Using Dictionary Compression Algorithms to Identify Phases in Program Traces

Sheeva Afshan, Phil McMinn, Neil Walkinshaw

December 2, 2009

Abstract

Program execution traces record the sequences of events or functions that are encountered during a program execution. They can provide valuable insights into the run-time behaviour of software systems and form the basis for dynamic analysis techniques. Execution traces of large software systems can be huge, incorporating hundreds of thousands of elements, rendering them difficult to interpret and understand. One recognised problem is the phase-detection problem where the challenge is to identify repeating phases within a trace that may correspond to the execution of particular features within the software system. This paper proposes an abstraction technique that uses the well-known LZW dictionary compression algorithm to systematically identify such phases. The feasibility of this approach is demonstrated with respect to a small case study on a Java program.

1 Introduction

Understanding precisely how a software system behaves is vital for a large proportion of routine software maintenance and development tasks. When the system in question is unfamiliar, the challenge of understanding exactly how and when different parts of the system interact with each other can be daunting, especially when the system is large and complex. It has been estimated that the task of program comprehension can account for 50-60% of the entire effort invested into the development of a software system [1, 2].

Many questions about program behaviour can only be answered by observing the program as it executes. To answer such questions, a range of dynamic analysis techniques have been developed. These 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).

Traces tend to record a vast amount of data for each execution, much of which is irrelevant to the task at hand. The *phase identification* challenge is to identify trace segments that might correspond to well-defined units of functionality. As an example, given a trace of a word-processing program, the developer may only be interested in the phases that are responsible for a particular feature such as displaying a picture, or loading a file. Even if they have been careful to execute these features extensively during the trace collection, actually isolating those parts of the trace that are relevant can be an overwhelming task.

So far the phase identification challenge has been addressed by visualisation; visual abstractions such as message sequence charts (and variants) [3], signals [4, 5], and similarity matrices [5, 6] 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 (e.g. in the word processor trace, a phase that is responsible for loading a file may visually be very similar to one that is responsible for saving a file). Furthermore, such approaches often fail to provide insights into how different phases may be related to each other.

The challenge of phase identification is not unique to trace analysis. The work in this paper is based on the observation that the notion of phases has formed the basis for dictionary compression algorithms, which have been well established for several decades. The main goal of such algorithms is to identify any repeated 'phases' in the data being compressed, and to replace each phase with a simple symbol, which points to an entry in a dictionary of recorded phases.

This paper shows how dictionary compression algorithms can be used to identify phases in program traces. We have chosen the LZW compression algorithm [7] as a basis, but in principle any lossless dictionary compression algorithm is suitable. The algorithm automatically builds a dictionary of observed method call sequences in a trace and compresses the trace according to the entries of the dictionary. The final dictionary can be interpreted as a list of phases, and it is argued that these correspond to abstract units of functionality within the program. The main contributions of this paper are as follows:

- A technique that uses the LZW algorithm to identify trace phases
- An openly available implementation of the technique, along with a processor that can accept traces from the Eclipse TPTP trace generator
- Evaluation with respect to a small case study.

The rest of the paper is organized as follows. Section 2 describes the necessary background knowledge about dynamic analysis and compression algorithms. Section 3 describes our trace abstraction technique. Section 4 evaluates the technique on a small case study. Section 5 describes related work, and section 6 describes our conclusions and discusses future work.

2 Background

2.1 Dynamic Analysis and Program Traces

A program trace is a recording of a program execution. This usually incorporates control events that record the encountered control points in the source code (method entry / exit points, thread start and stop events etc.), along with associated with data states that detail variable assignments, object instance information etc. The specific trace elements vary according to the tracing framework and the programming language of the underlying system.

The contents of a trace are usually determined by the tracing technique. At the simplest level, a trace may consist of a sequence of print-statements inserted into the code at relevant points. Sophisticated tracing frameworks such as Eclipse TPTP allow the user to specify filters that govern which parts of the system are included, as well as the level of detail of a trace (e.g. whether it should include object-specific information). When the system under analysis includes concurrent threads, the trace can be restricted to focus on a specific thread.

The format of a trace depends on the task at hand. In this work we are concerned with the identification of phases (see below), where each phase is a distinctive sequence of elements in the trace. For the sake of the presented technique, it is important that the trace can be recoded into a sequence of characters or symbols, where each symbol corresponds to a unique element in the trace. The traces considered in this paper are simply sequences of method entry points. We have chosen this trace format for the sake of illustrating this technique; depending on the nature of the program under analysis it it is possible to tailor the approach to deal with more complex trace contents.

2.2 Phase Identification

In the context of software execution traces, a phase is a distinctive period of activity. Usually, a phase in a trace corresponds to the execution of some well-defined unit of functionality within the program. For example, with respect to some text editor, the act of loading a text file will produce repeated patterns of IO events and method calls, whereas running the spell-checker will include loops that process each word in the document and compare it to a dictionary entry, producing a different pattern (or phase) of method calls and events.

In the context of this work where the trace is a sequence of method invocations, a phase is a particular distinctive sub-sequence of method invocations. Thus, the challenge of phase identification is to identify suitable sub-sequences that are likely to correspond to units of functionality in the program.

Phase analysis has a wide range of applications. If the developer is attempting to understand the dynamic program behaviour by analysing a trace, the first task is to identify the phase that is particularly relevant to their inquiry. A more specific program analysis problem is the task of "feature analysis" (discussed in the related work section), where the broad aim is to understand a

```
Input: Stream
Uses: initialiseDictionary(), getChar(S), contains(Dict, String),
      generateNewEntry(Dict, String), output(S)
/* initialiseDictionary returns initial dictionary with entries for
   each ASCII character, getChar returns the next character in the
   stream, contains returns true if there is an entry for String in
   Dict, generateNewEntry returns the dictionary code for the new
   entry String, output appends S to the compressed data stream.  */
1  Dict ← initialiseDictionary();
2  CurrentString ← getChar(Stream);
3  while Stream is not empty do
4  |  Char ← getChar(Stream);
5  |  if contains(Dict, CurrentString + Char) then
6  |  |  CurrentString ← CurrentString + Char;
7  |  else
8  |  |  I ← generateNewEntry(Dict, CurrentString + Char);
9  |  |  output(I);
10 |  |  CurrentString ← Char;
11 |  end
12 end
```

Algorithm 1: Basic LZW algorithm

system in terms of its user-level features and the source code that implements them. Phase analysis can often play an important role here because of the relationship between trace phases and program features. In the hardware domain, dynamic phase detection [5] is used to optimise the allocation of resources such as memory and processor time for specific program phases.

2.3 Dictionary Compression and the LZW Algorithm

Dictionary compression algorithms are a lossless form of data compression. The aim is to eliminate "redundancy" in the data, which takes the form of repeated patterns in the data. The basic idea is that there is some dictionary with set patterns (ideally these should be known to regularly appear in the data), so that every time such a pattern is encountered, the pattern can be substituted by a pointer to the corresponding entry in the dictionary.

When the the data to be compressed contains patterns that are well known (e.g. the text is known to be in the English language), it is possible to produce fixed dictionaries that are tailored to these patterns. However, when there is no available prior dictionary, the only option is to build it up as the compression proceeds. When this is the case, the performance of the compression algorithm hinges on its ability to identify suitable entries for the dictionary from the data (i.e. large patterns that are repeated often).

One of the most successful algorithms in this regard is the Lempel-Ziv-Welch (LZW) algorithm [7], which has become one of the most popular compression methods and has formed the basis for popular data formats such as GIF images.

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

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

Recoded trace:
```
abcdddeeefgheeifghfghjbc
```

Pre-processed trace:
```
abcdefgheifghfghjbc
```

Figure 1: Pre-processing a trace from the text editor

The algorithm is presented in pseudo-code form in algorithm 1. More concrete examples of how the dictionary operates on data will be provided in the following sections.

3 Trace Abstraction

The challenge of identifying phases in a program trace is akin to the challenge faced by a dictionary compression algorithm. The ultimate aim of both techniques is to identify repeated distinct patterns in the data. This paper shows how compression algorithms can be used for phase analysis in execution traces. In our case we have chosen to use the LZW algorithm, the related work section will discuss other potentially useful compression algorithms.

Meaningful phases cannot be extracted from raw trace data; though straightforward, the pre-processing step is an important one, and is discussed in section 3.1. This is followed by an overview of how the LZW algorithm can be applied to processed traces, and a discussion on how to interpret the phase information contained in the resulting LZW dictionary. The process will be illustrated with a small example from a fictional text processor.

3.1 Pre-processing the trace

Before a trace can be processed by the LZW algorithm it needs to be pre-processed. This is for two reasons. Firstly, 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 resulting dictionary entries would be the result of repeated patterns in the naming conventions, and

Final dictionary contents: `a, b, c, d, e, f, g, h, i, j, ab, bc, cd, de, ef,fg, gh, he, ei, if, fgh, hf, fghj, jb`

Figure 2: This shows the trace compression state as it processes the trace from left to right. After each character is processed, the currentString line shows the currentString variable (see algorithm 1), and the bottom line shows what is being added to the dictionary.

tags, as opposed to the patterns of repeated method invocations that constitute program phases. Secondly, interesting phases tend to correspond to complex patterns of different method invocations, but traces often tend to consist of large repeated portions that are simply repeated calls to the same method (e.g. if a method is called in a while-predicate). This information is rarely of significance for phase identification; for the sake of identifying a phase we would not usually care whether an isEmpty method has been called 5 or 10 times in a row, all we need to know is that it has been called. Since such repeated patterns tend to result in lots of spurious phases, we remove these loop method-calls.

Thus our pre-processing stage has two steps. The first step recodes each complex method-signature into a single symbol for the trace. The second step simply removes any looped method calls. The process is illustrated in Figure 1. Without pre-processing the trace, the most repeated character sequence in the trace is the comma followed by a space, which is irrelevant to the phase behaviour. The pre-processing removes these accidental sub-sequences; any sub-sequences that are identified in the final trace are guaranteed to be actual event-sequences, which do imply actual phases. The format of the original trace used in Figure 1 is of course only illustratory conventional trace formats are much more verbose, but the process remains the same.

Our pre-processor implementation takes Eclipse TPTP traces as input, simplifies it and returns an output file which is the simplified version of the input trace. The pre-processor reads from the trace and stores each event a hash table, associating each event with a unique symbol. The output is a text file which represents a simpler version of the input execution trace.

3.2 The LZW Implementation

The LZW implementation faithfully implements the algorithm as shown in Algorithm 1. Traditionally the emphasis has been placed on the compressed output. In our case, the emphasis is instead placed on the dictionary; this is where all of the program phases are stored.

One feature of the LZW algorithm is the fact that it can process streams of data as well as static files. In other words, it does not need a complete a-priori

knowledge of the input; it compresses the file on a character-by-character basis, producing a stream of string of dictionary references as they are encountered.

In the context of dynamic analysis this is an important benefit. Depending on the program and circumstances, program traces can be extremely large, making them expensive to store and analyse. However, this algorithm does not necessitate traces to be stored in their entirety; it can analyse the trace *on-line*, during the tracing process itself. This means that the trace never needs to be stored in its entirety; the maximal amount of storage space required by the algorithm at a given point is the length of its longest dictionary entry at that time[1].

Figure 2 shows the state and dictionary contents of the algorithm as it compresses the pre-processed trace from an execution of the text processor. Before processing the trace, the dictionary is initialised (shown in the final dictionary contents entries "a"-"j"). Then, the algorithm processes the trace one character at a time, going from left to right, according to the LZW algorithm. Starting with "a", it is already in the dictionary, so it is stored in the "currentString" variable. We move on to "b" and concatenate it to "a". The string "ab" isn't in the dictionary yet, so it is added, and the currentString variable is set to "b" and the process continues.

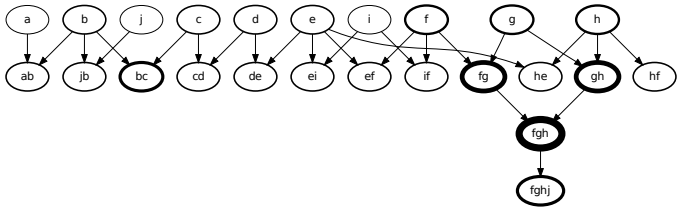
Looking at the results from this specific example, the dictionary ends up being almost as large as the initial trace; there are 25 elements in the trace, but the dictionary ends up with 24 elements, which would be perceived to be poor from a compression point of view. This can however be explained by the fact that the LZW algorithm is intended for sequences of data that are much larger; the beginning of the stream is known as an *adaptation period*, where the dictionary is populated with likely entries. The adaptation period ends when the majority of the sub-strings encountered are already stored in the dictionary this is when compression begins in earnest. Thus, in reality, when we are dealing with realistic, large-scale traces, the dictionary will usually be of a trivial size compared to the trace.

3.3 Phase Analysis

Even if the dictionary is much smaller than the trace, it will still usually contain a vast number of entries. All of these are by definition valid phases; they are all sub-sequences of trace events that have been observed at least once. However, analysing the dictionary to identify those phases that are somehow "useful" for the task at hand could seem daunting.

Fortunately the dictionary is constructed in such a way that it is possible to (a) analyse phases in terms of their relationships to each other and (b) identify those phases that are potentially more significant than others. These two aspects are elaborated below.

[1] It is important to note that this is only true for building the dictionary. The computation of phase-significance scores to pick out potentially important phases (described later) would require the entire trace.



Dictionary contents with scores: `a(1), b(2), c(2), d(2), e(2), f(3), g(3), h(3), i(1), j(1), ab(2), bc(4), cd(2), de(2), ef(2),fg(6), gh(6), he(2), ei(2), if(2), fgh(9), hf(2), fghj(4), jb(2)`

Figure 3: Hasse diagram depicting the relationships between dictionary elements

3.3.1 Relationships between phases

The contents of a dictionary constructed by the LZW algorithm adhere to a partial order. As shown in the algorithm, every time a new composite string is created, the dictionary is checked to see if it already exists. If it does, a new character is appended, otherwise a new entry is made to the dictionary. Thus, a new entry will always subsume prior entries.

For most tasks, a phase only useful if they are at a suitable level of abstraction. As an example, entry "g" in the dictionary ("copy") is not useful in itself because there is no informaiton about the context in which it is used. Ideally, the developer would want to find larger phases that provide a better insight into its typical usage context.

For this task, it is possible to exploit the subsumption relationship. Instead of trawling through the raw text of the final dictionary contents, the entries can be represented in terms of their subsumption interrelationships as a Hasse diagram, as illustrated with respect to our text-processor example in Figure 3. The "copy" node g is shown in the top row. This is linked to aggregate phases ("gh" and "fg"), which are both subsumed by a higher-level phase "fgh" ("select", "copy" and "paste" respectively). This phase is more useful than its constituents, because it provides a typical context for the use of "copy". Identifying this specific phase is however reasonably straightforward, because of the ability to trace relationships between low-level and more abstract phases in the Hasse diagram.

3.3.2 Identifying significant phases

Despite the relationships, there is still the question of how certain phases can be deemed more useful or significant than others. Looking at the Hasse diagram and ignoring emboldened nodes, there is nothing to suggest, for example, that phase "fghj" ("select, copy, paste, savefile") is any less significant than phase "fgh". Of course, from our knowledge of text editors, we know that saving a

file is a completely distinct function from copying and pasting, and intuitively would treat "fgh" as much more significant.

To address this, we augment each identified phase with a confidence score. Certain phases identified by the LZW algorithm will be encountered more often than others, and we can take advantage of this knowledge. This is conventionally ignored when LZW is applied in the field of data compression, but becomes very useful here. The notion of phase significance, will be used in the case study, and is defined as: $Sig = occurrances * size$, where *occurrances* counts the number of times the phase occurs in the original trace. The resulting phase significance scores are shown at the bottom of Figure 3, and are reflected in the width of the node borders in the Hasse diagram. Using this metric, the select-copy-paste phase discussed above is correctly highlighted as a phase that is particularly interesting.

3.4 Finding Missing Phases

In its form presented above, the LZW algorithm cannot guarantee to detect every possible phase in a trace. In other words, in a single pass over the data the algorithm is unlikely to identify every possible data pattern. The intention of the algorithm was always to identify the most effective data patterns [7], and not necessarily to be optimal. This renders the algorithm especially suitable for large data traces that cannot be easily stored and processed post-hoc. However, when the trace is relatively small, the potential for missing out potentially important phases becomes less tolerable.

In such circumstances it is possible to grow the dictionary by executing the algorithm over the data in multiple passes. For each pass the dictionary from the previous pass is used and expanded until no further dictionary entries are found. It has to be emphasised that this is only advised for small traces. If the traces are sufficiently large and comprehensive, the authors have found that a single pass tends to find all of the expected phases, as will be shown in the case study.

4 Case Study

Evaluating the LZW algorithm in terms of its ability to identify program phases is not straightforward. By construction, every entry into the dictionary is a 'phase' of some sort (a repeated sequence of function calls). However, establishing whether the dictionary includes phases that are in some sense useful is a very subjective problem. In this section we present a case study that demonstrates the feasibility of the approach, shows how the technique is applied, and provides some qualitative and quantitative insights into the generated sets of phases.

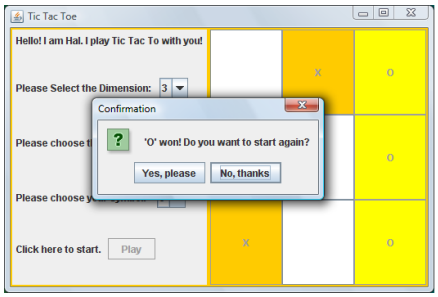


Figure 4: Screenshot from Tic Tac Toe Application

4.1 Subject System: Java Tic Tac Toe Game

The basis for our study is a small Java program called Tic-Tac-Toe. Tic-Tac-Toe is a game consisting of a minimum 3 by 3 grid. Each player is represented by X and O and takes alternating turns trying to ensure that three of their letters line up horizontally, vertically, or diagonally. The user can request varying sizes of grids up to 9 by 9. The grid layout is represented as a panel which consists of buttons, each of which represent a grid box. The buttons would respond when clicked with either an X or an O depending on whose turn it is.

While this application is not particularly large in terms of source code, it has a reasonably complex run-time behaviour. The source code of the Tic-Tac-Toe program contains 710 lines of commented Java code. A screenshot of the Tic-Tac-Toe interface is shown in figure 4.

4.2 Scenarios

The Eclipse TPTP tool was used to record the traces of the Tic-Tac-Toe program while performing a number of typical scenarios. The traces were filtered to include only methods in the immediate implementation, and to exclude any Java library methods (this is to facilitate manual analysis of the phases in the qualitative analysis). The resulting traces are in XML format. The XML files were then converted to text format using the process described in section 3.1.

Different scenarios were collected by increasing the dimensions of the board, from its minimum of 3 to its maximum of 9. Consequently, the seven scenarios vary in size and complexity. This is useful for the study, because we can contrast the performance of the LZW algorithm with respect to these different levels of trace complexity.

A game with larger dimensions will take longer and therefore will produce a larger trace, which would give the LZW algorithm greater opportunities to identify useful phases. As the first scenario requires playing with a 9-panel grid, the resulting trace is rather short and thus does not contain so many frequent events. Conversely scenario 7 will produce the largest trace.

	Trace size	Compressed size	Comp. ratio	Avg. sig.
1	157	66	57.96	5
2	337	90	73.29	10
3	623	116	81.38	16
4	1106	141	87.25	23
5	1799	168	90.66	32
6	2857	194	93.21	42
7	4263	220	94.84	57

Table 1: Trace and compression metrics

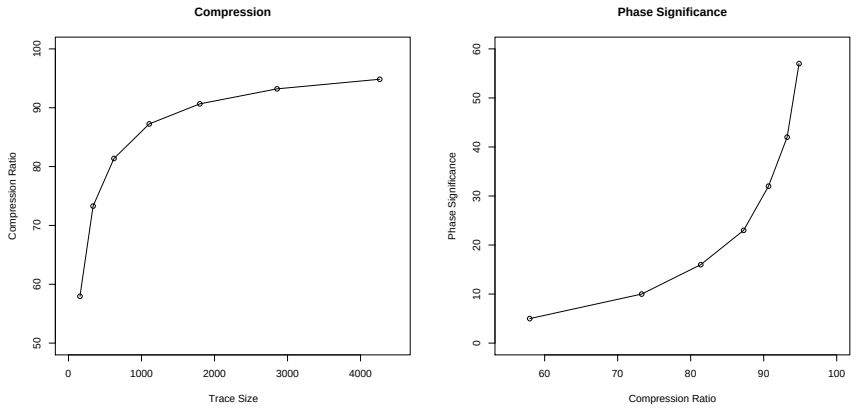


Figure 5: Relationships between trace size, compression rate and phase significance

4.3 Quantitative Analysis

Table 1 displays a selection of metrics for the (pre-processed) traces that were recorded from the seven scenarios. Trace size indicates the number of method entry points in the trace and size of compressed trace shows the number of elements in the compressed trace. The phase significance is computed as described in section 3.3.2. The compression ratio is computed as:

$$Ratio = (1 - \frac{compressedSize}{uncompressedSize}) * 100$$

Figure 5 highlights two trends that provide a useful illustration of algorithm performance. The top chart plots compression-ratio against trace size. It shows that, for smaller traces, the compression rate is relatively low, but that as the traces increase in size, and contain more repeated phases, the number of phases identified (and so the compression rate) increases rapidly and then gradually levels off. This is essentially supported by Welch's notion of an "adaptation

phase" mentioned previously [7]. The second chart plots the average phase significance against the compression ratio of each trace.

4.4 Qualitative Analysis

Ultimately, the application of the LZW algorithm will only make sense if the detected phases are useful. For that to be the case, they must (a) correspond to executions of actual program features and (b) be identifiable as such, i.e. it must be reasonably straightforward to pick out the significant phases. A further important factor is what we refer to as recall; when a phase is known to be contained in a trace, it must be identified as such by the algorithm.

To investigate the phase contents we have taken traces 2 and 7, and executed the algorithm each one. The resulting sets of phases are shown in two Hasse diagrams in figure 6. It is too small to read the contents of each phase, but provides a good idea of the number of phases identified, their interrelationships, and their relative significance (in terms of the thickness of the node edges).

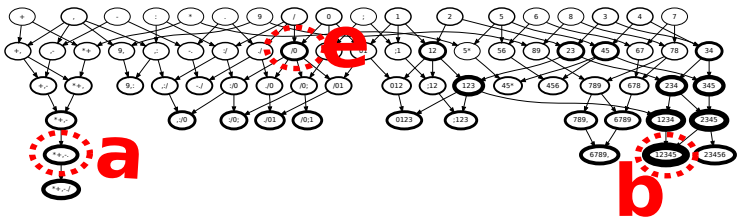
It is important to note that we are not seeking to evaluate the visual representation of the phases, of which there are many better alternatives. The Hasse diagram is a direct representation of the partial-ordering of our phases, and is simply used to provide a high-level illustration of the phases. Potentials for visualisation are discussed in the related work section.

The table in Figure 6 contains a selection of phases that would be expected, and are highlighted in the Hasse diagrams. Phase a corresponds to the selection of a move by the user. Phase b corresponds to the process of committing a move to the grid. Phase c corresponds to the high level phase where a player selects a move and that move is committed to the board. Phase d corresponds to a move being selected by the computer and committed to the board. Finally phase e is a very simple phase that corresponds to the retrieval of a co-ordinate from the grid.

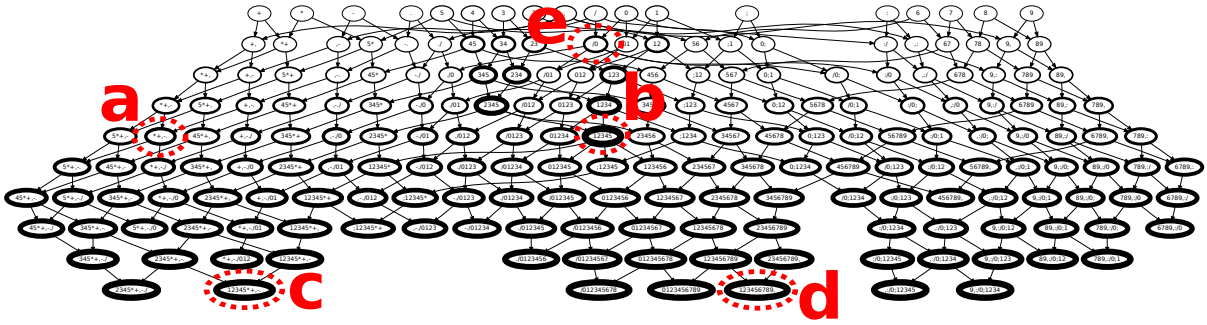
From the results shown in figure 6, there are two remarks to be made:

4.4.1 There is a direct relationship between the number of events or method calls in a phase, and the number of times the phase must appear in the trace in order to be identified by the LZW algorithm

Every time a sub-phase is identified, if it is not in the dictionary it is added, and the matching process has to start afresh. Thus, if there is a phase with seven elements, it must occur at least as often as that in the trace, so that all of its subsequences can be added too. If we permit the dictionary of phases to be built from multiple runs of the LZW algorithm over the same trace, this constraint becomes less of an issue, and large phases can be identified even if they occur infrequently. In our examples, the phases are produced from two iterations of the LZW algorithm.



(a) Trace 2



(b) Trace 7

a - player selects move

MyGrid.actionPerformed
MyTicTacGrid.playerDoes
MyTicTacIndex-init-
MyTicTacIndex.stringToIndex
MyTicTacIndex.testIndex

b - Move committed

MyTicTacReference.setBox
MyTicTacReference.updateEmptyBoxes
MyTicTacReference.sequenceLength
MyTicTacGrid.gameOver
MyTicTacReference.getEmptyBoxes

c - Move committed, player selects move

MyTicTacReference.setBox
MyTicTacReference.updateEmptyBoxes
MyTicTacReference.sequenceLength
MyTicTacGrid.gameOver
MyTicTacReference.getEmptyBoxes
MyGrid.actionPerformed
MyTicTacGrid.playerDoes
MyTicTacIndex.-init-
MyTicTacIndex.stringToIndex
MyTicTacIndex.testIndex

d - move committed, algorithm selects move

MyTicTacReference.setBox
MyTicTacReference.updateEmptyBoxes
MyTicTacReference.sequenceLength
MyTicTacGrid.gameOver
MyTicTacReference.getEmptyBoxes
MyTicTacGrid.algorithmDoes
MyTicTacReference.chooseAMove
MyTicTacReference.ramdomChoice
MyTicTacReference.isEmpty
MyTicTacIndex.-init

e - Obtain co-ordinates

MyTicTacIndex.getX
MyTicTacIndex.getY

Figure 6: Hasse Diagrams for traces 2 and 7, along with a selection of phases

4.4.2 Significant phases are easier to identify when they occur in different contexts

In the introduction, a phase was introduced as a *distinctive period of activity*. A phase is easier to distinguish when it occurs often in multiple contexts. This becomes apparent in the Hasse diagram (b) for trace 7. Phase b occurs often in various contexts; it is used both for committing the move by the human, as well as committing the move by the machine to the grid. On the other hand, as the phases become larger, they occur in fewer contexts, which results in many larger phases with a similar significance score. At a high level, the trace simply consists of the user carrying out a move, followed by the computer, followed by the user (phases c and d). It becomes difficult to distinguish at which point one phase begins and the other ends, because unlike phase b, they are only every used in the same context.

4.5 Discussion

Our preliminary results from this small case study are promising. They indicate that dictionary compression algorithms do identify useful phases along with their interrelationships. As with any technique that is based on dynamic analysis, the usefulness of the results ultimately depends on the contents of the input traces. In our case the identification of a useful selection of significant phases requires a set of input traces that are large and diverse. Larger traces facilitate the identification of large phases. Trace diversity reduces the number of false-positive phases by emphasising phase boundaries.

It is important to note that this work only considers how to identify phases and their interrelationships, and does not explicitly address the task of browsing them. At first glance the number of phases identified may seem overwhelming. However, the fact that the phases are organised in a partial-order (as visualised in the Hasse diagrams), and are associated with significance scores, substantially attenuates this problem. Hasse diagrams can be explored by using a variety of well-established techniques to home-in on useful sub-partitions of concepts (or phases in our case). As an example, such exploration techniques are routinely used in the field of Formal Concept Analysis [8], where an analysis can produce vast concept lattices (also viewed as Hasse diagrams).

There are of course several open questions that have not been covered by this case study, which will need to be addressed by a more systematic study. This is elaborated in section 6.

5 Related Work

5.1 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 [3]. Sherwood *et al.*

[5] and Cornelisson and Moonen [6] 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 [9] 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 [4] 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 [10]. This can only be achieved by abstracting away facets of the information that are considered to be important. For example, in visualising a trace as a signal, Kuhn and Greevy ignore method names, data values etc. and consider only the depth of the call-stack at each point in the trace. One inherent danger is that pertinent or interesting information can be lost in the abstraction process. With most (non-interactive) visualisation approaches, 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.

The only extent to which our technique relies on abstraction is in the pre-processing step, where immediate repetitions of elements are reduced and elements are recoded to symbols. This makes it possible to clearly establish phase-interrelationships in the structured and hierarchical manner shown in the Hasse diagrams. Of course, the flip-side to this is that there is a lot of information to contend with, and in this respect, combining it with a more suitable visualisation or interaction technique would be very useful. This idea is elaborated in the future-work section.

5.2 Application of Information Theoretic Approaches to Trace Analysis

Data compression is one of the core topics in the field of Information Theory, which is broadly concerned with quantifying, coding and compressing data. Here, the techniques that have been used to analyse program traces are divided into signal processing and compression techniques.

5.2.1 Signal processing

Although previously mentioned in the context of visualisation, Kuhn and Greevy [4] ultimately represent the trace as a signal. This can therefore be analysed (albeit superficially) by a range of established signal-processing techniques that can suggest those parts of the trace that correspond to a feature.

Huffmire and Sherwood [11] propose a wavelet-based phase classification approach. This is intended for the analysis of traces of data (represented as a large matrix, where each row corresponds to a memory address and each column corresponds to the execution time). In summary the trace can treated as a picture, which permits the application of wavelet-based image-classification techniques to identify segments of the trace that are similar. With respect to the

work presented in this paper, it is intended for a different scenario. Their work is intended specifically for data traces, whereas our technique is intended for traces that can be recoded as sequences of discrete symbols. Whereas their phase classifications are primarily intended to guide resource allocation at runtime, our phases are intended to aid program comprehension tasks.

5.2.2 Trace Compression

Larus [12] used a compression algorithm to capture a complete execution profile from an execution trace in terms of its basic blocks and control-flow edges. The rationale is that, at such a low level, traces often fail to precisely record a sufficient amount of information in a concise manner. To address this problem, Larus used the SEQUITUR dictionary compression algorithm [13], which compresses a string into a context-free grammar. In essence, he presents a domain-specific enhancement of the compression algorithm that is tailored to the compression of low-level program traces.

Gao et al. [14] suggest an alternative trace compression approach that is based on the GZIP algorithm, another popular dictionary compression approach. They point out that, although the compression ratio is not as high, the rate of compression is up to 40 times faster than the SEQUITUR algorithm.

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. Larus and Gao's work attempts to provide compression algorithms to address this need. Tools such as Eclipse TPTP incorporate simple compression algorithms by default. However, the work presented in this paper does not suggest the use of compression algorithms for the sake of compression. Compression algorithms (at least dictionary compression algorithms) serve to eliminate repetitions. This paper proposes that phase-analysis techniques can take advantage of dictionary algorithms in general to better identify phases and their interrelationships. The SEQUITUR algorithm does however have the useful property that it makes hierarchies within the data to be compressed explicit, a feature that is particularly valuable for phase-analysis and its potential use for phase identification is discussed in the future work section.

5.3 Feature Identification Approaches that use Rich Trace Information or Incorporate Source Code-Level Information

The challenge of feature-identification is to answer the question: Which source code statements are responsible for implementing feature X?. Achieving this without executing the source code is challenging [15]; the code that implements a feature is often spread across multiple files and functions within the system, whose interrelationships can often only be established by monitoring their runtime interactions.

Although tracing is useful, it is rarely sufficient to simply execute the feature and record the executed statements. A host of challenges have to be addressed,

such as identifying a sufficiently broad set of traces, and ensuring that no spurious statements are included. The problem has its roots in the work on software reconnaissance by Wilde et al. [16], 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. [17] have attempted to combine dynamic analysis with static analysis and Formal concept analysis to identify potential features. More recently, Greevy and Ducasse [18], Kothari et al. [19] and Wanatabe *et al.* [20] have all worked on the identification of trace phases to identify features in the source code.

The above approaches are constrained to traces that record a substantial amount of information about the underlying system, or static analyses that extract facts at a source-code level. They are either restricted to traces of object-oriented systems, where traces include information about individual object-instances, or have to be combined with static-analysis techniques of the source-code. The approach presented in this paper is more light-weight; the trace can be of any system, as long as it can be recoded to a sequence of symbols. The output is not merely a discrete set of phases, but a hierarchy of phases that makes their interrelationships explicit.

6 Conclusions and Future Work

Dictionary compression algorithms aim to eliminate any repetitions in the data to be compressed. The challenge of identifying these repetitions in data is analogous to the challenge of identifying phases (repeated sequences of methods or events) in program traces. This paper has shown how the abilities of dictionary compression algorithms can be leveraged to address the phase-identification problem.

So far, the approach has only been demonstrated with respect to a small number of traces from a small system. This has served its purpose of demonstrating the feasibility of the approach, but cannot provide any significant insights into the usefulness of the phases, or the scalability of the approach. We are currently investigating the use of a more diverse set of traces from more complex systems to produce a more comprehensive and systematic evaluation.

As noted in the related work section, there are a host of alternative dictionary compression algorithms that would be suitable for the task of phase identification. We intend to investigate the use of such algorithms, particularly the SEQUITUR algorithm, and to compare the resulting phases (and phase hierarchies) to those produced by the LZW technique.

Finally, we intend to investigate suitable visualisation approaches for the phase-hierarchies that are produced. Hasse diagrams are more useful from a set-theoretical perspective than a visual one, and there are numerous visualisation techniques that are probably better suited for interaction, and browsing through potential phases.

Acknowledgment

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.

References

[1] T. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[2] V. Basili, "Evolving and packaging reading technologies," *The Journal of Systems and Software*, vol. 38, no. 1, pp. 3–12, Jul. 1997.

[3] 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)*. USENIX, 1998, pp. 219–234.

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

[5] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE Micro*, vol. 23, no. 6, pp. 84–93, 2003.

[6] 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.

[7] T. Welch, "A technique for high performance data compression," *IEEE Computer*, vol. 17, no. 6, pp. 8–19, Jun. 1984.

[8] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

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

[10] 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*, vol. 81, no. 12, pp. 2252–2268, 2008.

[11] T. Huffmire and T. Sherwood, "Wavelet-based phase classification," in *Proceedings of the 15th International Conference on Parallel Architecture and Compilation Techniques (15th PACT'06)*. Seattle, Washington, USA: ACM, Sep. 2006, pp. 95–104.

[12] J. Larus, "Whole program paths," in *PLDI*, 1999, pp. 259–269.

[13] C. Nevill-Manning and I. Witten, "Compression and explanation using hierarchical grammars," *Computer Journal*, vol. 40, no. 2/3, pp. 103–116, 1997.

[14] X. Gao, A. Snavely, and L. Carter, "Path grammar guided trace compression and trace approximation," in *HPDC*. IEEE, 2006, pp. 57–68.

[15] N. Walkinshaw, M. Roper, and M. Wood, "Feature location and extraction using landmarks and barriers," in *23rd International Conference on Software Maintenance (ICSM'07)*. IEEE Computer Society, 2007.

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

[17] T. Eisenbarth, R. Koschke, and D. Simon, "Locating Features in Source Code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[18] 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.

[19] 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.

[20] 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.