Towards Anomaly Comprehension: Using Structural Compression to Navigate Profiling Call-Trees

Shen Lin, François Taïani, Thomas C. Ormerod, Linden J. Ball
Lancaster University, Lancaster, UK
{s.lin6, f.taiani, t.ormerod, l.ball}@lancaster.ac.uk

ABSTRACT

Developers must often diagnose anomalies in programs they only have a partial knowledge of. As a result, they must simultaneously reverse engineer parts of the system they are unfamiliar with while interpreting dynamic observation data (performance profiling traces, error-propagation channels, memory leaks), a task particularly difficult. To support developers in this kind of comprehension task, filtering and aggregation have long been suggested as key enabling strategies. Unfortunately, traditional approaches typically only provide a uniform level of aggregation, thus limiting the ability of developers to construct context-dependent representations of a program’s execution. In this paper, we propose a localised approach to navigate and analyse the CPU usage of little-known programs and libraries. Our method exploits the structural information present in profiling call trees to selectively raise or lower the local abstraction level of the performance data. We explain the formalism underpinning our approach, describe a prototype, and present a preliminary user study that shows our tool has the potential to complement more traditional navigation approaches.

Categories and Subject Descriptors
B.8.2 [Performance and reliability]: Performance Analysis and Design Aids

General Terms
Performance, Human Factors

Keywords
program comprehension, performance profiling

1. INTRODUCTION

Modern software applications increasing rely on a complex ecosystem of third party components that considerably hinder the diagnosis of anomalies in program behaviour (instability, low performance, memory leaks). Modern software applications are typically made of parts developed by distinct teams in independent organisations, are continuously expanded and corrected, and resemble a living organism, constantly evolving in a loosely controlled manner.

This constant evolution and somewhat organically grown structures mean that non-functional properties (performance, security, dependability) are often poorly understood. To face this challenge, and improve the overall quality of their products, developers need tools and techniques that help them understand the non-functional behaviour of their platforms. Typical non-functional properties are unfortunately systemic properties that require both a global and detailed understanding of a system’s operations. Large software products are collaboratively developed; they integrate numerous third party components; and as a result no developer can claim to thoroughly understand them. Because current non-functional analysis techniques typically require a good prior understanding of their target system, developers are in need of powerful filtering and aggregation techniques, two strategies that have long been recognised as key enablers for program comprehension. Unfortunately, traditional approaches typically only provide a uniform level of aggregation, thus limiting the ability of developers to construct context-dependent representations of a program’s execution. In this paper, we focus more particularly on the analysis of dynamic CPU usage traces, and propose a localised approach to interactively navigate the CPU usage of unfamiliar programs and libraries.

Our method exploits the structural information present in profiling call trees to selectively raise or lower the local abstraction level of the performance data. Our approach builds on prior works that combine static and dynamic software data [16, 12, 6, 5, 21], and exploits the structural information contained in dynamic profiling traces to reduce the amount of information presented to users while retaining a systemic overview of performance phenomena.

In this paper, we present the rationale for our approach (Section 2), explain its underlying intuition (Section 3.1), provide a graph-based formalisation of its workings (Section 3.2), describe an exploratory prototype (Section 3.3), and report on a small user study to assess its benefits and challenges (Section 5). Finally we present related work (Section 6) and conclude (Section 7).

2. PROBLEM STATEMENT

As development cycles shorten, complex applications are increasingly developed using off-the-shelf components [13]. Although the functional behaviour of these components are
slowly because a given component's deployment. CPU consumption is a good representative example of this situation: a system might be executing components, and are thus hard to predict before the system's deployment. CPU consumption is a good representative example of this situation: a system might be executing slowly because a given component A is being used by another component B under an adverse workload that A was not designed to handle.

Analysing and diagnosing such situations is not trivial. It often involves a combination of black-box and grey-box profiling, the second approach providing finer-grained details of the software's behaviour. Among grey-box techniques, sample-based profiling is particularly popular due to its limited level of interference. A sampled-based profiling tool periodically interrupts an application and captures the state of the currently active thread. This state may be limited to the currently executing function, or might include additional context information, from the direct caller (as in gprof [7]), up to the full call path from the thread's starting point (e.g. hprof [10] or STAT from the Paradyn project [4]). When full call paths are captured, the result is a set of weighted stack traces that reflect the application’s CPU usage. A particularly active part of the code will appear proportionally often in the stack traces, thus allowing developer to track hot-spots. The traces themselves document the sequence of nested calls that lead to some code being executed, thus helping with diagnosis.

Figure 1 shows an example of hprof output for an hypothetical biology simulation program. Three traces have been observed: Trace 1 three times, Trace 2 once. Using a simple prefix-tree (trie) algorithm, this weighted set of traces can be transformed in a weighted call tree (Figure 2). Weights are indicated in square brackets, and reflected in the node sizes. From the profiling tree we can infer that lib2.Muscle.contract() uses twice as many CPU cycles (× 4 samples) as lib2.Muscle.stops() (× 2), essentially because lib2.Muscle.contract() uses lib3 twice as much as lib2.Muscle.stops().

On non-trivial systems, however, this profiling tree can become quite large, even on limited experiments. For instance a simple distributed scenario on the Grid Computing platform Globus [19] can result in a profiling tree with more than 1000 nodes. To help developers navigate such large trees, industrial analysis tools (e.g. HPjmeter [3] or Eclipse TPTP [1]) typically offer an interface such as the one of Figures 3 and 4 that allows developers to explore a profiling tree by expanding or collapsing tree branches.

Although useful, these approaches largely ignore concomitant issues of program comprehension that arise in larger and more complex cases such as that of Figure 4, showing a Globus execution. They assume developers have a reasonable command of the programs they analyse and can provide the structural and behavioural models that are needed to make sense of the data. Unfortunately, in larger and more complex systems, individual developers often only have a partial knowledge of the various parts of the software and must therefore simultaneously reverse engineer the parts of the system they are unfamiliar with while diagnosing performance issues. To support this kind of comprehension task, filtering and aggregation have long been recognised as key enabling strategies to analyse software data. Unfortunately, traditional approaches typically only provide a uniform level of aggregation, in which one code entities (a method, class or
package) appear at the same level of abstraction throughout the representation. This is problematic for highly context-dependent performance data, and limits the ability of developers to construct context-dependent representations of a program’s execution. To address this challenge, we propose a localised approach to navigate and analyse the CPU usage of little-known programs and libraries. Our method exploits the structural information present in profiling call trees to selectively raise or lower the local abstraction level of the performance data.

Our approach builds on prior software analysis techniques that combine static and dynamic data [16, 12, 6, 5, 21]. It differs from these works, however, in that it is interactive (users do not need to provide a prior specification of compaction rules as in BLOOM or AVID [16, 21]), and localised (users can apply different levels of compaction to the same program elements in different parts of the profiling tree, contrarily for instance to the work of Cornelissen et al [5]).

3. APPROACH

3.1 Intuition

The traditional approach for navigating a profiling tree such as that of Figure 2 consists in selectively hiding or showing subtrees. The represented information remains however at the same level of abstraction: each node corresponds to the invocation a method along a particular call path starting at the tree’s root.

In this paper we propose to explore an alternative approach by varying the level of abstractions at which different parts of the profiling tree are represented. As in previous related works [16, 12, 6, 5], our premise naturally unfolds from an intuitive understanding of classes and objects as interactive entities: When a method lib2.Nerve.transmit() calls another method lib3.Signal.travel(), we can understand the same interaction as the class lib2.Nerve calling the class lib3.Signal (Figure 5). Exploiting the organisation in packages used by Java classes, we can push this analogy further, by considering the same call to be an invocation from the package lib2 to lib3 (Figure 6). Because Java packages in concrete applications are usually nested, this process is recursive, and a call from java.security.AccessController.doPrivileged() to org.apache.axis.utils.ClassUtils.forName() can then be seen as a call from java to org.apache.

The two previous examples of Figure 7 and 6 are however uniformly compacted. All nodes are represented at the same level of abstraction: that of classes for Figure 7 and top-level package for Figure 6. Developers might however wish to zoom-in by lowering the abstraction of one particular part of the graph, while maintaining the rest of the graph in its compacted form. Figure 7 shows such an example, where the classes of lib2 are shown (Lung, Muscle, Nerve), but the other packages (lib1 and lib3) kept fully compacted. Our technique further extends this approach by allowing users to select local levels of abstraction that only apply in one part of the profiling tree. As a result, the same program element might be expanded at different granularity levels in different parts of the graph. For instance, Figure 8 shows how the right-hand side lib3 package is locally expanded, while the same left-hand side package remains compacted.

3.2 Formalisation

Two elements are required to construct the kind of locally compacted profiling trees we have just sketched:

- The ability to specify the local compaction level that should apply for a particular package in a particular area of the profiling tree.
A localised merging mechanism that captures the interplay of both structural and behavioural closeness to determine the final abstraction level of each program execution points.

To address the first point, we associate each node with a granularity level, an integer that represents how much of the node’s full name should be represented in the rendered tree. For instance, Figure 9 shows the value of granularity levels (in circles) leading to the compaction of lib2 in a single node (Figure 10). The granularity level of a node determines its compacted name (essentially a prefix of its full name), by indicating how many elements of the node’s name should be retained in the final graph. For instance in Figure 9, node `lib2.Lung.inhale()` has a granularity level of 1, meaning that it should merge with nodes in its vicinity (essentially descendants or siblings) that also belong to lib2 (represented by the ‘lib2’ set of nodes on Figure 9). The resulting compacted node will then be represented by its top-level package `lib2` (in bold) in the compacted tree (Figure 10). All nodes outside `lib2` have a granularity level of ‘3’, meaning they should be represented with 3 name elements (in this case package, class and method).

Compacted names create a ‘take-over’ relationship between nodes (shown with red arrows on Figure 9, and noted ⊲ in the following) which indicates how nodes should merge in the resulting graph. The goal of this relationship is to capture the interplay of both structural and behavioural closeness to implement a localised merging mechanism. Structural because only nodes that belong to a common enclosing package (e.g. `lib2` in Figure 9) should merge together. Behavioural because this merging should only happen between nodes that lay in each other’s vicinity in the call-tree.

The concept of vicinity is meant to encompass children and siblings, but needs to be defined somewhat more broadly to capture the situation where two nodes are brought close together because their parents have merged. For instance in Figure 9, two leaf nodes refer to the method `lib3.Signal.travel()`. In the fully expanded tree of Figure 9, these two nodes are neither siblings, nor descendants of one another, and are therefore represented as independent nodes. However, once the nodes belonging to `lib2` are merged into one
compacted node (upper enclosing shape in Figure 9), both
\texttt{lib3.Signal.travel()} nodes become 'siblings' referring to the
same program element and should therefore also be merged
(with an appropriately updated weight, as explained below).

More formally, we say a node \textit{A takes over} a node \textit{B} (\textit{A} \sqsubseteq \textit{B}) if and only if one of the following conditions holds:

- \textit{A} and \textit{B} are the same node (the relation is reflexive)
  \[ \textit{A} = \textit{B} \quad (1) \]

- the compacted name of \textit{A} is a prefix of the compacted
  name of \textit{B} (this includes the case when both compacted
  names are equal) and
  \[ \textit{A} \sqsupset \text{parent}(\textit{B}) \quad (2) \]

- or the parents of both \textit{A} and \textit{B} are taken over by
  the same node, i.e.
  \[ \exists \text{D} : \text{D} \sqsupset \text{parent}(\textit{A}) \land \text{D} \sqsupset \text{parent}(\textit{B}) \quad (3) \]

where \text{parent}(\textit{X}) denotes the parent of node \textit{X}.

Note that Condition (2) encompasses the case where \textit{A} is
\textit{B}’s parent, since \textit{A} \sqsupset \textit{A} (by Condition (1)). Similarly,
Condition (3) encompasses the case where \textit{A} and \textit{B} are siblings,
by selecting \textit{D} = \text{parent}(\textit{A}) = \text{parent}(\textit{B}).

For instance, in Figure 9, \texttt{lib2.Lung.inhale()} (whose
compacted name is 'lib2', and hence should be represented as
a ‘lib2’ node) takes over both \texttt{lib2.Muscle.contract()} and
\texttt{lib2.Muscle.stop()} because of rules (1) and (2). \texttt{lib2.Lung.-}
\texttt{inhale()} also takes over the two \texttt{lib2.Nerve.transmit()} nodes
because of rule (2). Finally the two \texttt{lib3.Signal.travel()} nodes
take each other over symmetrically because of rule (3).

The nodes of the resulting \textit{compacted tree} are the con-
ected components of the take-over relationship (represented as
free-form shapes in Figure 9). The weight of each com-
 pacted node is that of the highest node being merged in
the original tree, if there is only one such node (e.g. \texttt{lib2-}
\texttt{Lung.inhale()} in Figure 9), or the sums of the weights of the
highest nodes if there are several (such as the two leaf nodes
\texttt{lib3.Signal.travel()} in our example).

### 3.3 Prototype

#### 3.3.1 User interactions

Using the previous mechanism, two actions can be offered
on each node of a compacted tree: localised expansion and
compaction. Essentially, compacting a node will lower the
granularity level of all the nodes in the original profiling tree
that correspond to the selected compacted node. An expa-
nsion is the reverse: the granularity is raised. In both cases a
new merged tree is computed, and a dynamic animation is
used to highlight how nodes either merge or separate.

To limit the amount of change provoked by compaction and
expansion, we prevent users from lowering the granular-
ity level of a compacted node below that of its parent (since
by construction a child cannot take over its parent). We
also forbid situations in which take-over relationships would
occur across more than one level of package hierarchy: e.g.
a node with compacted name \texttt{lib2} cannot take over a node
with compacted name \texttt{lib2.Muscle}. The granularity
level of the second node would first need to be lowered
to \texttt{lib2.Muscle}. In both cases, the offending action is simply
blocked, and an explanatory message displayed to the user.
Both limitations are design decisions rather than inherent
constraints of the compaction mechanism.

#### 3.3.2 Implementation

Our prototype, called ProfVis, implements the above com-
 paction and expansion features using the graphical program-
ing framework Processing [2] (Figure 11). Although the lo-
calised structural compaction we propose can be combined
with more traditional tree navigation features, our prototype
limits interaction to structural expansion and compaction to
facilitate the study of this particular navigation approach.
The available actions offered to a user are pan (mouse drag),
zoom-in ('+') , zoom-out ('-') , package-expand a node (left
click), package-compact a node (right click), global expan-
sion by one level (right arrow), and global compaction by
one level (left arrow).

The CPU utilisation of each node is represented by its
area, and structural closeness by colours: nodes belonging
to the same enclosing class or package are shown in similar
hues. We also use a simple semi-circular graph layout: nodes
and their children are recursively allocated angular sectors
in the 2D plane and positioned on a radial layout.

A screenshot of the prototype is shown on Figure 11 when
running on the same Globus profiling data as Figure 4, and
the layout of the fully compacted Globus tree is shown in
Figure 12. This fully compacted graph only contains 89
nodes, which compares favourably against the 1341 nodes in
the original profiling tree.

![Figure 11: The prototype applied to a Globus trace](image-url)

4. EVALUATION

#### 4.1 Overview and rationale

To explore the quantitative and qualitative issues involved
in the use of our prototype (ProfVis), we ran a small scale
user study with four users. Each user was asked to com-
plete the same comprehension task on four different pro-
grams (two small and two larger ones) with ProfVis, and
with the textual navigation tool we showed in the introduc-
tion (called TreeTable), to act as a comparison point.

We chose TreeTable as our baseline rather than a more
advanced technique (e.g. the spiral visualisation of BLOOM
[16]) for two main reasons: We wanted to visualise the same
underlying profiling tree with roughly the same degree of
interaction freedom to facilitate comparisons, and we wanted
to provide our test users with a semi-textual interface widely
used in the industry [3, 1], which they might be more familiar
with.
We first trained each test-subject during roughly half an hour on each tool. During this training session, test sub-
jects (also called ‘users’ in the following) were showed a pre-
recorded presentation on sample-based profiling, the mean-
ing of an inclusive profiling call-tree, and the general princi-
pies of both the TreeTable and ProfVis tools. The subjects
were then asked to complete simple understanding tasks
with both tools on the toy example of Section 3, and on
a larger profiling trace (obtained during a run of ProfVis it-
self). During the training period, we answered any question
the users might have on either tools (e.g. TreeTable and
ProfVis), the target programs, or performance analysis.

We then moved to the study proper which consisted in
analysing the profiling traces of four target programs, two
small and two larger ones, using TreeTable for the first two,
and ProfVis for the last two. Table 1 shows some static met-
rics of all four programs (collected with LOCC [11]) along
the size of profiling call graph considered.

BubbleSort and Simulation are two toy programs specif-
cally developed for this study which respectively implement
a bubble sort of 1,500 words, and a simple physical simul-
ation of 2000 balls connected by springs. OPSBrowser is
a call-graph construction and manipulation engine that is
part of the CosmOpen reverse engineering tool [20]. Finally
ws-core-3.9.4 is the Web-Service core of the grid computing
middleware Globus in its version 3.9.4. The version 3.9.x
of Globus was the first to integrate web-service technol-
ogies, and is a good representative of the type of systems we
mentioned in the introduction: a large and complex soft-
ware assembled in a relatively short time (a few months) by
reusing a number of pre-existing components (notably the
Apache axis JAX-RPC engine). Some of its performance
issues have been discussed in [19], which provides a good
baseline to assess our test-users’ understanding.

The comprehension task given to test users was broken
down in four steps: (i) to explain how the program was or-
organised; (ii) to indicate which part(s) of the program (method,
class, or package) could be modified to improve its perfor-
mane; (iii) to create a snapshot (with the tool) that illustra-
tes (i) and (ii); (iv) to sketch a diagram of the program’s
organisation; and finally (v) to rate from 0 to 10 the level to
which they thought they understood the target program to
be able to improve its performance (Perceved Unders-
standing).

A first group of 2 users were asked to perform the above
task first with TreeTable on BubbleSort and Globus (in this
order), then with ProfVis on Simulation and OPSBrowser
(also in this order). The distribution of target programs was
crossed over for the second group of 2 users, with TreeTable
first applied to Simulation and OPSBrowser, followed by
ProfVis on BubbleSort and Globus. To increase the intelli-
gibility of identifiers, and emulate the abilities of developers
to look up additional information in a real-life situation, we
provided each user with a list of spell-out acronyms (e.g.,
PKCS7, SAX, WSRF) and library names (Xerces, Clay-
We asked each user to verbalise their activities, and recorded each session, both as a video, and a stream of interaction events (node expansion, contraction, etc.). We performed 16 sessions in total: four per users, 8 per tool.

We finally assessed the understanding of performance issues reached by each user on each task by comparing their recorded verbalisation, snapshot and diagram to a set of key expected observations derived from the traces of each target program. We counted how many of these observations they had found and normalised their score to 10 (Assessed Understanding). The number of expected observations was relatively low (between 4 and 6 observations per program), and high-level. For instance, on the Globus trace, we expected users to make five observations: (i) that the CPU’s time is roughly split between a bootstrap and runtime phase; (ii) that the bootstrap creates a container; (iii) that the runtime phase executes the remote service; (iv) that the bootstrap was slow because of XML processing; and (v) the service execution because of security issues.

5. RESULTS ANALYSIS

Our analysis focuses on two aspects: we first contrast the various understanding measures obtained in the experiments, and then discuss some of the interaction patterns we observed in the use of each tool. The following analysis is of course constrained by the small size and nature of our user study: rather than a full-fledged controlled experiment, our aim here is more to highlight potential issues and trends in the use of localised structural compaction for performance analysis.

5.1 Understanding

![Figure 13: Contrasting perceived and assessed understanding](image)

Figure 13 contrasts each user’s perceived understanding (for each task) with their assessed understanding. Our goal in asking users to assess their understanding was both to elicit a measure that eschewed any value judgement on our part, and reflected each user’s subjective experience while avoiding a potential social desirability bias towards ProfVis. As Figure 13 shows, the two measures are largely unrelated: some users thought they did well in some tasks, while missing most of the key points and thus scoring low on the assessed measure, while others did the reverse. Some patterns do seem to appear though: Users are best aligned with their assessed performance when analysing small programs with TreeMap (hollow rhombus); they tend to underestimate their understanding of large programs with ProfVis (solid squares); and tend to overestimate their understanding of both small programs with ProfVis (hollow squares) and large programs with TreeMap (solid rhombus).

One possible explanation is that TreeMap only displays as many nodes as the user has expanded. As a result users may easily perceive a large trace graph as smaller than it really is, and from there assume they have reached a reasonable understanding when they have missed some key parts of a program’s execution. By contrast, ProfVis forces users to confront a program’s full call-tree from the onset, even if in a highly compacted form. For instance, the fully compacted version of the Globus traces contains 89 nodes when ProfVis starts (Figure 11), while TreeMap only shows two lines for the same trace file.

![Figure 14: Cumulative distributions of understanding measures: small programs vs. larger ones](image)

As a complement to Figure 13, Figure 14 shows the cumulative distribution of perceived (top) and assessed (bottom) understanding measures for small (BubbleSort and Simulation) and large programs (OPSBrowser and Globus). Figure 14 shows the same information for TreeMap and ProfVis. For instance on the top chart of Figure 13, 5 sessions yielded a perceived understanding of 8 or more for small programs (solid dots). Figure 14 shows that although users felt they understood less of larger programs, we did not perceive a noticeable difference in our assessment. This probably simply reflects that our key expected observations (Section 4.2) were adapted to each program’s size and complexity, thus ironing out some of effects of size on the measure.
Of key interest for the present work, Figure 15 indicates a slight advantage for ProfVis over TreeTable in terms of both assessed and perceived understanding.

5.2 Interaction patterns and strategies

For each task session, we recorded the depth (in the displayed tree) at which users compacted or expanded nodes. Plots of this depth of interaction against time is shown for TreeTable (top) and ProfVis (bottom) in Figure 16 for large programs (OPSBrowser and Globus) and in Figure 17 for small programs (BubbleSort and Simulation). Figure 16 clearly shows that on large programs, most users adopt a depth-first strategy with TreeTable, rapidly moving deep into the call tree along a single execution branch (generally that of the most weighted child), and only occasionally backtracking through large jumps back to the top of the tree. By contrast users go far less deep with ProfVis, and tend (for the majority at least) to keep interacting at the same depth over long periods of time (appearing as 'plateaus' on the Figure). A similar trend can be discerned for small programs (Figure 17), although not as clearly.

This pattern might be explained by the difference or presentation in the two tools. The layout of TreeTable naturally encourages users to go deep first: the next child with the highest share of CPU usage is always the closest and lies in a predictable position. By contrast, the relative location of nodes in ProfVis evolves in a two-dimensional plane with each new interaction. As a result node positions are far less predictable, possibly deterring users from rapidly moving away from their current position.

5.3 Threats to validity

Besides the inherent difficulty in defining and measuring understanding, and the small size of our study, our measurement of the understanding reached by our users is influenced by a large number of factors besides the particular visualisation tool being used. For instance, the semantic information born by identifiers is obviously critical to users in our experiments, since our test subjects did not know the target programs, nor had access to any source code. Identifiers might be more or less descriptive, and might speak to one subject more than to another. How they are interpreted is also obviously influenced by a user’s prior knowledge: Most of our users declared a high proficiency in Java, but a generally low expertise in the other involved technologies (XML, Globus), and in the analysis of sample-based profiling traces.

The influence of prior knowledge and training goes however beyond the mere interpretation of identifiers. A minimal grasp of the meaning of profiling trees is for instance critical: profiling trees collapse concurrent activities in one single call-tree, and do not contain any information on the ordering or frequency of individual invocations. At least one of our test subjects misunderstood these limitations, and tried to reconstruct the exact sequence of calls each program was going through, a particularly hard, if not impossible task on the sole basis of the available information.

6. RELATED WORK

Performance analysis, both automated, and semi-manual, is a thriving area of research (e.g. [9, 23, 17]). For space reasons, we focus in the following on approaches that combine both static and structural elements and are generally geared
occurs off-line and is static. For each collection, the tool realises an execution, the user must first provide a mapping of low-level entities (objects) to higher-level groupings (collections) that makes sense for the task at hand. This grouping is directed toward comprehension and reverse engineering rather than fully automated diagnosis.

Numerous works have investigated the fusion of both structural and dynamic data to support program comprehension tasks [18, 12, 21, 16, 5, 10, 8]. The key idea is similar to ours: By selectively folding or hiding recurring patterns (in our case invocations belonging to the same enclosing package), these approaches decrease the complexity of the data to be represented, while retaining enough information to capture the program’s internal logic. Jerding et al. for instance proposed a pattern extraction technique that collapses identical subtrees in the original call-tree, and identifies duplicated subtrees generated by iteration and recursion [12]. This pattern-induced collapsing of subtrees has also been used by Puuw et al. in JinSight to help locate memory leaks in Java programs. Their technique groups objects according to their class and the other objects they refer to [6]. By compacting reference relationships into patterns, they help users comprehend complex object graphs, while maintaining enough information to discriminate objects accordingly to their situation of referencing.

AVID (Architectural VISualisation of Dynamics) presented by Murphy et al. [21] reduces the complexity of dynamic behavioural data by constructing an architectural view of a running object-oriented program. The tool records method invocations, object allocations and de-allocations. To visualise an execution, the user must first provide a mapping of low-level entities (objects) to higher-level groupings (collections) that makes sense for the task at hand. This grouping occurs off-line and is static. For each collection, the tool counts particular events (such as the number of objects allocated) and represents them as histograms attached to the collection. The tool also draws an edge whenever an object in a particular collection invoked an object in another one, and labels this edge by the number of invocations between the two collections. The current state of the call stack at the point of visualisation is represented as a path running through the collections that are traversed by the program’s thread (which the authors call an hyperarc).

Shimba [18] is a reverse-engineering environment for Java that supports the parallel exploration of both static and dynamic views of a program. Shimba allows users to correlate structural and behavioural data by filtering one type of data using the other (a technique termed model slicing). Shimba offers advanced analysis techniques to reduce the size of dynamic data: (i) it can synthesise statecharts from sequence diagrams; (ii) it can also detect behavioural patterns in sequence diagrams and replace them by a repetition construct.

Similar to Shimba, BLOOM [16] is an integrated system for software visualisation, covering data collection, analysis, and visualisation of both static and dynamic information. One of its key features is a visual language that allows users to specify what should be represented and how. BLOOM works on event traces that contain method invocations, exits, and memory management events (allocation, de-allocation), and encompasses performance analysis. Among the analysis provided, BLOOM can construct direct acyclic graphs from the trace data in which identical call-paths are collapsed together. Closely related to the work presented in this paper is BLOOM’s package encoding analysis that allows users to specify how particular library calls should be merged together. Rather than being interactively determined by the user, however, the merging policy is defined in an external specification written in XML.

A number of recent works have proposed to use interactive structural compaction to help developers analyse dependencies between program entities [15, 5]. DA4Java [15] and Creole\footnote{http://www.thechiselgroup.org/creole} for instance use nested nodes to represent structural relationships between program entities, and allow users to vary the level of abstraction at which nested nodes are represented, as we do. Cornelissen et al [5] use a similar technique in the context of circular bundle views, based on a technique first proposed by Holten [10]. Circular bundle views arrange a program’s elements (methods, classes, packages) in a circle, and represent the call relationships among these elements (e.g. A is calling B during a particular observation window) as bundled edges in the centre of the circle. Hierarchical relationships between program elements (package P encloses class A) are denoted by using concentrical circles for each hierarchical level, and insuring the angular span of P encloses that of A.

As in our work, Creole, DA4Java, and circular bundle views allow users to collapse elements into their enclosing parent (i.e. their enclosing class or enclosing package), in which case edges are correspondingly updated. Contrarily from our work, however, these approaches are limited to interaction diagrams where elements are only represented once, in contrast to call trees, where the same method might appear in multiple locations. As a result, they are unable to realise localised compactions, and simply apply a uniform level of compaction to the same element across their respec-
8. REFERENCES


7. CONCLUSION

We have presented a novel navigation approach to help developers explore complex dynamic profiling information by selectively raising or lowering the abstraction level of the parts of the program’s execution they are visualising. Our approach exploits the structural information found in profiling traces. We have realised a prototype implementing this navigation technique, and have presented an early evaluation campaign that hints at the potential benefits of our approach when compared against the current industrial practice in profiling tree navigation.

We purposefully implemented a limited prototype to explore the benefits of our compaction technique. However, because our technique essentially produces an alternative, more compact tree, it can be combined with almost any additional tree navigation and visualisation approach, such a branch-base collapsing, or advanced layout and panning techniques. This integration is indeed an aspect we would like to study further in the future.

tive representation. By contract, the approach we propose allows the same element to be represented at different levels of abstraction within the same graph, a key advantage for behavioural representations, in which the same structural elements might appear in different unrelated contexts.

Structural collapsing is more generally related to the notion of graph roll-up, discussed for instance by Wattenberg [22]. A graph roll-up aggregates all nodes sharing a particular predicate, and merges the corresponding edges while updating both node and edge meta-data. In contrast to our technique, however, graph roll-ups are uniform, whereas in our proposed approach aggregation propagates locally in the profiling tree to deliver a localised merging mechanism.