Using Compression Algorithms to Support the Comprehension of Program Traces

Neil Walkinshaw
Department of Computer Science
The University of Sheffield
Sheffield, UK
nw@dcs.shef.ac.uk

Sheeva Afshan
Department of Computer Science
The University of Sheffield
Sheffield, UK
s.afshan@dcs.shef.ac.uk

Phil McMinn
Department of Computer Science
The University of Sheffield
Sheffield, UK
p.mcminn@dcs.shef.ac.uk

ABSTRACT

Several software maintenance tasks such as debugging, phase-identification, or simply the high-level exploration of system functionality, rely on the extensive analysis of program traces. These usually require the developer to manually discern any repeated patterns that may be of interest from some visual representation of the trace. This can be both time-consuming and inaccurate; there is always the danger that visually similar trace-patterns actually represent distinct program behaviours. This paper presents an automated phase-identification technique. It is founded on the observation that the challenge of identifying repeated patterns in a trace is analogous to the challenge faced by data-compression algorithms. This applies an established data compression algorithm to identify repeated phases in traces. The SEQUITUR compression algorithm not only compresses data, but organises the repeated patterns into a *hierarchy*, which is especially useful from a comprehension standpoint, because it enables the analysis of a trace at at varying levels of abstraction.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*tracing*

General Terms

Algorithms, Documentation

1. INTRODUCTION

Many questions about program behaviour can only be answered by observing the program as it executes. A range of dynamic analysis techniques have been developed that are based on the use of program traces (recorded program executions), which can incorporate any run-time information such as variable values, as well as the sequence of function calls or events during the execution. Powerful tracing frameworks are increasingly becoming part of the routine development process (c.f. the Eclipse TPTP framework, or the extensive tracing frameworks that are built into emerging languages such as Erlang).

One of the main comprehension tasks [13] is the task of identifying repeated *phases* of behaviour – repeated patterns of trace elements such as method calls – that occur at runtime. Such phases indicate cohesive units of functionality that can be used as a basis for investigating the broader behaviour of the program. The task of identifying repeated phases becomes particularly challenging when the trace is large, and the patterns are complex. Identifying whether one part of a trace is similar to another part, and working out whether and how it relates to other phases is virtually infeasible if the trace is merely presented to the developer in a textual format.

Several visualisation tools have been developed that are intended to address this problem (amongst other comprehension tasks). Visual abstractions such as message sequence charts (and variants) [10, 3], signals [7, 12], and similarity matrices [12, 2] have all been used to identify potential phases in traces. The value of visualisation techniques lies in their ability to summarise large volumes of information, providing a succinct visual overview. However, they are ultimately dependent on the expertise of the person using them, and can be open to ambiguity. Two phases of a trace may appear visually similar, but contain significant functional differences. Furthermore, it is up to the developer to identify any relationships between low-level program phases and macroscopic phases that aggregate them.

The challenge of identifying repeated patterns of behaviour within a sequence is not unique to trace analysis. The work in this paper is based on the observation that the notion of phases also forms the basis for data compression algorithms. Compression algorithms are designed specifically to highlight repetitions in the data, so that these can be collapsed into a compressed form. Of course, compression algorithms are routinely used to compress program traces for storage. However, to the best of the authors' knowledge there has been no work on exploiting the mechanisms of compression algorithms for the sake of highlighting repeated phases and understanding program behaviour. The SEQUITUR algorithm by Nevill-Manning and Whitten [9] is particularly appealing in this respect. Whereas conventional dictionary compression algorithms store repeated patterns

of behaviour in a lookup-table, the SEQUITUR algorithm stores repeated patterns as a hierarchy. Thus, it not only identifies the repeated patterns of behaviour, but also identifies how they are related to each other.

The contribution of this paper is to show how the SEQUITUR algorithm can be used to understand program traces in terms of their repeated phases of behaviour. It is envisaged that the resulting approach will present a useful basis for understanding program behaviour. An openly-available framework is presented that (1) recodes traces into a suitable format for the SEQUITUR algorithm and (2) provides a visual representation of the resulting phase hierarchy. The technique is assessed on a large trace from an openly available systems (JHotDraw). The trace has previously formed the basis for the evaluation of the ExtraVis trace-visualisation system by Cornelissen *et al.* [3].

The rest of the paper is organized as follows. Section 2 describes the necessary background knowledge. Section 3 shows how phase hierarchies can be extracted from phases with the SEQUITUR algorithm. Section 4 presents a case study that shows how the approach is applied to a JHotDraw trace. Section 5 describes related work, and section 6 describes our conclusions and future work.

2. BACKGROUND

2.1 Context-Free Grammars

The SEQUITUR algorithm, which is introduced in the next section, produces a context-free grammar (CFG). A CFG is formally defined by four components $G = (\Sigma, N, S, P)$: Σ is a finite set of **terminals** - the set of symbols that belong to the underlying language. N is a finite set of **non-terminals**, where each non-terminal represents a set of sequences of terminals. S is a single non-terminal that represents the starting point for the language. P is a set of **production rules** that map a non-terminal to a string of zero or more terminals and non-terminals.

As a small example, we could have a grammar where $\Sigma = \{a, b, c\}$, $N = \{A, B\}$, $S = A$ and P is the set of rules $A \rightarrow abB$, $B \rightarrow A$, and $B \rightarrow c$. This represents the language of all sequences that contain at least one sequence $< a, b >$ and terminate with a c.

When a sequence is generated from a grammar, it can be interpreted in terms of a *parse-tree*. The root of the tree is S, the branch nodes are non-terminals in N, branches are defined by P, and the leaves from left to right are the terminals that constitute the sequence. This provides a hierarchical overview of the rules in the grammar that were used to construct the sequence.

2.2 The SEQUITUR Compression Algorithm

The SEQUITUR algorithm was developed by Nevill-Manning and Witten [9] specifically for the compression of strings (sequences of discrete symbols). Its strength is the fact that, as an artefact of the compression process, it produces an explicit hierarchical structure of how repeated patterns of elements in the sequence are related to each other. The basic idea is that a sequence such as: *"coding compiling compressing and comprehending"* contains a lot of repeated subsequences. An effective compression algorithm will reduce this redundancy as much as possible. To provide an intuition of the SEQUITUR algorithm, a CFG that it produces is shown in table 1. The grammar exactly produces the phrase. The initial rule (0) starts off with a reference to rule 1, which produces "co", this is followed by a "d", followed by a reference to rule 2, which contains a reference to rule 6, which produces "ing", etc. The compression occurs because a sequence of symbols that occurs multiple times (such as "co" in rule 1) can be replaced by simple references to the rule that represents that sequence. The hierarchical structure of the CFG is apparent; Rule 0 is the root of the tree, and the terminals are the leaves.

0 → 1 d 2 i l 2 3 s s 4 a 5 _ 1 m p 3 h e 5 6
1 → c o
2 → 4 1 m p
3 → r e
4 → 6 _
5 → n d
6 → i n g

Table 1: Rules produced for phrase "coding compiling compressing and comprehending"

The algorithm is founded on two constraints: (1) no pair of adjacent symbols can appear more than once in the grammar, and (2) every rule has to be used more than once. The input sequence is processed one symbol at a time. If a pair of symbols $< x, y >$ is observed that has appeared previously (violating constraint 1), a new rule $R \rightarrow < x, y >$ is generated and the pair are replaced by that rule. However, if there is already an existing rule $S \rightarrow < x, y >$, the pair are replaced by S instead. If the replacement with S in turn produces a repeating pair, then the rule generation / replacement process is repeated until constraint 1 is not violated. This forms a hierarchy of rules that summarise the input sequence.

So far the rules have only two terminals / non-terminals on their right-hand side. Several rules may have been generated that are only used once, violating property 2 (rules must appear more than once). When this is the case, every occurrence of the unique rule is replaced with the sequence of terminals and non-terminals on its right-hand side. This can in turn lead to larger rules with more complex right-hand sides. It is important to note that, for a given sequence, there are multiple possible valid CFGs that a SEQUITUR implementation could produce, for the work in this paper we adopt the openly available Java implementation by Frank[1]. For a more detailed discussion and illustration of the compression process, the reader is referred to the original paper by Nevill-Manning and Witten [9].

3. PHASE EXTRACTION

The process of extracting phase hierarchies and using these to understand program behaviour is illustrated with respect to a trace from a fictional text editor. The trace records the following sequence: Loading and displaying a text file, editing it by inserting text and copying and pasting, saving the file, and reloading it. We begin by showing how such a

[1]http://sequitur.info/java/

Original trace:
load, displayDir, selectFile, renderChar, renderChar, renderChar, renderChar, insertChar, renderChar, renderChar, insertChar, renderChar, select, copy, paste, renderChar, renderChar, select, copy, paste, renderChar, renderChar, renderChar, saveFile, displayDir, selectFile, load, displayDir, selectFile, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar, renderChar

Mappings:
load=a, displayDir=b, selectFile=c, renderChar=d, insertChar=e, select=f, copy=g, paste=h, saveFile=i

Recoded trace:
abcddddededfghddfghdddibcabcddddddddd

Figure 1: Pre-processing a trace from text editor

Production rules
0 → 1 2 2 2 3 3 4 i 5 1 4 4
1 → a 5 4 4
2 → e d
3 → f g h 4
4 → d d
5 → b c
Uncoded production rules
0 → 1 2 2 2 3 3 4 saveFile 5 1 4 4
1 → load 5 4 4
2 → insertChar renderChar
3 → select copy paste 4
4 → renderChar renderChar
5 → displayDir selectFile

Table 2: Rules produced from text editor trace

trace is processed, to enable the SEQUITUR algorithm to process it. We then show the resulting hierarchy of trace-phases, and show how it forms a suitable basis for established comprehension processes. This example is small and only serves to demonstrate how the process works. The true value of the process will become clear with the case study in the next section.

3.1 Pre-processing a trace

As input, we take a trace in its raw format (e.g. a large XML file of method calls generated by Eclipse TPTP). Before a trace can be processed by the SEQUITUR algorithm it needs to be pre-processed. It would not make sense to simply use the text file produced by the tracing framework, with its long method signatures, XML tags etc. Were this to be done, the vast majority of rules would be the result of repeated patterns in the naming conventions, and tags, as opposed to the patterns of repeated method invocations that constitute program phases.

The pre-processing step consists of reducing each trace element (i.e. method-signature) into a single symbol for the trace. Our pre-processor implementation can take traces from the Eclipse TPTP format, and the format used for the ExtraVis tool [3]. We select characters from the Unicode character set, which means that we can process traces with a virtually unlimited number of different event types or function names. The process is illustrated in figure 1 on a trace from a fictional text editor – the trace is simplified to short method names and the coding is restricted to ASCII characters for the sake of illustration.

3.2 Running the SEQUITUR algorithm

Running the SEQUITUR algorithm on the (recoded) text-editor trace in figure 1 produces the rules shown in table 2, where the initial rule S is rule 0. To make sense of the set of rules, the terminals have to be mapped back to their original method names, and the decoded set of rules is shown in the lower section of the table.

The algorithm produces six rules. Rule 0 represents the complete trace, and the other rules represent phases and sub-phases that occur within the trace. There is only one terminal in the top-level rule (0), because the saveFile method only appears once in the trace. All of the other elements are part of repeated patterns that are encoded by their own rules. From here on we refer to the CFG as the *phase-hierarchy*, where each phase corresponds to a rule.

We adopt a bottom-up approach to reading the hierarchy, and start off with phases that consist only of terminals. Phase 5 denotes the displaying of the contents of a directory and the subsequent selection of a file. Phase 4 denotes the repeated rendering of characters on the screen. Phase 2 denotes the typing of a character, followed by its rendering on the screen. There are also higher-level phases that are composed of both terminals and non-terminals. Phase 3 denotes the process of selecting, copying and pasting, followed by phase 4 (repeatedly rendering characters). Phase 1 denotes the process of loading and displaying a file; the load command, followed by phase 5 (displaying a directory and saving a file), followed by two occurrances of phase 4 (repeatedly rendering characters). Phase 0 is a special phase, because it represents the trace in its entirety, in terms of the rest of the identified phases.

Having identified the hierarchy of phases, there remains the challenge of actually using them as a basis for understanding how the program behaves. Attempting to understand phase hierarchies in their textual form, as shown in table 2 is a tedious process. The mental burden can be alleviated by displaying the hierarchy diagrammatically. Figure 2 displays the hierarchy in table 2 in a diagrammatical format. Each phase is displayed within its own rectangle (the thickness of the rectangle corresponds to the number of times that a rule has been referenced). Terminals are shown in ellipses and non-terminals are shown in rectangles. Each non-terminal is connected to its corresponding phase with a dashed arrow.

When the trace is large and complex, and produces a large phase hierarchy, the developer can resort to beacons (familiar identifiers or method names) to home-in on relevant phases. If the developer has no prior knowledge of the implementation, there is an alternative heuristic. If a phase is large and frequent, it can be assumed that it plays a significant role. Thus, each phase can be annotated with a *weight*, which is computed by multiplying its size (number of termi-

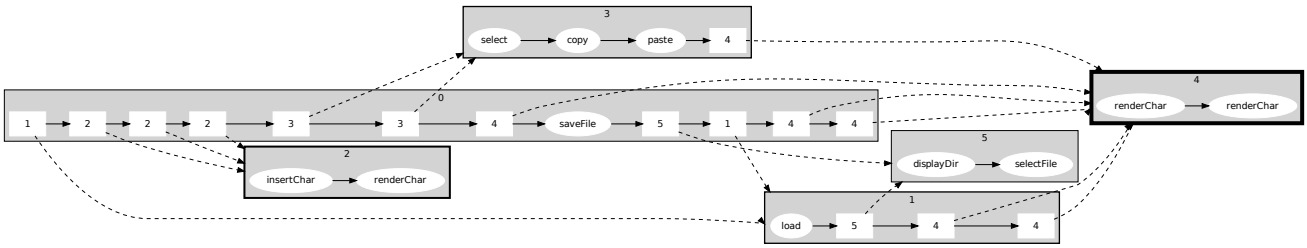

Figure 2: Diagram of phase hierarchy in table 2

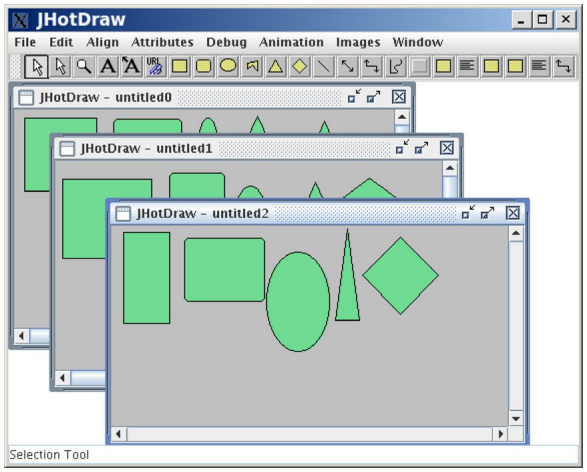

Figure 3: Screenshot from JHotDraw[3]

nals, including terminals belonging to its subphases) by the number of times it is executed. Phases with a higher weight tend to play a significant role, and form a good starting-point for exploring the general behaviour of the trace.

4. CASE STUDY - A JHOTDRAW TRACE

To assess the usefulness of the phase-hierarchies identified by SEQUITUR, it was evaluated on a trace from JHotDraw[2], a Java drawing framework that was developed with a strong emphasis on the use of design-patterns. The trace has been used previously for the evaluation of the ExtraVis tool by Cornelissen *et al.* [3]. For this evaluation, we presume that there is very little prior knowledge of the key features of the system. For the sake of generality, we ignore trace-features that are specific to object-oriented systems; we ignore object instances and simply reduce the trace to a sequence of method calls, where each method is denoted in terms of its signature.

The JHotDraw trace is intended to exercise the main features of JHotDraw. It consists of creating a new drawing canvas, and then adding five figures (a diamond, triangle, rectangle, rounded rectangle and ellipse). This is repeated a further two times, resulting in three similar canvasses, as shown in figure 3. According to Cornelissen *et al.*, the final trace is slightly filtered to remove noise in the form of mouse events. The trace consists of 161,087 method calls. The phase hierarchy was generated by encoding it (as described in section 3.1) and running the SEQUITUR algorithm. The resulting hierarchy consists of 858 rules. In other words, the SEQUITUR algorithm has identified 858 sequences of method calls (phases) that are known to be repeated multiple times at different points in the trace. These phases are related to each other; each phase is explicitly linked to its constituent sub-phases.

The complete hierarchy is very large; there are 2051 links between the 858 phases. Large, high-level phases tend to occur infrequently, but are built from large numbers of small, lower-level phases that can occur very frequently in lots of different contexts. The hierarchy of phases can be explored by selecting a reasonably large, frequent phase, and exploring its constituent lower-level phases. For the sake of illustration we select phase 709, which is shown in figure 4; it is reasonably large (consists of 93 method calls), and frequent (occurs 30 times in the trace). Due to space constraints the labels have been shortened; instead of labelling terminals by their complete method signatures, they just contain the name of the method. Due to the scaling, labels are difficult to read as-is, though these are zoomable in the PDF version of the paper. A full list of grammar-rules, as well as a stand-alone diagram with full-length method signatures are available online[3].

To explore the sub-hierarchy, the authors adopted a process of sequentially tracing through the rules in a depth-first manner. In doing so, the sub-phases are explored in the order in which they are executed. The high-level process of reading through the sub-hierarchy is enumerated below, and this can be followed with the annotations in figure 4, where each step is highlighted. Many of the steps described below require knowledge of the full method signatures (e.g. without knowing the class names, it is impossible to say which figure is being displayed), and these can be found in the online version[4].

(1) To explore phase 709 we start with phase R709.
(2) Tracing the references in a depth-first manner, phase R303 is the first major sub-phase to be fully executed. This sub-phase consists of a total of 22 method calls (including its sub-phases), and is itself executed 376 times throughout the trace.
(3) It begins with a sequence of method calls that set up

[2]http://www.jhotdraw.org

[3]http://www.dcs.shef.ac.uk/~nw/Files/woda2010/

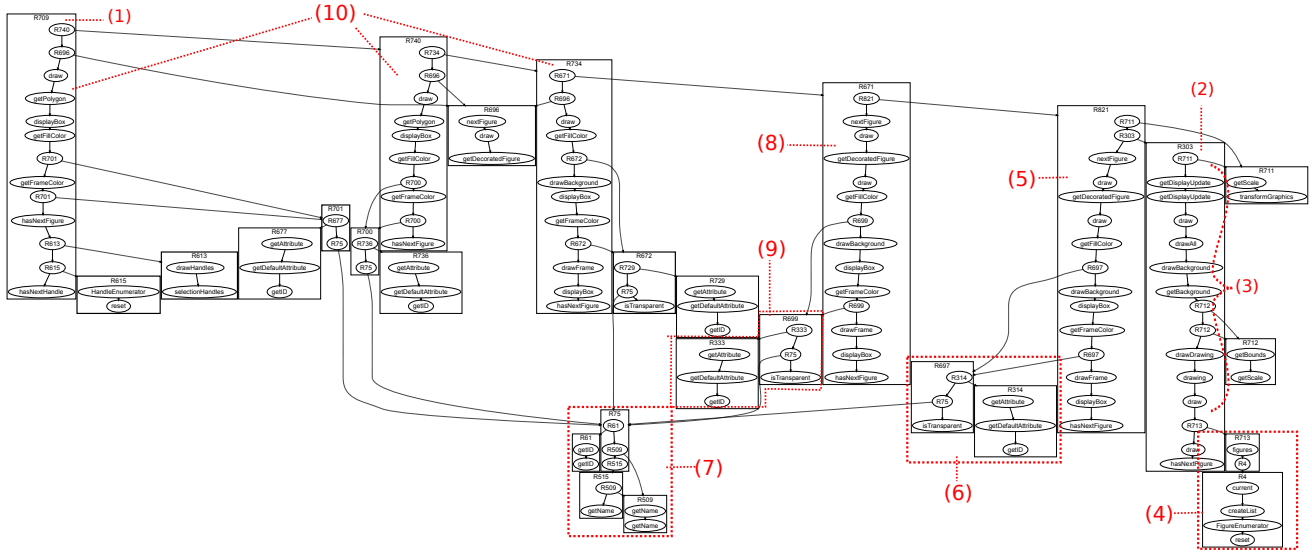

Figure 4: Annotated hierarchy for phase 709.

the canvas (working out the scale, background, etc.).
(4) It ends with a reference to a small phase R713 (executed a total of 414 times throughout the trace), which is responsible for setting up an iterable list of figures in the canvas, and commences to iterate through the figures.
(5) Having completed R303, the next phase to be fully executed is R821. From the class names we can see that this is responsible for rendering a rectangle figure.
(6) This phase contains two references to a subphase R697 which, together with R314 is responsible for processing figure-attributes that are specific to rectangle figures.
(7) However, R697 includes references to phase R75, which is responsible for processing attributes that are generic to all figure types. This subphase appears 2298 times throughout the trace.
(8) Having rendered the rectangle, it is followed by R671, which renders the round-rectangle figure in a similar way.
(9) Again, this contains references to a phase that is responsible for round-rectangle specific attributes, but which includes references to phase R75 to render the generic attributes.
(10) The subsequent phases (R734, R740, and R709) all draw their respective figures (ellipse, triangle and diamond respectively) in a similar fashion, first processing their figure-specific attributes with their own sub-phases, before they all end up referring to phase R75 to process generic figure attributes.

From this phase hierarchy we gain a useful insight into the way the figures are processed. JHotDraw figures are associated with sets of attributes, some of which are shape-specific, and some of which are more general. The phase hierarchy shows that each figure first calls specific methods that process its own shape-specific attributes, but that all figures subsequently call the same sequence of methods to process attributes that are general to all figures (see the cluster at the bottom of the hierarchy). As shown in the above description, the individual sub-phases are not used uniquely within the context of phase 709. Smaller phases such as R75, which is responsible for processing attributes for every figure, can appear thousands of times within the trace. However, within the context of rendering *all* of the figures onto the canvas – as shown in phase 709 – this shows how it is used.

5. RELATED WORK

Trace Visualisation

Currently the most popular approach to identifying phases in traces is to adopt visualisation techniques. Tracing tools such as Eclipse TPTP are equipped with the ability to visualise traces as large sequence diagrams [10]. Sherwood *et al.* [12] and Cornelisson and Moonen [2] independently describe an approach that identifies phases of behaviour with the aid of a matrix-based approach, where similar phases are clustered together as different-sized blocks along the diagonal of a similarity-matrix. Reiss [11] describes a block-based phase visualisation of method invocations, where the height of a block represents the number of calls made, and the width represents the number of allocations made. Finally, Kuhn and Greevy [7] present a technique to visualise a trace as a signal.

Visualisation is ultimately concerned with the challenge of summarising a large amount of information onto a physical window [3]. One inherent danger is that pertinent or interesting information can be lost in the abstraction process. Although a set of phases may look similar to each other, it is difficult to explore the nature of any possible interrelationships between phases from the visualisation alone. As shown above, the information that is produced in the phase-hierarchies is complementary to existing visualisations. It shows which chunks of a trace are identical, and shows how they relate to each other.

Trace Compression and Reduction

Compression algorithms and traces go hand in hand. A conventional trace may be several gigabytes in size and need to

be compressed in some way. So far, compression algorithms have been mainly used solely to reduce the amount of space required to store a trace. To the best of the author's knowledge, the fact that they intrinsically identify phases has not been exploited so far.

Larus [8] used the SEQUITUR algorithm to capture a complete execution profile from an execution trace in terms of its basic blocks and control-flow edges. Ultimately the trace serves merely as a means of recording trace information, and the actual hierarchical information that is generated during compression is ignored. This work uses the SEQUITUR algorithm, but with the expressed aim of exploiting the generated phase-hierarchy to better understand the hidden trace structure. Preliminary work by the authors [1] employed the LZW dictionary compression algorithm to identify phases from traces. However, with this approach the resulting set of phases is 'flat', there are no explicit relationships between high-level phases and lower-level ones. Furthermore, if there is a large phase at the beginning of the trace, it is not discovered. These weaknesses are eliminated by using SEQUITUR.

Feature Identification Approaches

The feature-identification challenge is to answer the question: "Which source code statements are responsible for implementing feature X?". The problem has its roots in the work on software reconnaissance by Wilde et al. [15], who combine traces that are known to execute a feature with those that are known *not* to in order to identify the relevant source code. Eisenbarth et al. [4] have attempted to combine dynamic analysis with static analysis and Formal concept analysis to identify potential features. More recently, Greevy and Ducasse [5], Kothari et al. [6] and Wanatabe *et al.* [14] have all worked on the identification of trace phases to identify features in the source code.

6. CONCLUSIONS AND FUTURE WORK

The process of understanding program traces largely revolves around the identification of repeated patterns of trace elements, and working out how different patterns might be related to each other. Similarly, the process of compressing a stream of data revolves around identifying repeated patterns, so that redundant repetitions can be eliminated where possible. This paper shows that, thanks to this analogy, established data compression techniques can be exploited for the purpose of program comprehension, and has demonstrated this with the SEQUITUR algorithm.

The evaluation has so far been largely qualitative, but shows that the technique is promising. In our future work we will use a larger set of traces from a broader range of systems. We will also identify a set of metrics to form the basis for a more quantitative approach. This will involve the identification of a set of target phases for each trace, and measuring the precision. Our future work will also look at existing visualisation tools to investigate how the hierarchical information can be used to make the visualisations more useful to the developer.

Acknowledgments

Walkinshaw and McMinn are supported by the EPSRC REGI grant EP/F065825/1, McMinn is also supported by the EPSRC Misbehaviour grant EP/G009600/1, and Walkinshaw is also supported by the EPSRC STAMINA grant EP/H002456/1.

7. REFERENCES

[1] S. Afshan, P. McMinn, and N. Walkinshaw. Using dictionary compression algorithms to identify phases in program traces. Technical Report CS-10-01, Department of Computer Science, The University of Sheffield, 2010.

[2] B. Cornelissen and L. Moonen. Visualizing similarities in execution traces. In *Proceedings of the International Workshop on Program Comprehension through Dynamic Analysis (PCODA'07)*, 2007.

[3] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[5] O. Greevy and S. Ducasse. Correlating Features and Code using a Compact Two-sided trace analysis approach. In *In Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, 2005.

[6] J. Kothari, T. Denton, S. Mancoridis, and A. Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, Oct. 2006.

[7] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings of the International Conference on Software Maintenance (ICSM'06)*, pages 320–329. IEEE Computer Society, 2006.

[8] J. Larus. Whole program paths. In *PLDI*, pages 259–269, 1999.

[9] C. Nevill-Manning and I. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40(2/3):103–116, 1997.

[10] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 219–234. USENIX, 1998.

[11] S. Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the Workshop on Dynamic Analysis WODA'05*, St. Louis, USA, 2005.

[12] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[13] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering java software systems. *Softw, Pract. Exper*, 31(4):371–394, 2001.

[14] Y. Wanatabe, T. Ishio, and K. Inoue. Feature-level phase detection for execution trace using object cache. In *Workshop on Dynamic Analysis (WODA'08)*, 2008.

[15] N. Wilde and M. Scully. Software Reconnaissance: Mapping Program Features to Code. *Software Maintenance: Research and Practice*, 7:46–62, 1995.