Exploiting the Analogy Between Traces and Signal Processing

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

Abstract

The main challenge of dynamic analysis is the huge volume of data, making it difficult to extract high level views. Most techniques developed so far adopt a fine-grained approach to address this issue. In this paper we introduce a novel approach representing entire traces as signals in time. Drawing this analogy between dynamic analysis and signal processing, we are able to access a rich toolkit of well-established and ready-to-use analysis techniques. As an application of this analogy, we show how to fit a visualization of the complete feature space of a system on one page only: our approach visualizes feature traces as time plots, summarizes the trace signals and uses Dymanic Time Warping to group them by similar features. We apply the approach on a case study, and discuss both common and unique patterns as observed on the visualization.

Keywords: reverse engineering, dynamic analysis, trace summarization, dynamic time warping, features, feature-traces, 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.

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 [6]. Dynamic analysis together with program visualization may be used in debugging, evaluating and improving program performance and in understanding program behavior. The context of our dynamic analysis is feature-centric reverse engineering (i.e. we exercise a systems' features on an instrumented software system and capture traces of their runtime behavior).

In this paper we propose a novel approach to tackle the problems of dynamic analysis by treating execution traces as signals in time. The domain of time series offers a rich toolkit of well-established and ready-to-use techniques, which become applicable on traces as we draw such an analogy. We show evidence of the usefulness of applying signal processing analysis techniques to traces, by providing a visualization that fits up to two dozen feature traces on one single screen at once. We show how this supports the reverse engineer to interpret and reason about the dynamic information. In particular, we address the following reverse engineering questions:

- *How does a visualization with dozens of feature traces fit on one screen?*
- *Which groups of features behave similarly at runtime, that is which traces are similar?*
- *Which execution patterns are common to all features, which are unique to a single feature?*
- *How does the architecture of a system (for example layers) map to features traces?*

In this paper, we use SmallWiki [4] as an example case study. The same SmallWiki case study has been analyzed in another work by Greevy *et al.* with a metrics-based approach [8]. For this paper we traced 7 additional features, such that now the feature space includes a total of 18 feature traces.

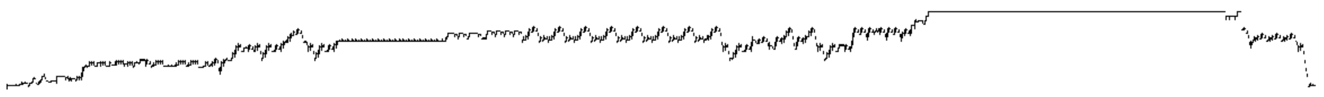

Figure 1. An execution trace as signal, showing on the y-axis the nesting level and along the x-axis the sequence of execution; summarized with Monotone Subsequence Summarization using gap size 0.

Structure of the paper. In Section 2 we draw the analogy between traces and times series and in Section 3 we visualize trace signals as time plots. In Section 4 we presents a trace summarization based on splitting the time series into monotone subsequences. In Section 5 we employ Dymanic Time Warping to measure the similarity between traces. In Section 6 we apply feature characterization and concept location on time plots. Section 7 dives into the details of a case study, while Section 8 discusses pros and cons of the analogy. In Section 9 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 10.

2. Traces are Signals in 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 [5]. 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.

We provide a formal definition a *trace* as a chronological sequence T of one or more execution events, such that 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 based on their depth in the call hierarchy, we define the *nesting level* recursivley as

$$\text{Level}(e_m) = \text{Level}(e_n) + 1$$

for all $e_m \in \text{Children}(e_n)$ and $\text{Level}(e_1) = 1$. Now we can order the execution events in two ways, either by time or by nesting level.

3. Visualization as Time plot

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, see 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.

Typically an execution event includes information about execution time. However, in this paper we omit execution times and retain the order of events only. 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 both the summarization technique we present in Section 4 and the Dymanic Time Warping retain their results.

4. How to Summarize Traces

As the size of a trace may range from some ten thousand to millions of method calls, we have to summarize the time plot somehow in order to fit it onto one screen. In this section 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 it 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 execu-

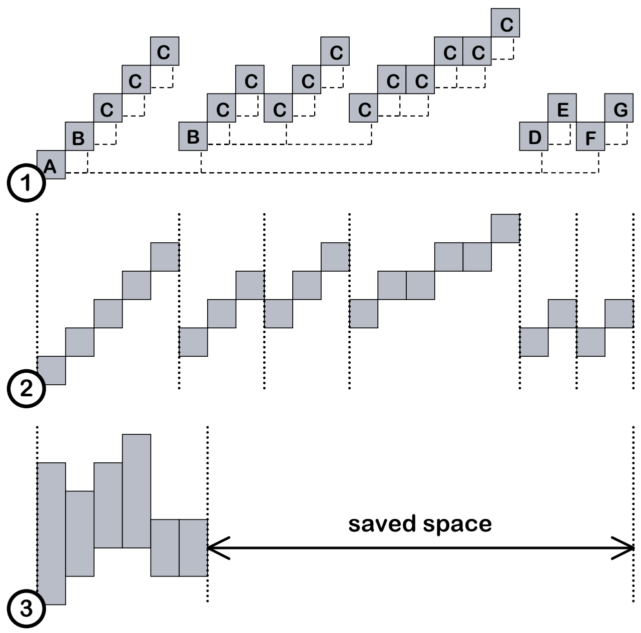

Figure 2. From top to bottom: 1) The outline of a trace, rotated by one quadrant such that time is on the X-axis. 2) We remove method names and retain the nesting level only, each discontinuity is marked with a dotted line. 3) Each monotone subsequence is compressed into one method-call-chain, saving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

tion continues with the latest sibling of the previous events.

The first two cases are monotonnally increasing subsequences, whereas the latter 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 one event of the summarization. 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

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

into pieces $c_1 \dots c_m$ and these pieces become the events of the summarized trace. Further, 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, that is even a trace with ten thousand events fits on one screen.

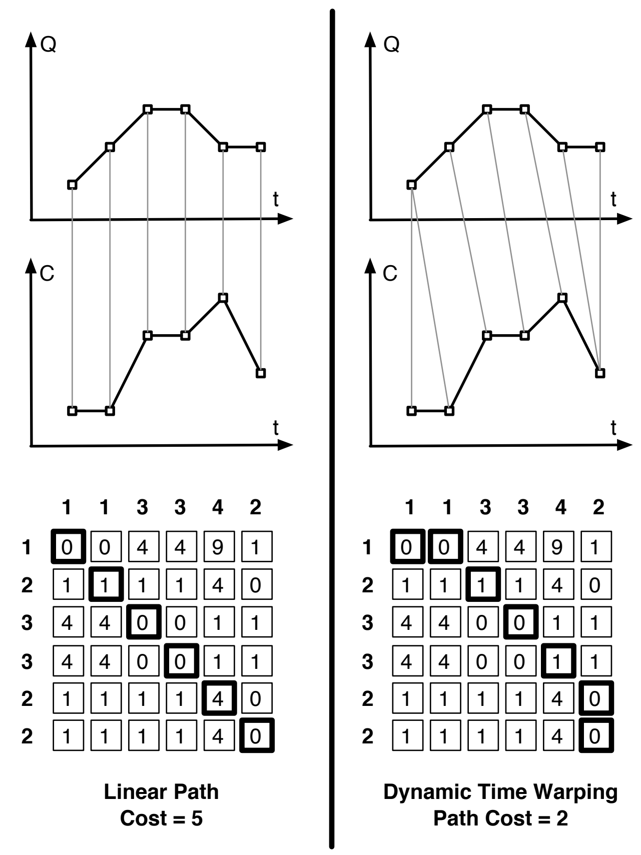

Figure 3. Example of exact matching versus Dymanic Time Warping, the comparisons are highlighted in bold. The exact matching distance is 5 while the DTW distance is 2.

5. How to Compare Traces

Our motivation to compare traces is to identify similar features. We assume that if two traces show a high degree of similarity, then their corresponding features are similar as well. The technique we use to measure the similarity between traces is known as Dymanic Time Warping.

Dymanic Time Warping has been applied in many fields such as manufacturing [7], medicine [2], and originally in automatic speech recognition [22], where it is often used in the context of hidden Markov models to cope with different speaking speeds. In the case of execution traces, it is often the case that the time plots of two traces have approximately the same shape, but due to additional branches or loops, do not line up on the X-axis. In order to measure similarity between two traces, we must warp the time axis of one (or both) of their corresponding time plot sequences to achieve a better alignment.

As illustrated in Figure 3, Dymanic Time Warping compares each data point in one signal with each data point in the other signal and uses dynamic programming to find the minimal distance between the two signals. This allows one signal to fall back behind the other signal and catch up later on. In other words the signals are *warped* non-linearly to match each other.

To apply Dymanic Time Warping, we have to provide a distance metric on execution events. As a first experiment, we used the difference between the nesting level as distance metric—which failed. It is much wiser to use the nesting level's *derivation* as base of the distance metric. A small gedanken experiment illustrates why: consider for example that we add some additional nested calls in the beginning of a trace signal: the nesting level would increase for all events by a fixed number, and therewith increase the distance between two otherwise similar signals, whereas the derivation of the level is preserved.

Therefore, let us define the distance distance between e_n and e_m as:

$$\mathrm{dist}(e_n, e_m) = \big(\Delta e_n - \Delta e_m\big)^2 + \mathrm{p}(e_n, e_m)$$

with $\Delta e_n = \mathrm{Level}(e_n) - \mathrm{Level}(e_{n+1})$ as the derivation of the nesting level. Plus a penalty if e_n and e_m are not sharing some common trait, such as not calling the same method, not being performed on the same class or within the same package.

For the case study used in this paper, we applied *Semantic Clustering* to locate concepts [16] and use the semantic similarity (which is a value between 0 and 1) as the base of the penalty:

$$\mathrm{p}(e_n, e_m) = p_0 * \big(1 - (e_n \sim e_m)\big)$$

with $e_n \sim e_m$ as the semantic similarity between the receiver class of the compared events, and p_0 as a parameterizable threshold. A choice of $p_0 = 100$ has been shown to yield good results.

6. Mining Feature Traces

A key issue when analyzing feature traces, is to detect subsequences in the trace signal that are relevant, specific or even unique to a feature [8, 5, 27]. We combine the presented signal processing techniques to support visual data mining in the complete feature space of an application:

- First we take the time plots of all feature traces, and summarize them with a gap size that is large enough to make all traces fit onto one screen only.
- Then we arrange them, one below the other, such that the most similar traces are placed near each other. We use Dymanic Time Warping to compute the similarity, and dendrogram seriation to determine such an ordering.
- Next, we decorate the time plots with information such as feature characterization [8] or concept location [16].
- And finally, we search for both recurring and unique patterns in the traces.

Figure 7 shows the complete feature space of the SmallWiki case study, it is discussed in Section 7.

6.1. Decorating Time Plots with Color

A plain time plot displays the nesting level only, other information about the called method and the classes of sender and receiver is missing. Therefore, we decorate the time plots with colors to show additional information. We use two kind of decorations: either coloring the time plot itself, or highlighting selected subsequences with color marks.

Coloring the time plot on the one hand is best suited to illustrate a partition of the trace signal into different phases, each phase of the signal is colored with a different color. Highlighting selected subsequences on the other hand, is best suited to illustrate properties that are focus to hot-spots, and it is possible to display quantitative information by varying the diameter of the circles used as color marks.

Figure 4 illustrates the different decorations as used in the case study, these are from top to bottom:

1. The plain signal, summarized with *Monotone Subsequence Summarization* (see Section 4).
2. The signal is decorated with colors, for each dot the colors shows the concept of its corresponding event. The *Semantic Clustering* detected four con-

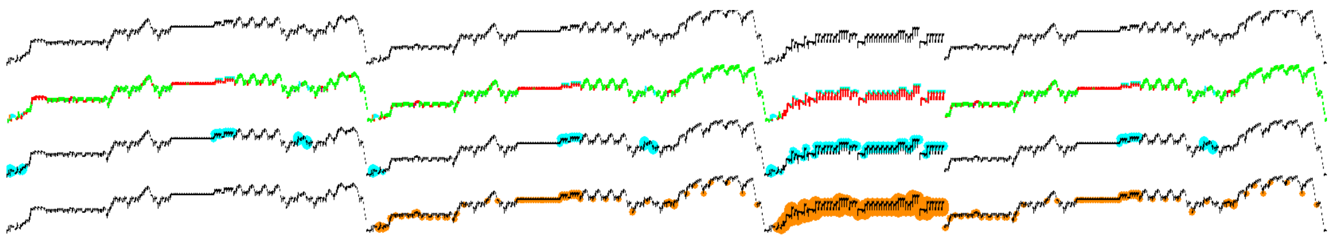

Figure 4. The same trace signal with four decorations: a) plain signal summarized with gap size 1, b) colored by concept, c) highlighting the usage of concept cyan and d) highlighting the usage of single feature classes.

cepts (represented by four distinct colors)[1] and each event is colored with the concept implemented by its receiver class.

3. As the concepts Red and Green dominate the feature space, we decided to highlight all locations of concept Cyan with a decoration of its own.

4. And finally, to support the detection of unique feature behavior, we highlight all receiver classes that are unique to one feature of the set of features we traced. In other words we highlight all classes with feature characterization "single feature class" [8].

7. Case study: SmallWiki

For our experiments with the technqiues described in the previous sections, we chose *SmallWiki* [24], 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 assume that each link or button on a page triggers a distinct feature of the application. 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 4, and we represent each feature trace visually as a time plot on one screen.

1 Each of these concepts is a layer of the application, and we use the terms concept and layer synonymously in this paper. More in-depth information on detecting semantic concepts and relating them to layers is out of the scope of this paper, we refer the reader to a previous work [16].

7.1. Detecting Concepts in SmallWiki

By applying semantic analysis approach [16] to SmallWiki, we detect 4 distinct concepts, which we represent with four colors (ordered by the number of the classes in the concept):

The Red Concept consists of 42 classes. This is the largest concept. It groups classes that represent functionality that corresponds to classes that are characterized as *single-feature* by our feature characterization measurement[8]. For example this concept groups the class `RSSChangeFeed`, which is only present in the `rss` and the class `PageHistory` which is only present in the `ShowHistory`.

The Blue Concept consists of 27 classes. The classes of this concept represent the elements of the SmallWiki pages such as the class that defines if a page can be modified (`EditableProperty`) or models the page title (`TitleProperty`).

The Green Concept consists of 6 classes. This concept groups classes such as `HtmlWriteStream` and `Response`. These classes are responsible for providing the general http dialog functionality of the web application.

The Cyan Concept consists of 4 classes. This concept groups the classes `User`, `BasicRole`, `AdminRole` and `Permission`. The developers confirm are concerned with modeling the user of SmallWiki and authentication concerns.

7.2. 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 and Figure 7, 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 at least the first three observations on layers hold true for any trace signal.

The layers appear as phases. The hypothesis that layers partition the signal by nesting levels does not hold, rather we observe that the layers partition the signal into consecutive subsequences. Thus the nesting level of an event does not correlate with the nesting of the called method within the systems static architecture.

Each of the layers has its own signal. We observe that each layer has in its phases a distinct signal of its own. For example on Figure 7, the signal of layer Green has a large amplitude and oscillates with low frequencies in long loops of more than hundred method calls. Whereas the signal of layer Red and Blue has a small amplitude and oscillates with high frequency in short loops of less then a dozen method calls (in fact the frequency is that high, that in a summarized trace, it shows up as a steady signal).

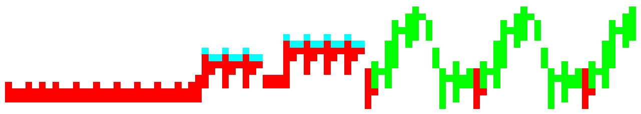

Figure 5. Layers partition call-chains by nesting level: the method-call-chains in the middle starts within layer Red and end in layer Cyan.

Call-chains are layered by level. Contrary to the trace as a whole, the layering indeed partitions the call-chains by nesting levels. Consider for example Figure 5, each of these call-chains starts within layer Red and end in layer Cyan. The same is true for most call-chains at the end of a longer red phase, within the whole feature space as the right bottom subfigure in Figure 7 shows.

All features share a common introduction. We observe that all features start with the same introduction. This introduction corresponds to the time plot of the resolveURL feature. This makes sense due to the nature of SmallWiki as is 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 summarized by removing or *factoring out* the common introduction part of the trace.

Shared parts may include variations. Even thought the same introduction is shared by all trace signals, our analysis reveals variation points. On Figure 7 the introductions of the features copychild, addfolderchild, addpagechild and removechild for example include a variation point, they contain a sequence of calls from layer Red to layer Blue which are not present in the other traces. Whereas the introductions of history and editpage, even though they look like all other introductions, make use of a "single feature class": most probably these are dedicated subclasses of the common introduction implementation.

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 that feature (that is characterized as *single-feature*). This is due to the generic nature of the SmallWiki application. Moreover, as the features we exercised are user-triggerable actions, they all involve exercising common functionality to handle the http request-response dialog of a web application.

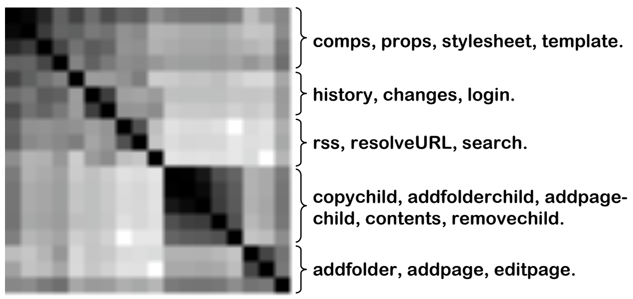

Figure 6. Correlation matrix of all feature traces, and how they fall apart into clusters of similar traces (ordering is the same as on Figure 7).

Some features are very similar. Our time series representation of feature traces as shown in Figure 7 reveals which features are closely related (that is they reveal common patterns of behavior). The features copychild, addfolderchild, addpagechild and removechild are all invoked from the same page in SmallWiki. We verify our findings with the developers and they confirm that these features are actually exercising the same code. The variations of functionality is determined by parameters passed in the methods.

The features comps, props, stylesheets and edittemplate are clustered into one group on Figure 6, these four features also reveal similar time plots on Figure 7. Again the developers confirm our findings, as these features are concerned with *look-and-feel* aspects of the system.

Not all features are equally similar. Figure 7 shows the features addFolder, addPage are grouped as similar features. The feature *editPage* appears to be similar to the previous two features but then exhibits a strong variation. The similar parts of these features indicate that the features are conceptually related.

8. Discussion

IIn 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, it is not per se guaranteed that 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.

However, our experiments showed evidence that the nesting level is indeed a good discriminator. Even applying the Dymanic Time Warping similarity with a null penalty of $\mathrm{p}(e_n, e_m) = 0$ yielded a reasonably good clustering of feature traces. Furthermore, it was exactly for these reasons that we included the penalty in the formula of the distance metric as a means to include additional properties in the computation of the similarity.

In the same way, the decoration of the time plots enrich the signal with additional properties. It is in fact possible to decorate the time plot with any property the execution events may have: if the property is mutually exclusive, coloring the trace itself is the decoration of choice, if the property quantified and is concentrated in hot-spots, the color marks are the decoration of choice.

On the meaning of amplitude and frequency. Our experiments disproved the assumption, that the nesting level if an execution event correlates with the layer of the called method. So the question remains as to what the meaning of the nesting level is. We will discuss this in terms of amplitude and frequency, both taken from the domain of signal processing. A high amplitude may indicate heavy use of delegation, recursion or any other source of nested method calls. A low amplitude, on the other hand, is a good indicator for locality of an algorithm, as only very few methods are used repeatedly. Similarly when considering the frequency: a high frequency is an indicator for a tight loop structure that locally performs one task, while a low frequency might span over hundreds of execution events and reveals high level repetitions such as subsequently executed, but similar tasks.

On the one-to-one mapping between features and traces. 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 feature in a row. It is an open question, how to best model this rhizome of relations between and among features and traces. We plan to investigate further studies on this issue.

9. 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[28], visualization of runtime information [20] and trace summarization techniques [10, 9]. Furthermore we apply techniques from the research domain of signal processing, in particular Dymanic Time Warping [13, 12].

Keogh *et al.* have applied Dymanic Time Warping to Data Mining to detect similar patterns in large data sets [13, 12]. We exploit this technique in the context of reverse engineering of dynamic feature behavior to detect similarities between feature traces. This technique is ideally suited to handling large amounts of data.

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, 28]. This research is directly related to our work.

In the context of reverse engineering and system comprehension, Zaidman and Demeyer [28] propose an 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

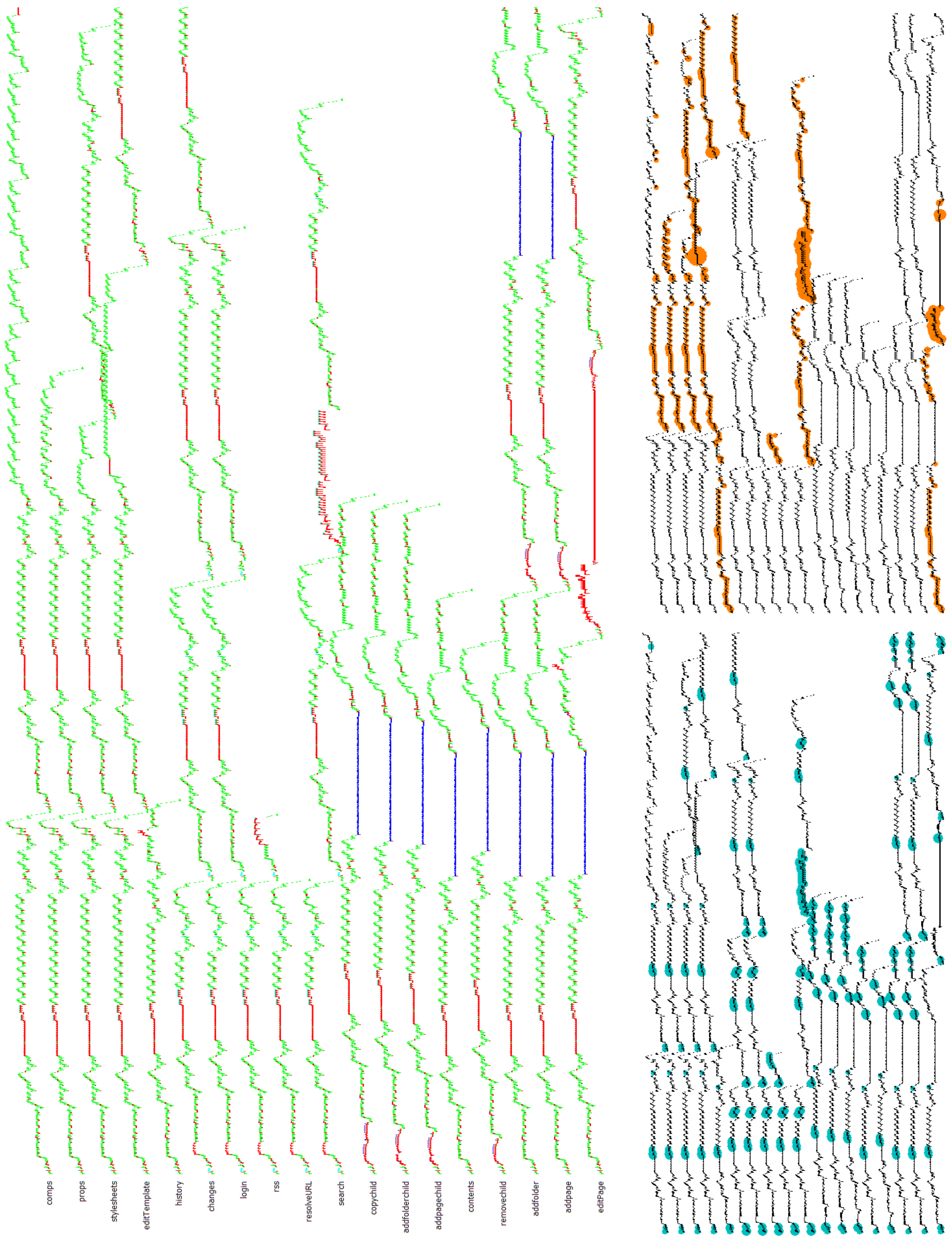

Figure 7. The complete feature space of the SmallWiki case study: (left) colored by concept, (bottom right) highlighting the usage of concept cyan and (top right) highlighting the usage of single feature classes.

work is reverse engineering and system comprehension. We extend this work by exploiting a range of analysis techniques from the domain of signal processing.

Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids [21, 25]. Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids [21, 18].

Substantial research has been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [17], Jinsight and its ancestors [20], and Graphtrace [14]. Vion and Drury [26] use 3D to represent the runtime of objects in distributed and concurrent systems. De Pauw *et al.* present two visualization techniques. In their tool Jinsight, they focused on interaction diagrams [20]. Thus all interactions between objects are visualized. The aim of Jinsight is to support user to understand, tune and debug a program. It is useful for performance analysis, memory leak diagnostics and program comprehension. Reiss [23] developed *Jive* to visualize the runtime activity of Java programs. The focus of this tool was to visually represent runtime activity in real time. The goal of this work is to support software development activities such as debugging and performance optimizations.

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. He 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. We explore the hypotheses outlined in this work in the context of our SmallWiki case study and reveal that in this case the nesting levels do not correspond to architectural layers.

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. Also this metaphor enables us to to exploit the ready-to-use techniques of signal processing.

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]. Recent work of Nagkpurkar and Krintz [19] 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.

10. Conclusions and Future Work

We proposed to draw an analogy between dynamic analysis and signal processing, to make technique from the field of signal processing available for dynamic analysis. We described how to transform traces into time series, and how to apply time plots and Dymanic Time Warping on execution traces. We showed evidence of the usefulness of our analogy by visualizing up to two dozen of feature traces on one screen only, using these two techniques.

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 relevant information. And we employed Dymanic Time Warping to measure the similarity between trace signals, and arranged them accordingly on the screen. In that way, we successfully managed to visualize the complete information about the feature space of an application within the bounds of one single screen.

We implemented this visualization in a tool on top of the Moose reengineering framework [3], using the Hapax tool [15] to perform the *Semantic Clustering* and the TraceScraper tool [8] for the dynamic analysis.

Considering SmallWiki as a case study, we made a set of observations which answered out initial questions. 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, but we assume that at least those presented below hold true for any feature space.

- Using Dymanic Time Warping to measure the similarity between trace signals, confirmed that related features have similar execution traces.

- Decorating the time plots with colors related to concepts, revealed that architectural layers split traces into phases.
- Decorating the time plots with both rare concepts and the usage of "single feature classes", revealed that relevant information which might help to discriminate feature traces is restricted to limited hot-spots.
- On the other hand, over 90% of the a trace's signal contain generic content, in particular we observed a common setup shared by all feature traces.

We plan to further investigate on these promising results, in particular we plan to automate the detection of relevant hot-spots in the feature space using pattern matching and data mining algorithms.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077). We also thank Tudor Girba for his constructive comments on this work.

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] E. Caiani, A. Porta, G. Turiel, M. Muzzupappa, S. Pieruzzi, F. Grema, C. Malliani, A. Cerutti, and S. Cerutti. Warped-average template technique to track on a cycle-by-cycle basis the cardiac filling phases on left ventricular volume. In *IEEE Computers in Cardiology*, volume 25, 1998.

[3] 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.

[4] 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.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

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

[7] K. Gollmer and C. Posten. Detection of distorted pattern using dynamic time warping algorithm and application for the supervision of bioprocesses. In *On-Line Fault Detection and Supervision in Chemical Process Industries.*, 1995.

[8] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[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. F. Jerding, J. T. 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] E. Keogh. Exact indexing of dynamic time warping. In *Proceedings 28th International Conference on Very Large Databases, Hong Kong*, pages 406–417, Dec. 2002.

[13] E. Keogh and M. Pazzani. Scaling up dynamic time warping to massive datasets. In *Proceedings 3rd European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 1–11, 1999.

[14] M. F. Kleyn and P. C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, Nov. 1988.

[15] A. Kuhn. Hapax – enriching reverse engineering with semantic clustering, Nov. 2005.

[16] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2005)*, pages 113–122, Los Alamitos CA, Nov. 2005. IEEE Computer Society Press.

[17] D. B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 342–357. ACM Press, 1995.

[18] J. I. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Proceedings of the 1st Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*, pages 32–40. IEEE, June 2002.

[19] 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.

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

[21] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[22] L. Rabiner, A. Rosenberg, and S. Levinson. Considerations in dynamic time warping algorithms for discrete word recognition. In *IEEE Transactions. Acoustics, Speech and Signal Processing*, pages 572–582. IEEE, 1978.

[23] S. P. Reiss, editor. *Visualizing Java in Action*, May 2003.

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

[25] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[26] J.-Y. Vion-Dury and M. Santana. Virtual images: Interactive visualization of distributed object-oriented systems. In A. Press, editor, *Proceedings of OOPSLA 1994*, pages 65–84, 1994.

[27] W. E. Wong, S. S. Gokhale, and J. R. Horgan. Quantifying the closeness between program components and features. *J. Syst. Softw.*, 54(2):87–98, 2000.

[28] 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.