Using a Variant of Sliding Window to Reduce Event Trace Data

Andy Zaidman, Serge Demeyer

Lab On Reengineering
University of Antwerp
Middelheimlaan 1, 2020 Antwerpen, Belgium
`Andy.Zaidman@ua.ac.be, Serge.Demeyer@ua.ac.be`

Abstract. Understanding how components interact with their neighboring components is a necessary prerequisite for the evolution of legacy software systems. Dynamic program analysis is known to provide deep insight in component interaction protocols, however such techniques must all cope with a tremendous scaleability problem. Therefore, this paper proposes a heuristic which reduces program traces based on a frequency spectrum analysis of program events. Based on a small case-study, we conclude that the heuristic is able to identify interesting component interaction patterns in program traces that consist of one to two million events.

1 Introduction

Software engineers who are specialized in the field of dynamic analysis often have to contend with large amounts of trace data. This trace data is used for regaining architectural insight, profiling an application, measuring performance and many other purposes.
To give an idea of the amount of trace data that is generated: a well-structured Java program consisting of 5 classes generates approximately 6000 events while running for 1 second. Although this execution scenario lasts only one second, it is easy to see that tracing large scale industrial applications would lead to sizes of trace data that are very difficult to handle [1].
Moreover, for the purpose of regaining architectural insight these huge amounts of detail of the trace are not needed: the event trace contains all the method calls that a program makes, but for the purpose of regaining architectural insight, we are mainly interested in the component interaction protocol. A lot of events don't influence this interaction protocol, so we aren't really interested in them.
The heuristic we propose is based on a combination of *Frequency Spectrum Analysis* [2] and a *sliding window* mechanism, well-known in the world of telecommunications [3]. In using both these techniques we are able to identify *key events*, events we find interesting, and mark regions where there are a lot of these key events.
For the purpose of regaining architectural knowledge through dynamic analysis, reducing the amount of trace data is very important to keep algorithms for detecting patterns in the trace as efficient as possible. Bottom-line is that we want to identify key events and their surrounding events and continue working with them and not the entire trace.

2 Context

To give a better idea as to how much data is really involved in a trace capturing mechanism, we look at Table 1. As you can see, the first three values in each column show the amount of trace data involved when tracing the program.
The three following values were collected when using a primitive, but effective reduction-scheme. Specifically, we used a filter on the trace capturing mechanism which excluded all method-calls to methods from classes we decided we were not interested in. For the example in Table 1 we excluded all classes from the Java API[1](Java Standard Edition, version 1.4.1). This kind of reduction operation should not be confined to excluding low-level method calls; also method-calls to parts of the software we're not interested in for our reengineering or understanding purposes can be filtered out in a similar way, e.g. filtering out all calls to the XML Xerces library, that's frequently used for logging purposes in Java programs.

	Jetty 4.1.4	Jext	jEdit 4.1	Tomcat 4.1.18	Fujaba 4
Classes (total)	11423	11094	10603	13258	15630
Events	948913	3215983	2071216	6582356	12522380
Unique events	2231	10038	9900	4925	858505
Classes (with filter)	3707	3835	3344	3482	4253
Events	746207	702607	182425	1076173	772872
Unique events	1128	1621	1666	2359	95 073

Table 1. Magnitude of trace data.

As Table 1 shows, the amount of reduction varies a lot: in the best case the filtering mechanism reduces the trace to about 6% of the original trace (in the case of Fujaba). Worst case we get 78% of the original trace (the Jetty case study).
This primitive filtering technique helps in keeping the volume of trace data under control, but it also partially solves another fundamental problem when tracing programs: the *probe effect*. This can be described as the interference of the monitoring process of the software under consideration on the program itself [4]. This is especially true in the case of multi-threaded programs, but also plays a crucial role in performance critical applications.
This effect can be compared to the well-known *uncertainty principle* from Heisenberg (discovered in 1927) in the field of quantumphysics that postulates that it is impossible to know the exact place of a particle and the exact impulse at the same time. It goes even further, because the more you know about the place of the particle, the less you know about it's impulse, and vice versa.
What we call the *probe effect* is really not that different, because the more trace data we want to capture, the less accurate the execution patterns become (due to the interference from the tracing process). The opposite is also true.

[1] The Java API contains all the libraries for standard Java functionality, e.g. strings, math, inter process communication, ...

3 Heuristic

The heuristic combines two techniques, namely *Frequency Spectrum Analysis* and a variant of a *sliding window* protocol. In this section we'll discus both and how we can combine them.
Before we continue, perhaps now is the time to give a definition to an *event*. From our point of view an event is something like:

```
Class.methodname(Type of parameter 1,
                     Type of parameter 2, ...)
```

Notice that we don't use object identifiers (OID's). As we are interested in making an abstraction, we aren't interested in specific instances of classes.

3.1 Frequency Spectrum Analysis

The concept of *Frequency Spectrum Analysis* (FSA) as introduced by Thomas Ball [2] has some interesting properties:

- The use of low versus high frequencies to partition the program by levels of abstraction.
- The use of frequency clusters to identify related computations in the program.
- The use of specific frequencies to find computations related to the program's observed behavior (e.g. how much input/output)

Table 1 shows that the number of unique events is rather limited with respect to the total number of events, which clearly means that there is a lot of repetition in the executed methods. When we combine this knowledge with the concept of FSA, we will try to identify ”*key events*”. From Figure 1 we will deduce what we understand under key events. What we can see here, is that there is a small number of methods which are responsible for most of the events. In the case of Fujaba, depicted in Figure 1, 1% of the methods are responsible for 75% of the events.
How can this knowledge help us in discerning key events?

- Frequently called methods are probably an indication of low-level functionality, called by many different types of methods and a such they give no real indication of an interaction protocol pattern (they can however be part of a pattern, but they are not necessarily an indication of a pattern). We consider them to be non-important.
- Methods that are called infrequently also give no indication for patterns. These are also considered non-important.

This leaves us with an interval of key events. Where the exact boundaries for this interval are, is hard to determine. It depends on the level of granularity when performing the trace operation, the kind of application,...

3.2 Sliding window mechanism

We were looking for a mechanism that would do three things:

- Reduce the size of the trace data.

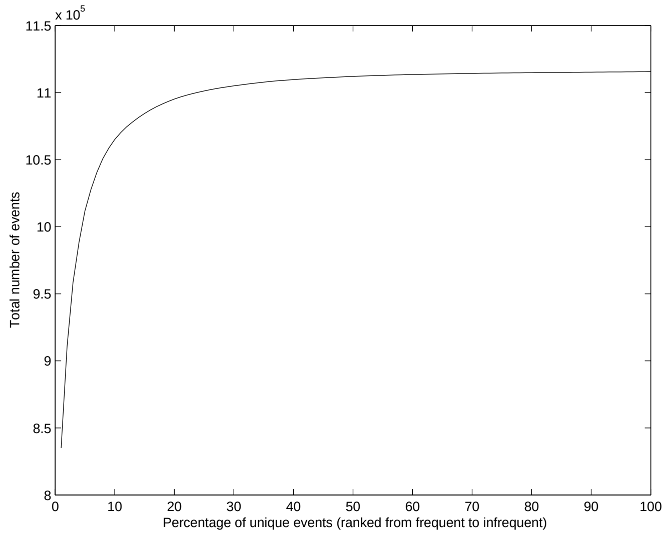

Fig. 1. FSA from Fujaba 4: a small number of methods are responsible for a large number of events

– Retain key events that we have identified during the FSA.
– Since our ultimate goal is to find sequences of events that can be abstracted into a pattern of execution, we are not only looking at the key events. Thus, we'll also be looking at the (non-key) events that surround the key events.

To try and meet these three goals, we set up a mechanism that - at first sight - resembles a sliding window mechanism, well-known in the world of telecommunications [3]. We will explain the inner-workings of the mechanism with the help of Figure 2.

We take a window to be a small view on the huge event trace. This window moves over the input data. When the content of the window is deemed interesting, the contents of the window, i.e. a sequence of events, is then redirected to the output. So, what does the algorithm do exactly? It searches for *key events*, but it also looks at the events that precede and follow the key events, i.e. it looks at the immediate *surrounding* of the key event. If, for example, event$_{i+2}$ is also marked as a key event, we want to *enlarge* the window size, because we are more interested in the whole sequence of events that contain these two key events, then in the separate two sequences that can be formed, e.g. [event$_{i-2}$, event$_{i+2}$] and [event$_{i}$, event$_{i+4}$].
The algorithm we present here has a variable window size, because of several reasons:

– interesting parts of the trace may be very concentrated in locality. If we find an interesting sequence $>$ windowsize, we are now perfectly able to adapt our windowsize to it and output the entire sequence.

$$\begin{array}{l} \ldots \\ event_{i-3} \\ \left.\begin{array}{l} event_{i-2}\\ event_{i-1}\\ \mathbf{event}_i \\ event_{i+1}\\ event_{i+2}\end{array}\right\}\text{window} \\ event_{i+3} \\ \ldots \end{array}$$

Fig. 2. Example of a window (size 5). Standard windowsize, because there is only one key event in window.

– if we were to switch to a fixed windowsize, we would also introduce redundancy in the output. If two important events lie next to each other, but both lie in different windows, it would introduce duplication as surrounding events would be doubled (in fact, the second key event would lie in the surrounding of the first window, and the first key event would lie in the surrounding of the second window).

This chaining effect causes some troubles, especially when the `predefinedWindowsize` is quite large (>5): the reduction of trace data becomes insignificant. For this reason we've added a second parameter, the *importance* of a sequence (`predefinedImportance` in the algorithm below). This parameter tells how many key events are expected within a sequence, in other words it puts a lower boundary on the density of important events in a sequence.

The algorithm in pseudo-code:

```
// predifinedWindowsize and predefinedImportance
// are the parameters of the algorithm
distance = 0;
importance = 0;
while(still more events)
{
   if(currentevent == important)
   {
      foundImportant = true;
      distance = o;
      ++importance;
   }
   else
   {
      if(! foundImportant)
      {
         if(eventqueue.size() > predefinedWindowsize)
         {
            eventqueue.removeFirst();
         }
         else
         {
            ++distance;
         }
      }
      eventqueue.add(currentEvent);
                  // add current event to window
   }
   if((distance > predefinedWindowsize)
                  & (importance > predefinedImportance)
   {
      sequencefound = true;// write the entire
                                // eventqueue to output
      distance = 0;   // reset distance
      importance = 0; // reset importance
      foundImportant = false;
   }
   currentEvent = trace.getNext(); // get next event
}
```

The most important question remains however: how good is this heuristic. At this moment, we do not have conclusive data on this. Preliminary results show that with the right parameters for the heuristic, the reduction is considerable (up to a factor 90 smaller). When used for the purpose of regaining architectural insight, we do have enough data to work with. Further tests are needed to determine the quality of the reduction operation, i.e. aren't we throwing away too much data.

4 Conclusion

This paper presents a novel heuristic for reducing the huge amounts of trace data that a program leaves behind. Reduction is in many cases necessary for optimal post-mortem analysis. Further tests will verify the quality of the reduction operation.

References

[1] Smith, R., Korel, B.: Slicing event traces of large software systems. In: Automated and Algorithmic Debugging. (2000)

[2] Ball, T.: The concept of dynamic analysis. In: ESEC / SIGSOFT FSE. (1999) 216–234

[3] Clark, D.: Window and acknowledgement strategy in tcp (1982) RFC813.

[4] Andrews, J.: Testing using log file analysis: tools, methods, and issues (1998) Proc. 13 th IEEE International Conference on Automated Software Engineering, Oct. 1998, pp. 157-166.