. It is by performing feature analysis in the first place that we discover such relationships. Thus, obtaining the best feature definition for an analysis is based on the analysis itself. This clearly suggests an iterative approach to feature definition based on the findings of feature analysis. We plan to investigate this more in the future.

6. Related Work

The basis of our work is directly related to the field of dynamic analysis [1], in particular in the context of reverse engineering[14], visualization of runtime information [2] and trace summarization techniques [10, 9].

Many approaches to dynamic analysis focus on the problem of tackling the large volume of data. Many compression and summarization approaches have been proposed to support the extraction of high level views to support system comprehension [8, 9, 14]. This research is directly related to our work.

In the context of reverse engineering and system comprehension, Zaidman and Demeyer [14] propose an

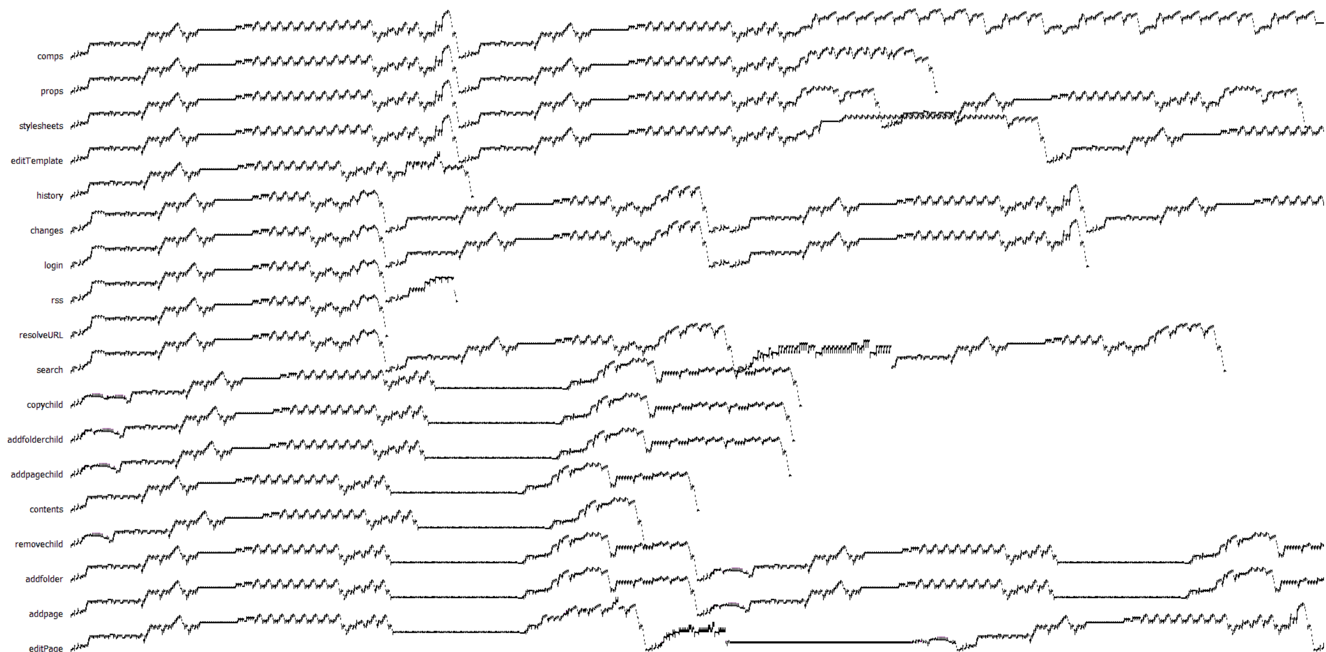

Figure 4. The complete feature space of the SmallWiki case study on one page.

approach to managing trace volume through a heuristical clustering process based on event execution frequency. They use a heuristic that divides a trace into recurring event clusters. They argue that these recurring event clusters represent interesting starting points for understanding the dynamic behavior of a system. Their goal is to obtain an architectural insight into a program using dynamic analysis. The context of our work is reverse engineering and system comprehension. We extend this work by exploiting a range of analysis techniques from the domain of signal processing.

The trace summarization techniques such as that proposed by Hamou-Lhadj are directly related to our summarization approach [9]. He describes a trace summarization based on the ideas of text summarization and proposes that the trace summarization take an entire trace as input and return a summary of the main executed events as output. Summarization is based on selection and generalization techniques of text summarization.

A primary contribution of our approach is the ability to represent entire traces on one screen. Other researchers have addressed this. Jerding *et al.* propose an approach to visualizing execution traces as Information Murals [11]. They define a *Execution Mural* as a graphical depiction an entire execution trace of the messages sent during a program's execution. These murals provide a global overview of the behavior, They also define a *Pattern Mural* which visually represents a summary of a trace in terms of recurring execution patterns. Both views are interdependent. Our signal views have the advantage that they reflect the time and sequence of dynamic data.

Pattern detection in dynamic behavior is a research question that has been addressed by many researchers. Hamou-Lhadj and Lethbridge describe an algorithm that extracts patterns in execution traces. They present a set of matching criteria that the use to decide when two patterns are considered equivalent [10]. De Pauw *et al.* apply pattern extraction algorithms to detect recurring exection behavior in traces [3]. Recent work of Nagkpurkar and Krintz [12] describe a technique whereby they characterize the behavior of programs as *phases*. These phases represent repeating patterns in the trace. They decompose a program into fixed-sized intervals of events and combine these according to how similar the intervals are.

7. Conclusions and Future Work

In this paper we drew an analogy between dynamic analysis and signal processing and we described how to transform traces into time series. We visualized traces as time plots, and presented a summarization technique that reduces the length of the trace signal by 50% to 90%, while preserving information relevant to our research goals.

We implemented our signal visualization in DynaMoose, a dynamic analysis tool integrated with the Moose reengineering framework [4]. Using time plot vi-

sualization and Monotone Subsequence Summarization we have been able to fit the complete visualization of 18 traces containing over 200'000 events on one single screen. Furthermore, due to the capacity of the human visual system in detecting pattern, this visualization made possible to discern patterns both within and between the traces that would other have been obfuscated by the vast amount of raw data. We plan to further investigate on these promising results, using pattern matching and data mining algorithms.

Analysis of our SmallWiki case study reveals patterns in traces. We recognize that further research in the form of more case studies is required before we can conclude that all phenomena observed in this paper are common to any feature space. However we assume that at least some of the observed patterns are generalizable on most case studies. In our case study, we observed that all traces share a common introduction sequence, which however shows slight variations in some traces. Also we observed that there are large sequences shared by multiple traces, and that there are very few patterns which occur in one sole trace only.

This leads us to our initial question whether our definition of features is flawed, and in fact, a many-to-many relationship between traces and features is much more probable than a simple one-to-one relationship. For example in our case study, traces such as copychild, addfolderchild, addpagechild and removechild seem to implement variations of the same feature, while traces such as for example editpage seem to implement multiple features in sequence. It is an open question, how to best model this graph of relations between and among features and traces. We plan to investigate this more in the future.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006).

References

[1] T. Ball. The concept of dynamic analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[2] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93*, pages 326–337, Oct. 1993.

[3] W. De Pauw, S. Krasikov, and J. Morar. Execution patterns for visualizing web services. In *Proceedings ACM International Conference on Software Visualization (SoftVis 2006)*, New York NY, Sept. 2006. ACM Press.

[4] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[5] S. Ducasse, L. Renggli, and R. Wuyts. Smallwiki—a meta-described collaborative content management system. In *International Symposium on Wikis (WikiSym'05)*, pages 75–82, New York, NY, USA, 2005. ACM Computer Society.

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[7] M. Fowler. *UML Distilled*. Addison Wesley, 2003.

[8] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society Press.

[9] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2005.

[10] A. Hamou-Lhadj and T. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proceedings of 1st International Workshop on Dynamic Analysis (WODA)*, May 2003.

[11] D. Jerding, J. Stasko, and T. Ball. Visualizing message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology, May 1996.

[12] P. Nagpurkar and C. Krintz. Phase-based visualization and analysis of java programs. In *Elsevier Science of Computer Programming, Special issue on Princples of programming in Java*, volume 59,Number 1-2, pages 131–164, Jan. 2006.

[13] L. Renggli. SmallWiki: Collaborative content management. Informatikprojekt, University of Bern, 2003. http://smallwiki.unibe.ch/smallwiki.

[14] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 329–338, Mar. 2004.