2010 14th European Conference on Software Maintenance and Reengineering

Utilizing Debug Information to Compact Loops in Large Program Traces

Del Myers and Margaret-Anne Storey
Department of Computer Science
University of Victoria,
Victoria, BC, Canada
Email: delmyers.cs@gmail.com, mstorey@uvic.ca

Martin Salois
Defence Research and Development Canada
Valcartier, QC, Canada
Email: martin.salois@drdc-rddc.gc.ca

***Abstract*—In recent years, dynamic program execution traces have been utilized in an attempt to better understand the runtime behavior of various software systems. The unfortunate reality of such traces is that they become very large. Even traces of small programs can produce many millions of messages between different software artifacts. This not only affects the load on computer memory and storage, but it also introduces cognitive load for users, affecting their ability to understand their software. This paper discusses an algorithm which combines data from multiple sources–dynamic execution traces, source code, and debug information–in order to drastically reduce the number of messages that are displayed to the user. We introduce the algorithm and apply it to the Java programming language. The algorithm is employed against several Java software systems to investigate its effectiveness in compacting loops. Its usage is demonstrated in the context of a visualization based on UML Sequence Diagrams.**

***Index Terms*—reverse engineering; program traces; algorithm; compaction; visualization**

I. Introduction

Software is complex and can be very difficult to understand. Maintenance of software systems often requires some level of reverse engineering as developers investigate source code in order to understand their software. Unfortunately language features such as polymorphism or late binding limit the expressive power of source code and static analysis. Dynamic techniques such as execution traces help to address these problems. Execution traces offer the advantage that they report precisely the actions of a program at runtime.

Execution traces suffer from a number of shortcomings, however. One is that they tend to be very large. This is a critical shortcoming when software tools attempt to visualize the contents of execution traces. It is very important that information be hidden [1], removed [9] or compacted [4] in order to display large amounts of information.

Another shortcoming is that raw execution traces are not able to express the semantics of why traced events occurred in the context of the originating source code. Our previous experiments with programmers showed that they rely heavily on source code even when presented with a sequence diagram visualization of an execution trace [1]. Guéhéneuc and Ziadi have made a similar observation and suggest that tools and visualizations of dynamic execution traces should incorporate information from static source code [5].

These challenges led us to develop a research project called Dynamic Interactive Views for Reverse Engineering (Div$_{\exists}$ı).[1] Div$_{\exists}$ı was developed as a set of plug-ins for the Eclipse integrated development environment (IDE). It offers users of Eclipse the ability to create execution traces of their Java applications or Eclipse plug-ins, and inspect them using graphical views which employ combined static and dynamic analysis. In developing this project, we discovered that the size of visualizations can be drastically reduced if the source code that defines the executed program is taken into account.

The main contribution of this paper is an algorithm that is able to compact execution traces through the use of debug information and loops found in source code. The algorithm is used for visualizations and tools that integrate dynamic program behavior with static source code. The solution makes use of information that is commonly available on most platforms.

The remainder of this paper proceeds as follows. Section II discusses related work. Section III describes the algorithm in detail. Section IV describes an experiment that uses several industrial software systems to evaluate the effectiveness of the algorithm. Section V describes how the algorithm can be applied in reverse engineering tools by way of an example in the Div$_{\exists}$ı tool. Section VI discusses the limitations of the method and offers indications of when other methods should be employed. Finally, section VII summarizes the contributions and concludes this paper.

II. Related Work

Much research and many tools have been presented that attempt to aid users in their understanding of the dynamic processes of software. Hamou-Lhadj and Lethbridge [6] and our previous work [1] both present surveys of such tools and techniques. Most center around various visualization methods which attempt to display large amounts of data in compact ways. Some recent examples are Massive Sequence and Circular Bundle views [4]. Both of these methods use visual attributes such as proximity, color, and line thickness to relate software artifacts by the "calls" relationship while minimizing the screen or print area that is used. The two views are synchronized in a single tool and give users the ability to gain

[1] http://diver.sf.net

1534-5351/10 $26.00 © 2010 IEEE
DOI 10.1109/CSMR.2010.19

an overview of the trace and effectively locate features in the software. These examples rely purely on execution traces and not on source code.

Object-message based diagrams such as UML sequence diagrams are also popular. The Eclipse Test and Profiling Tools Platform [21] is an example of one such tool. These tools have the advantage that they can display detailed information about the trace, but they suffer from the fact that they require a large area for display. It can be difficult to infer, for example, repeated patterns in the software since the repetitions cannot be displayed simultaneously.

One common attribute of execution traces is that they are highly repetitive. This allows for high compression ratios in program traces and therefore some lessons can be learned from trace compression literature. One source of heavy repetition in execution traces is that of the simple loop. Ketterlin and Claus [10] use loop recognition to compress the values found in traces of memory. Their technique is able to compress execution traces more than 8 000 times better than standard file compression techniques such as Bzip2 [18]. For the work in this paper, we are not with concerned trace *compression*, since our goal is not to reduce the size of stored trace data. We are interested in what we will call trace *compaction*. By compaction, we mean the reduction of the amount of trace information that must be displayed in a visualization of a trace. The trace compression literature indicates that loops are a good candidate for such reduction.

The repetitive nature of execution traces has also proven useful in the field of trace analysis. Safyallah and Sartipi [15] use data mining techniques such as *sequential pattern mining* to extract repeated execution patterns from a trace and display them to the user. Hamou-Lhadj and Lethbridge [7] use repeated sub invocations to create directed-acyclic graphs which are used to define the concept of a *comprehension unit*. Comprehension units were used again in a later work to aid in the removal of utility methods from the trace visualization [8]. Bohnet *et al.* [2] use a related approach in a metric named *call fingerprints* based on the similarity between the sub-invocations of different parent invocations. Putting a threshold on this metric allows traces to be pruned by eliminating invocations that are too similar. All of these examples work only on execution trace data without considering source code.

Some work has attempted to match execution traces to source code. As early as 1994, Kimelman *et al.* [11] offered an application called PV which had an *active loop view*. This view acted as a postmortem debugger, highlighting source code as it is reached in a play-back of an execution trace. PV was tied to the IBM AIX system, relying on the trace data that it supplied. The process given does not, in fact, attempt to compact traces for visualization, but it does help users to locate source code from within an execution trace. Systä [17] augmented static visualizations of Java software in Rigi [14] by annotating them with code coverage information from execution traces. Our approach is different in that it uses code information to compact and visualize execution traces.

Finally, much work has focused on finding repeated patterns in order to aid in visualization, particularly through the use of loop detection. MaintainJ [13] follows a simple process of removing sequential calls to the same method. Jerding *et al.* removed repetitive patterns in software visualizations using hashing and loop detection. Our earlier work, the OASIS Sequence Explorer [1], utilized an algorithm that is able to detect nested loops and allows users to swap out different iterations of the loop to see how the program behaved on each iteration. Lui *et al.* [12] search UML sequence diagrams to infer loops. In this process, sequence diagrams are treated as long character strings, and loops are inferred by creating suffix trees for the strings. The process is not specific to execution traces, but it can be easily applied to them. None of these approaches use source code information.

Briand *et al.* [3] treat the issue as a modelling problem. They use the Object Constraint Language (OCL) to unify extensive models of Java execution traces and UML Sequence diagrams. They compact loops in sequence diagrams by using combined fragments. However, the use of OCL requires that they strictly comply with the UML and its limited vocabulary. They do not offer an explicit algorithm or set of procedures for applying their process.

All of the work discussed here indicates that the field of reverse engineering has been quite active with respect to efforts to reduce the visual complexity of dynamic execution traces. Our work combines key elements from the previous research. One insight is that dynamic traces are very repetitive and that these repetitions can be expressed as program loops. Another insight is that loops are expressed in program source code. We propose a simple process that combines elements of dynamic program behaviour and the definition of the program in static source code. This gives us the ability to greatly compact execution traces for the purpose of visualization while unifying elements of dynamic and static reverse engineering.

III. The Algorithm

The basic idea of the algorithm is relatively simple. We consider one invocation of a program statement at a time and try to compact all of the statements that it invokes. What constitutes an invocation is language and application dependent. Method or function calls are some of the most commonly traced invocations, but other statements, such as variable assignments or expressions, could also be considered invocations.

We use the source code representation of the program to compact the trace. In order to to this, it must be possible to match a traced invocation to its original representation in source code. Debug information is used for this purpose. A standard debugging compiler will typically record the source code line number of an invocation. However, a line number is not sufficient to make a match in source code, since many statements may exist on a single line. The trace data must also contain a representation of the invocation being made which can be matched to the source code. Most compilers will store the signature for method or function invocations in symbol tables or within the debug information in the executable. This

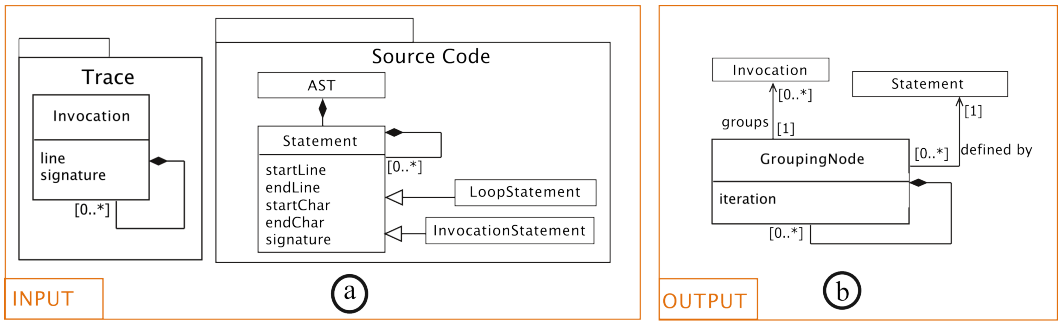

Fig. 1. The data structures. (a) is the input data, and (b) is the output data

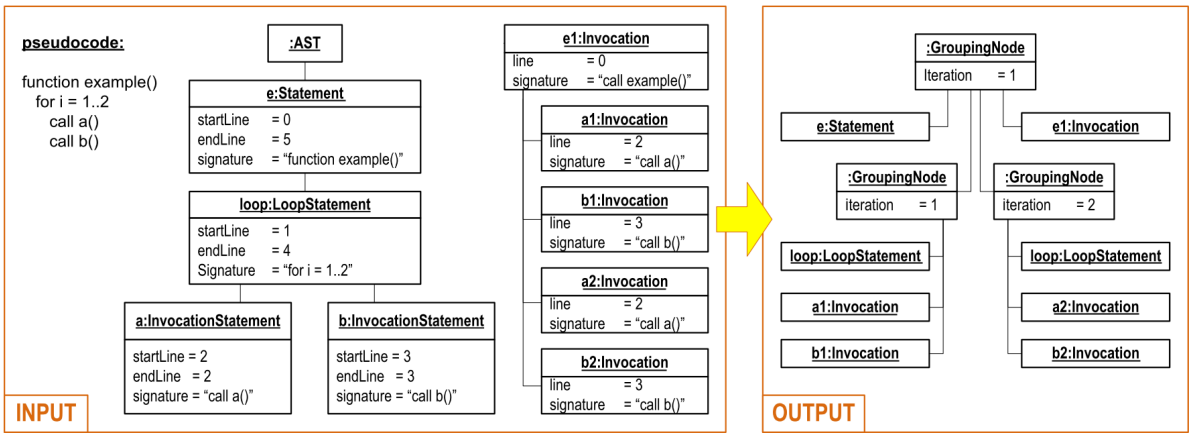

Fig. 2. An example transformation from the input data to the output data

makes method or function calls particularly good candidates as input to our algorithm. Some debugging compilers may also store the column value for the originating source code for an executed instruction. In such cases, the column value may be used instead of a signature.

With these three sources of information (source code, debug data, and trace data), the remaining problem is to match invocations to the loops in which they are contained in the originating source code. Once this is done, we can prune loops by culling out the uninteresting iterations. The following subsections discuss how to match invocations to source.

A. Data Structures

We use several simple tree data structures as input for the algorithm (see Figure 1(a)). The first data structure is a call tree stored for the trace. Two data items must be stored with each invocation in the call tree: a line number and textual representation of the invocation which we call a *signature*. Both can be retrieved from the debug information or symbol tables stored with the program.

The second data structure is the abstract syntax tree (AST) of the source code. Standard ASTs for many programming languages will work. The most important part of the AST is the statement. We define statements as any program construct, and a statement may contain any number of other statements. We list two specializations of statements–loops and invocations–because they are important concepts in the algorithm. Invocation statements require the program to do some work such as invoke a function or assign a variable. A class definition is an example of a statement that is not an invocation. All statements must be associated with the region of text that they occupy in the originating source (`startLine`, `endLine`, `startChar`, `endChar`), and a signature. These will be used to match them to the invocations in the trace.

The objective of the algorithm is to group repeated calls found within loops. The GroupingNode tree structure of Fig. 1(b) is used for this purpose. A GroupingNode has one statement which defines a group of invocations. This statement could be the block that defines a loop or a method. It also contains a list of invocations which occur within the group. In these elements, the GroupingNode mirrors the abstract syntax tree. Loops may repeat (i.e. iterate) during the execution of a program. When this occurs, a new GroupingNode for the loop statement that is being repeated is created with a new list of invocations that occur within the loop. This new GroupingNode represents a higher iteration of the same loop statement, so it is given a higher iteration count.

An example of the transformation from the input data to the output data is shown in Fig. 2. In this example, we have the AST for some code that consists of a single loop which

is executed twice and invokes two methods each time. The corresponding trace is shown beside it. It contains 5 method invocations (one start invocation and 4 children). On the far right, the data is combined into one grouping tree.

The grouping tree is used to reduce the amount of data that must be visualized. Each GroupingNode has an iteration associated with it. In order to reduce the size of the visualized trace, we simply display one iteration at a time. A selection critera must be used to decide which iterations to show. The experiment in section IV will use one criterion. It is also possible to allow users to select iterations interactively as will be discussed in section V.

B. Algorithm Details

The main part of the algorithm is listed in Fig. 3. It is called Group-Invocations, and it performs the transformation explained in section III-A. Due to space restrictions, Group-Invocations is the only part of the algorithm listed in full detail. Other details will be explained within the text.

The algorithm takes an invocation and its associated AST statement as input. The invocation must be one that is able to have child invocations, such as a method call. The associated statement for this invocation is the method definition and not the originating call.

Since the purpose of the algorithm is to associate invocations to corresponding statements, the first five lines set up associative structures. The GroupingNode tree that will be the output of the algorithm is created on line 3 and is rooted at $root$. m (line 4) is used to look up the statement associated with an invocation. gs is used as a quick look-up into the GroupingNode tree. When looking up a GroupingNode for the statement representing the loop, the algorithm is only concerned with its current iteration, so gs contains references to the GroupingNode that represents the latest iteration of a loop for a given statement. The references held in gs are used by the sub-algorithms Get-Grouping and Next-Iteration to update the tree structure. Lines 4 and 5 initialize the associations in m and gs.

Lines 6 through 31 contain the main loop of the algorithm in which each sub-invocation is processed. The first step (line 7) is to locate the source code that caused the invocation of the current sub-invocation c. This is done by utilizing the debug information that has been stored with the trace. The sub algorithm Associated-Statement traverses the AST starting at statement a to find the line that matches the invocation c. Once that line is found, the statements on that line are matched to the signature of c. In the case that multiple statements with the same signature as c occur on the same line, the first statement is returned. Parent-Block (line 8) traverses up the AST, starting at $m[c]$ to retrieve the block statement (i.e., loop) b in which the previously located statement is contained. If no loop can be found, then a is returned.

Get-Grouping creates the necessary GroupingNodes needed to associate c with b. First, the associative array gs is checked to see if b already has a GroupingNode associated with it. If it does, then $gs[b]$ is returned. If it does not, then a new GroupingNode associated with b is created and put in $gs[b]$. Get-Grouping must also ensure that GroupingNodes exist for all of the containing blocks for b, so it recurses on Get-Grouping(Parent-Block$(b), gs)$ until $gs[b] = root$. This way, Get-Grouping ensures a consistent tree structure.

What is left is to ensure that the current GroupingNode, $gs[b]$, represents the correct iteration of the loop represented by b, and then to add the invocation c to the group. First, if the GroupingNode $gs[b]$ has no invocations in it (line 10), then it must be on the first iteration, and c can be added to it (line 12). Otherwise, we compare line numbers and associated AST positions to check if the trace is looping. The basic idea is this: if the current invocation c precedes the last invocation l (line 13) in the syntactic structure of the source code, and both are in the group $gs[b]$ (line 9), then the trace has begun

Algorithm Group-Invocations

Require: A traced Invocation i
Require: A Statement a that defines i
Ensure: A GroupingNode tree structure as output

```
1:  Let m be an associative array of Invocations to Statements
2:  Let gs be an associative array of Statements to GroupingNodes
3:  Create a new, empty GroupingNode, root to represent the root of the node tree
4:  m[i] ← a
5:  gs[a] ← root
6:  for all c of i.children do
7:      m[c] ← Associated-Statement(a, c)
8:      b ← Parent-Block(m[c])
9:      gs[b] ← Get-Grouping(b, gs)
10:     if gs[b].Empty() then
11:         gs[b].Add(c)
12:     else
13:         l ← Last-Invocation(gs[b])
14:         if l.line <= c.line then
15:             if m[l].Is-Child-Of(m[c]) then
16:                 m[l] is expected to be later than m[c] according to syntax... continue
17:             else if m[c].Is-Child-Of(m[l]) then
18:                 if m[l].startChar <= m[c].startChar then
19:                     gs[b] ← Next-Iteration(b, gs)
20:                 end if
21:             else if m[l].startChar >= m[c].startChar then
22:                 gs[b] ← Next-Iteration(b, gs)
23:             end if
24:         else
25:             gs[b] ← Next-Iteration(b, gs)
26:         end if
27:         gs[b].Add(c)
28:     end if
29: end for
30: return root
```

Fig. 3. Grouping invocations into loops

a new iteration of a loop, and the groups must be updated.

Normally, one would assume that if the line for l is less than the line for c, then the program is progressing without looping. This normal case of loop iteration is dealt with on line 25. However, there are exceptions to this rule. Lines 14 through 23 take care of the exceptions.

If the statement associated with the last invocation is a child of the statement for the current invocation then the relationship is reversed. For example, `foo(bar())` will call `bar`, then `foo`, but `foo` comes before `bar` in the source code. So, line 15 checks for this condition and allows the algorithm to proceed without creating a new iteration for the loop.

The algorithm also has to deal with complications that occur when it encounters loops that only span a single line. Lines 21 through 23 deal with the normal case by comparing the location of the starting character for the statements associated with the last and current invocations. For example, given a loop that contains only the single line `foo(); bar();`, each iteration of the loop would begin with `foo`. It is on the same line as `bar` but with a lower character index.

Once again, if there is a parent-child relationship, then the character ordering in source code is reversed. Lines 18 through 20 take care of such instances of single-line loops. Lines 18 through 20 also handles an exception that can occur in some object-oriented languages in which a method may be called on the return value of another method. For example in `foo().bar()`, `foo` gets called before `bar`, but in the AST, `foo` is a child of `foo().bar()`, and they both have the same start position. So, when a loop occurs on a single line, the relationship is similar to that of `foo(bar())`.

The above discussion accounts for parent-child relationships for invocation statements that occur on a single line. Similar checks as above can be added to line 25 of the algorithm to handle cases that are broken across multiple lines.

The last detail to explain is the functioning of the Next-Iteration sub-algorithm. Next-Iteration updates the Grouping-Node tree structure when a loop occurs. First the current iteration for the GroupingNode in $gs[b]$ must be stored. We call this iteration it. The associative array gs is then cleared for the AST statement b and all of its children. Next, Get-Grouping is called for b and gs. Since gs has been cleared, this will force Get-Grouping to create a new tree structure for the AST of b which will hold all of the invocations for the next iteration of the loop. $gs[b]$ is set to this new grouping, and its iteration number is set to $it+1$. The new $gs[b]$ is returned, which updates the tree for the next loop. The algorithm runs on one invocation at a time and can be run iteratively over each invocation as required.

C. Caveats

There are a number of instances in which the algorithm will fail. First, it assumes that invocations in the trace can always be matched with statements in source code (line 7). This is not the case for some programming languages. For example, Java will insert method invocations that load classes from disk the first time that they are referenced in the program. Such invocations are typically side effects of invocations that are present in the source code, and the problem can be solved by associating invocations with their side effects. These cases are language dependent and must be dealt with on a language to language basis.

Second, the sub-algorithm Associated-Statement cannot guarantee that it returns the correct statement in the case that multiple invocations with the same signature occur on the same line of code. It is not possible to distinguish the two invocations because the only data given in the input for the invocation is the signature and the line number. This can be a problem in examples such as single-line loops. In such a case the algorithm may count more iterations than there actually are, though invocations will still be associated with the correct code block. Standard coding conventions will normally eliminate this problem.

Finally, the algorithm assumes that invocations occur in the same order in the trace as they do in the source code. This will not always be the case with optimizing compilers which may reorder invocations in order to speed up processing. Compiler optimizations should be turned off to fully benefit from this algorithm.

D. Extensions

The algorithm presented is not limited to just detecting loops. It can also be used to detect other block statements such as conditional or error handling blocks. In these cases, the blocks are treated like loops that execute only once. They can then be presented in a user interface to enrich the visualizations with more semantic information related to the source code.

IV. Experiment

The goal of this algorithm is to locate portions of execution traces that are repetitious due to loops and compact them for visualization purposes. It is impossible to formally prove the effectiveness of the algorithm as the amount that the algorithm can compact a trace visualization will depend on the source code and traces that are given as input. Source code varies widely between projects. Therefore, we chose to perform an experiment that will demonstrate the level of compaction that this approach can achieve. The following subsections describe the experiment design and results.

A. Experiment Design

The algorithm described in this paper has been implemented in our Open Source research project Diveri. Diveri offers the ability to capture, visualize and compare dynamic execution traces within the Eclipse IDE. It can also enhance static structure views such as source code editors and the Java package explorer based on the contents of the dynamic trace. We used the implementation in Diveri to evaluate the algorithm's ability to compact execution traces.

The Diveri implementation of the algorithm works on an AST for the Java programming language. It is able to recognize method calls and constructor invocations and associate

them with their containing loops. All Java 6 loops are recognized including the `for`, `while`, `do...while`, and the extended `for` (otherwise known as the `foreach` loop).

We tested Diveɔɪ's implementation of the algorithm against three industrial Open Source Java applications and utilities: the Eclipse IDE 3.5R1 [19], the HSQLDB database engine 1.8.1 [22], and the Jetty web server platform 6.1.21 [20]. These three applications were chosen because they are high-quality software products that are popular within the software industry, and their source code is readily available. They also support a wide variety of different computer usage scenarios.

Diveɔɪ was used to generate traces for each of the programs. Since each program is designed to supply very different functionality from the others, a use case had to be chosen for each.

Eclipse. Eclipse was chosen because it is an interactive graphical application, and it supplies functionality for many typical computer usage scenarios (graphical interaction, document editing, file manipulation, etc.). The use case for this experiment was that of opening a project in the Eclipse workspace. This use case was chosen because it is a simple operation that is typical of normal Eclipse usage, and it exercises many portions of Eclipse. Diveɔɪ allows users to pause and restart traces at any time. In order to trace information related to opening the project, a break-point was set within the `open` method of the `Project` class of the Eclipse Resources plug-in. When the breakpoint was hit, the Diveɔɪ tracer was started, and Eclipse was allowed to run until the user interface indicated that the project had been opened, at which point the Diveɔɪ trace was paused, and Eclipse was closed down. Note that Diveɔɪ captures data from all threads as long as the trace is running, so any operation that occurred while the project was being opened was recorded regardless of whether it was directly related to the opening of the project. This is not considered a defect in the experiment, as it is part of the normal operation of Eclipse.

HSQLDB. HSQLDB is a good use case because it exercises low-level data and disk manipulation. To gather data for HSQLDB, the database was configured to run in embedded mode within the Java virtual machine. A program was created that used the Java Database Connectivity drivers for HSQLDB to perform a simple select on a pre-populated table. After the selection, the program iterated through all of the results of the query and cached them into a local data structure. This program exercises the functionality that is typically used in database systems by loading tables, retrieving data, and storing that data for manipulation in the program. Once again, Diveɔɪ was used for the trace, and was left paused until a breakpoint was hit immediately before the execution of the query. The trace was then started, and the program was allowed to run until all of the results had been cached, at which point the trace was paused and the program was exited.

Jetty. Jetty was chosen as a use case because it represents another very common instance of computer usage. It is a multi-threaded HTTP server that hosts web pages for communication over network connections. Diveɔɪ was configured to trace from the beginning of the Jetty program in order to record the server start-up process. One page request was also made for the Jetty default test page by using a web browser on the local host. Once the page was loaded into the browser, Jetty was shut down and the trace ended. This usage was chosen to, once again, exercise the common functionality of the software as it loaded, and processed communication over the network.

For all three cases, Diveɔɪ was configured to record all method and constructor invocations and store them into a local database. For the purpose of this discussion, we will call the set of all recorded invocations in a use case **I**. For each case, 400 method or constructor invocations were selected using a random procedure. We will call this set **R**. Invocations were only considered if they, in turn, caused at least 15 more invocations. This was done in order to rule out parts of the call tree that would not cause significant stress on a visualization whether compacted or not. Each of the invocations were tested to discover how many method invocations would need to be displayed if not compacted and how many would need to be displayed if compacted. The following definitions were used.

Definition 1 For any $i \in \mathbf{I}$ the set $\mathbf{S}(i) = \{s \in \mathbf{I} \mid s$ is invoked by $i\}$ is called the sub-invocations of i.

Definition 2 For any $i \in \mathbf{I}$ the set $\mathbf{C}(i) = \{c \in \mathbf{S}(i) \mid c$ is (1) not contained in any loop or (2) contained in only the first iteration of any loop$\}$ is called the compaction of i.

What this means is that to compact a set of sub-invocations of i, we made sure that the selected sub-invocations were reachable within the first iteration of any nested loops. In visualization terms, this means, "only display the first iteration of any loop." For example, given an invocation i of the pseudo code in Figure 4, $\mathbf{S}(i) = \{a_{(1)}, b_{(1,1)}, b_{(1,2)}, c_{(1,2)}, a_{(2)}, b_{(2,1)}, b_{(2,2)}, c_{(2,2)}\}$ and $\mathbf{C}(i) = \{a_{(1)}, b_{(1,1)}\}$ (given in order of invocation). Note that no invocation of c occurs in $\mathbf{C}(i)$ because it never appears in the first iteration of the nested loop.

Algorithm Loops Example

```
for l ← 1 to 2 do
   invoke a_(l)
   for m ← 1 to 2 do
      invoke b_(l,m)
      if m = 2 then
         invoke c_(l,m)
      end if
   end for
end for
```

Fig. 4. An example of nested loops

These rules were used to ensure that every invocation was reachable through a simple series of iterations. Different results would have arisen for different iterations of the loops. However, loops are repetitive by nature, so the results would not be expected to be significantly different.

B. Results

After the data was collected for the three cases, we took several measurements to help indicate the level of compaction that this algorithm enables. As stated earlier, a random subset, $\mathbf{R}$, of 400 invocations were taken from the set of all recorded invocations. The data collected is summarized in Table I. The measurements we took are defined as follows.

- **Total Invocations** $|\mathbf{I}|$: the total number of calls made during the experiment
- **Percent of Invocations Measured** $\%|\mathbf{I}|$: since a random sample of 400 invocations were recorded (i.e. $|\mathbf{R}| = 400$), this is defined as $(400/|\mathbf{I}|) \cdot 100\%$
- **Total Sub-Invocations** $\sum |\mathbf{S}(i)|$: the total number of sub-invocations measured, defined as $\sum_{i \in \mathbf{R}} |\mathbf{S}(i)|$
- **Total Compaction** $\sum |\mathbf{C}(i)|$: the total number of invocations in all sets $\mathbf{C}(i)$, defined as $\sum_{i \in \mathbf{R}} |\mathbf{C}(i)|$
- **Overall Compaction Ratio** $Compact(\mathbf{R})$: the ratio of compacted invocations to un-compacted invocations over all $i \in \mathbf{R}$. Simply the ratio between the previous two measurements
- **Average Compaction Ratio** $\overline{Compact(\mathbf{R})}$: the average of the sum of the compaction ratios for each $i \in \mathbf{R}$, defined as $\frac{\sum_{i \in \mathbf{R}} |\mathbf{C}(i)|/|\mathbf{S}(i)|}{|\mathbf{R}|}$. This is the amount that the sub-invocations for a single invocation are compacted, on average
- **Number of Loops** $|\mathbf{L}|$: the total number of invocations in $\mathbf{R}$ that contain loops. In other words the set $\mathbf{L} = \{i \in \mathbf{R} \mid i \text{ contains at least one loop}\}$

As can be seen in Table I, each of the traces recorded more than 2 000 000 invocations. Of the recorded invocations, 400 (approximately $.02\%$) were sampled for the experiment in each case.

Both Jetty and Eclipse yielded similar results. To ensure that the samples were representative of the population, we performed a normal probability plot over the average compaction ratio of the sampled data and found that they both followed a normal distribution. We can say with confidence that our samples are representative. In terms of overall compaction, Eclipse had a compaction ratio of .186 and Jetty had a compaction ratio of .149. This means that the compacted sample in the Jetty trace is 6.7 times smaller than the un-compacted one, in terms of sub-invocations. Similarly, the compacted sample in the Eclipse trace is 5.38 times smaller. In other words, if we considered a visualization that displayed all sub invocations of the 400 invocations sampled in Jetty, 85.1% need not be displayed due to looping.

The average compaction ratio indicates how much an individual invocation is compacted on average. Eclipse's average compaction ratio is similar to its overall compaction ratio (.218 versus .186). This indicates that the number and size of the loops in this trace of Eclipse are relatively well distributed and equal. In fact, there was only one invocation that could not be compacted due to a lack of loops in the code. Contrasting this result to Jetty, we see that the difference between the overall compaction ratio and the average compaction ratio is quite large (.149 versus .357). This indicates that there are several "dominator" invocations which allow for a high compaction. These dominators are invocations that contain small loops that iterate many times. The average case, though, is a larger invocation that contain loops without many iterations, or no loops at all. It can be seen that 82 of the 400 invocations (20.5%) did not contain any loops. Nonetheless, the average invocation is compacted by 74.3%.

HSQLDB yielded very different results than the other two cases. Of the 400 sampled invocations, none of them contained loops, so no compaction was possible. By inspecting the sum of all sub-invocations $\sum |\mathbf{S}(i)|$, we can gain more insight into why this is. For HSQLDB $\sum |\mathbf{S}(i)|$ is only 6 824 meaning that the average invocation in the sample set causes only 17.06 sub-invocations. This indicates that most of the calls in HSQLDB are low-level calls that are likely uninteresting. The trace for HSQLDB was further investigated to see what may be the cause of these low-level calls. We found that 387 of the 400 calls (96.8%) were in fact different invocations of the same method. The probability of 387 of the same invocations occurring in a sampling of a random set of more than 2 000 000 invocations is nearly 0. Therefore, the invocations that do not contain loops must dominate those few invocations that do. While there may be several invocations that could be compacted to a high degree, Table I is representative of the HSQLDB system.

C. Time Analysis

The run-time of this algorithm is also an important consideration. The compaction ratios may be high, but if the algorithm takes too long to run, users will wait too long to get their results, and the application will have failed to be an efficient aid to their reverse engineering tasks.

Theoretical analysis is not a practical approach. The algorithm delegates the work of parsing files and searching for matching code elements (such as the method definition for an invocation) to different processes. In fact, the implementation in Div$_{9\text{I}}$ delegates these tasks to the Eclipse Java Development Tools. These tasks are not trivial, but a time-analysis is not available for them. It is therefore very difficult to mathematically prove the run-time of the algorithm.

Instead, we opted for an experimental approach to evaluate the efficiency of the algorithm. We re-ran the same experiment as before except that our data point was the average amount of time it took for the algorithm to run. This included the times required to parse the file into an abstract syntax tree, to associate invocations with corresponding elements in the abstract syntax tree, and to detect the loops. This experiment was performed on a machine, with a 2.21 Ghz Dual Core processor and 2GB of RAM. The results are listed in Table II, rounded to the nearest millisecond.

The worst performer was Jetty at 167ms per run of the algorithm. This is approximately 6 executions per second. It is slow enough that it is not instantaneous, but it is more than fast enough for an interactive application. The speed could be improved significantly by caching the abstract syntax trees,

TABLE I
THE RESULTS OF THE THREE CASES

Case	$\|\mathbf{I}\|$	$\%\|\mathbf{I}\|$	$\sum \|\mathbf{S}(i)\|$	$\sum \|\mathbf{C}(i)\|$	$Compact(\mathbf{R})$	$\overline{Compact(\mathbf{R})}$	$\|\mathbf{L}\|$
Eclipse	2 214 478	0.018%	15 826	2 948	0.186	0.218	399
Jetty	2 304 848	0.017%	31 398	4 673	0.149	0.357	318
HSQLDB	2 333 381	0.017%	6 824	6 824	1.0	1.0	0

TABLE II
THE AVERAGE ALGORITHM RUN-TIME

	Eclipse	Jetty	HSQLDB	Mean
Time (ms)	112	167	33	104

and the associations with invocations. However, such a move would also put significant stress on system memory resources when considering large traces. The speed of the algorithm is fast enough as-is, and the memory trade-off is likely not worth the extra milliseconds for most applications.

D. Threats To Validity

The largest threats to the validity of the experiment are the small sample sizes, both in terms of the range of software tested and the small sample taken from each of the traces.

In regard to the range of the software tested, we note that all three applications are highly used and of industrial quality. The three applications are representative of a large range of common computing tasks.

To address the small samples taken from the traces (less than .02% of each trace), we performed some statistical analysis. However, it was found that both the Eclipse and Jetty measurements followed a normal distribution, so the sample size does not depend on the population size. The HSQLDB example was not a normal distribution, but the probability of the sampled invocations not being representative of the system was shown to be very low. It is not likely that more insight would have been gained from a larger sample size.

One final threat is that the algorithm was not correctly implemented. This threat has been mitigated through months of testing and inspection, though bugs may still be present.

V. APPLICATIONS

So far, we have discussed the compaction algorithm without any reference to its applications. In this section, we show how it has been utilized in the Open Source project Diviz. Diviz uses the algorithm to compact a view that is based on UML sequence diagrams. The view is modeled after our previous work [1] and Sharp and Rountev [16]. Specifically, it uses the concept of "combined fragments" to group together sequences of messages that are related in some way. Diviz uses the algorithm given in this paper to discover combined fragments based on source code. Standard UML has a limited vocabulary for combined fragments, including labels such as "loop" and "alt". Diviz extends this vocabulary by using the text of the statement that defines the block.

Figure 5 gives a simple example of the power of the algorithm to simplify trace exploration views. A small program was written which uses a `for` loop in Java to call two methods 100 times. Figure 5(a) shows the resulting trace without loop compaction, and (b) shows it with compaction. It can be seen that for even very simple programs, the compacted view is much simpler and easier to read. The label for the combined fragment also displays semantic information about the loop that it represents.

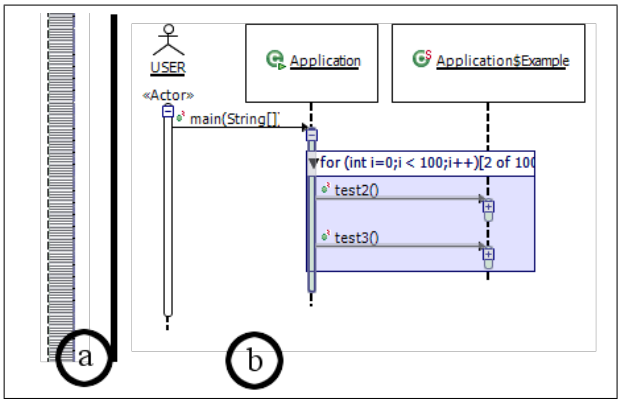

Fig. 5. (a) A sequence diagram zoomed to fit (b) the same sequence diagram compacted using our algorithm

When interacting with execution traces, it is not enough to hide iterations of loops. Different iterations can have different side effects, and so it must be possible to inspect them. Diviz offers the ability to interactively select which iteration to view via a pop-up menu (Figure 6). This allows users to check individual loop iterations to find whether various iterations cause different invocations.

Compacting loops hides information about the running program. Specifically, it is not always possible to see in the diagram if a method has been called in a trace or in what context it was called. Consequently, Diviz offers ways to expose method invocations by using the static source code views offered by Eclipse. Since Diviz must be able to pair invocations with source code in order to organize them in its sequence diagram view, the pairings can also be used in static views to reveal dynamic behavior. Particularly, Diviz extends the Eclipse Package Explorer with a *Reveal In...* action which can be used to reveal the first invocation of a method in a particular thread using the sequence diagram view. When the user selects the thread in which to view an invocation, Diviz automatically expands, scrolls, and swaps loop iterations in

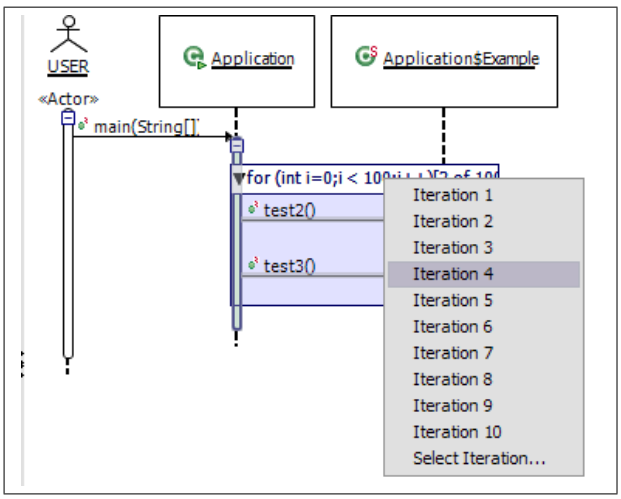

Fig. 6. Selecting different iterations in the sequence diagram

the sequence diagram view to reveal the context of the first invocation of the selected method.

Revealing an invocation from the Package Explorer creates a link between the static source code and the dynamic execution trace but in a limited way. Methods are likely invoked many times within an execution of a program, but methods are only represented once in the Package Explorer. To help solve this problem, Diveɔɪ also offers a time line which shows the various invocations of a selected method over the span of the execution trace. Invocations are shown as small vertical strips in the time line. Users can right-click on a strip and either reveal the invocation in the sequence diagram, or focus on it (Figure 7). Revealing expands and scrolls the view to show the invocation. Focusing roots the diagram on the selected invocation, creating a slice of the call tree. In both cases, loop iterations are swapped automatically to ensure that the invocation can be shown in context.

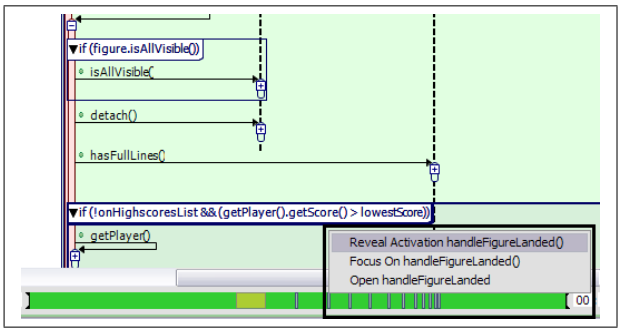

Fig. 7. Selecting invocations of a method using a time line

One final pertinent feature of Diveɔɪ is its implementation of the extension to the algorithm described in section III-D. Diveɔɪ is able to compact loops using the algorithm as well as create combined fragments for other Java code blocks. Diveɔɪ supports conditional blocks (`if` and `switch`) and error handling blocks (`try` and `catch`).

Figure 8 illustrates combined fragments for Java code blocks. It shows a `try` block, a `for` loop, an `if` block, and a `catch` block. Beyond the fact that the algorithm is able to compact the loops in the diagram (only 1 of 31 loops is shown), this figure also demonstrates the expressive power of the algorithm when used in visualizations. By inspecting the figure we can see not only what happened, but we are also able to see some information about why it happened. In this example, it can be seen that on the 31^{st} iteration of a `for` loop, which is counting down, the method `checkRange` is called on the `TestUtilities` class. Inside an `if` block, the condition `number <= 0` evaluates as true, so a new `IllegalArgumentException` is created, and the method immediately returns (indicating that the exception was thrown). Finally, we can see that the exception is caught in the `catch` block following the `for` loop. This can all be seen in a compact way on the sequence diagram without the need to navigate through source code which would typically require inspecting several different source files.

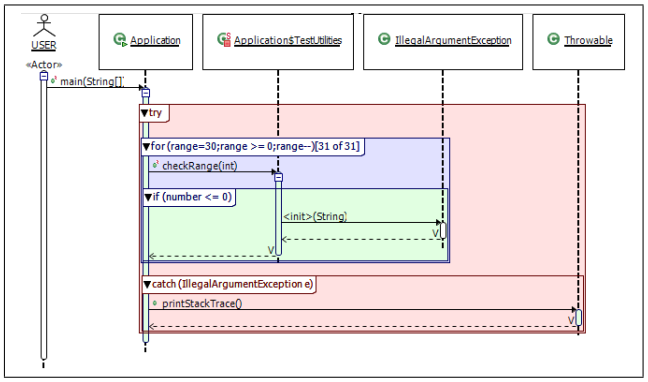

Fig. 8. Conditional and error handling blocks discovered by an extension to the algorithm

Besides visualization, there are other applications to the given algorithm as well. One is that it offers a fine granularity for profiling applications. Many profilers offer the ability to locate "hot spots" in code by indicating methods that are called often or that take a long time to process. This algorithm could give a finer granularity to indicate the amount of time spent in a loop or conditional block to help find bugs or possible candidates for optimization.

VI. Limitations

The algorithm in this paper suffers from limitations that make it unsuitable for some applications.

The first of the limitations arises from the fact that the algorithm requires data from a number of different sources including dynamic traces, source code, and debug information. If the source code or debug information is missing, this algorithm can not be used. In such cases, other methods such as the one given by Lui *et al.* [12] may be appropriate.

Using debug information introduces another limitation in that it is often computationally expensive to query a machine for the line number of execution. This increases the impact of

the tracing on the running application, so the algorithm is not suited to reverse engineering time critical systems.

Section II indicates that the algorithm is fast enough for interactive applications. However, it may be too slow as a pre-processor on very large execution traces. It is best applied in an iterative, interactive environment or on traces that have already been trimmed using other filtering mechanisms.

Finally, the HSQLDB example indicates that the algorithm is not suitable for applications that make heavy usage of low-level code. For this situation, the summarization method of Hamouh-Lhadj and Lethbridge [8] may be a better approach.

VII. Conclusions and Future Work

This paper has presented a method for linking source code and debug information with dynamic execution traces. The resulting algorithm is able to compact execution traces significantly. This greatly reduces the amount of information that needs to be displayed in a visualization. We conducted an experiment to demonstrate that the algorithm can be applied to large scale industrial software. The experiment also demonstrated that the algorithm is practical for interactive applications.

The utility of the algorithm was demonstrated in the example application Div$_{\text{ƨɪ}}$. Div$_{\text{ƨɪ}}$ uses the algorithm to display invocations within the context of the blocks of code in which they occur by using combined fragments in sequence diagrams. Different iterations of loops can be viewed, one at a time, by swapping them in the diagram. It is also possible to reveal method invocations by using static views of source code in Div$_{\text{ƨɪ}}$. In these cases, iterations are swapped implicitly as the invocations are revealed.

There is still more work to be done in this area of research. One obvious topic is to address the limitations of the algorithm. For example, when source code is unavailable, it may be useful to incorporate other compaction techniques which do not rely on source code. Also, the algorithm does not offer a lot of compaction in applications that make heavy use of low-level code. Augmenting the compaction by incorporating techniques such as the comprehension units of Hamou-Lhadj and Lethbridge [8] may help in such cases.

Finally, although the experiment in this paper demonstrates that utilizing source code and debug information makes it possible to gain high compaction ratios in execution traces, it has not been determined that the compaction will actually help users of trace visualizations. Our own usage of the Div$_{\text{ƨɪ}}$ tool has given us the insight that they do help. Still, user studies should be done to test the technique both alone and together with the other techniques developed by other researchers.

Acknowledgment

We would like to thank Philippe Charland, and David Ouellet of Defence Research and Development Canada for their participation, Tricia Pelz and Nick Matthijssen for their editorial work, and Peter Rigby for his analytical help. This research has been funded by Defence Research and Development Canada under contract W7701-82702/001/QCA.

References

[1] C. Bennett, D. Myers, M. Storey, D. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, pp. 291-315, 2008.

[2] J. Bohnet, M. Koeleman, and J. Doellner, "Visualizing massively pruned execution traces to facilitate trace exploration," in *IEEE Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 2009.

[3] L. Briand, Y. Labiche, and J. Leduc, "Toward the reverse engineering of uml sequence diagrams for distributed java software," *IEEE Transactions on Software Engineering*, vol. 32, pp. 642–663, Sept. 2006.

[4] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen, "Understanding execution traces using massive sequence and circular bundle views," in *ICPC '07: Proc. of the IEEE Int'l Conf. on Program Comprehension*, (Washington, DC, USA), pp. 49–58, IEEE, 2007.

[5] Y. Guéhéneuc and T. Ziadi, "Automated reverse-engineering of UML v2.0 dynamic models," in *Proc, of the ECOOP Workshop on Object-Oriented Reengineering*, 2005.

[6] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proc. of the Conf. of the Centre for Advanced Studies on Collaborative Research*, pp. 42–55, IBM Press, 2004.

[7] A. Hamou-Lhadj and T. Lethbridge, "Measuring various properties of execution traces to help build better trace analysis tools," in *Proc. of the IEEE Int'l Conf. on Engineering of Complex Computer Systems*, pp. 559–568, June 2005.

[8] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Proc. of the IEEE Int'l Conf. on Program Comprehension*, pp. 181–190, IEEE 2006.

[9] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing interactions in program executions," in *ICSE '97: Proc. of the 19th Int'l Conf. on Software Engineering*, (New York, NY, USA), pp. 360–370, ACM, 1997.

[10] A. Ketterlin and P. Clauss, "Prediction and trace compression of data access addresses through nested loop recognition," in *Proc. of the IEEE/ACM Int'l Symp. on Code Generation and Optimization*, (New York, NY, USA), pp. 94–103, ACM, 2008.

[11] D. Kimelman, B. Rosenburg, and T. Roth, "Strata-various: multi-layer visualization of dynamics in software system behavior," in *Proc. of the IEEE Conference on Visualization*, pp. 172–178, IEEE, Oct. 1994.

[12] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting duplications in sequence diagrams based on suffix trees," in *Proc. of the Asia Pacific Confrence on Software Engineering Conference*, pp. 269–276, Dec. 2006.

[13] MaintainJ Inc., "Maintainj - reverse engineer java like never before." Web page on-line http://www.maintainj.com. Cited on 16 Sept. 2009.

[14] H. Müller, K. Wong, and S. Tilley, "Understanding software systems using reverse engineering technology," in *Proc. of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences*, 1994.

[15] H. Safyallah and K. Sartipi, "Dynamic analysis of software systems using execution pattern mining," in *Proc. of the IEEE Int'l Conf. on Program Comprehension*, IEEE, 2006.

[16] R. Sharp and A. Rountev, "Interactive exploration of UML sequence diagrams," in *IEEE Workshop on Visualizing Software for Understanding and Analysis*, pp. 8–13, IEEE, 2005.

[17] T. Systä, "On the relationships between static and dynamic models in reverse engineering java software," in *Proc. of the Working Conference on Reverse Engineering*, pp. 304–313, Oct 1999.

[18] J. Seward, "bzip2: Home." Web page on-line http://www.bzip.org. Cited on 19 Oct. 2009.

[19] The Eclipse Foundation, "Eclipse.org home." Web page on-line http://www.eclipse.org. Cited on 13 Oct. 2009.

[20] The Eclipse Foundation, "Jetty." Web page on-line http://www.eclipse.org/jetty/. Cited on 13 Oct. 2009.

[21] The Eclipse Foundation, "Using uml2 trace interaction views." Documentation on-line http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.tptp.platform.doc.user/tasks/tesqanac.xhtml. Cited on 22 Sept. 2009.

[22] The HSQL Development Group, "Hsqldb." Web page on-line http://hsqldb.org. Cited on 13 Oct. 2009.