Trevis: A Context Tree Visualization & Analysis Framework and its Use for Classifying Performance Failure Reports

Andrea Adamoli
Faculty of Informatics
University of Lugano
Lugano, Switzerland
andrea.adamoli@usi.ch

Matthias Hauswirth
Faculty of Informatics
University of Lugano
Lugano, Switzerland
matthias.hauswirth@unisi.ch

ABSTRACT
When developers profile their applications to identify performance problems, they normally use profilers producing calling context trees. A calling context tree (CCT) represents the caller callee relationships of a program. A dynamically collected CCT provides method coverage information. The CCTs produced by profilers also include method hotness information. Trevis, our context tree visualization and analysis framework, allows users to visualize, compare, cluster, and intersect CCTs produced by profilers. We evaluate Trevis in the context of a novel profiling tool called FlyBy. FlyBy runs transparently on an end-user’s computer and continuously samples the applications’ call stack. When the user perceives the application as sluggish, she presses a “Was Slow!” button to tell FlyBy to file a performance failure report. The report contains the CCT based on the call stack samples FlyBy gathered over the last few seconds before the user pressed the button. We show how Trevis allows us to visualize and classify FlyBy bug reports.

Categories and Subject Descriptors
H.5 [Information Interfaces and Presentation]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms
Measurement, Performance, Design, Experimentation

Keywords
Tree visualization and analysis, perceptible performance

1. INTRODUCTION
Imagine the following scenario. End users, whenever they are dissatisfied with the application program they are working with, can press a “Complain!” button to file a complaint. The system transparently gathers information about the application’s behavior prior to the user’s complaint, and it sends the complaint, including the relevant behavioral information, to the developers. The developers classify the received complaints using the behavioral information and produce a list of bug reports, where each bug report is based on a number of similar complaints. They then find and fix the bugs, based on the dynamic information available, and they prioritize their work based on the popularity of each bug report in terms of its number of complaints.

In this paper we present a tool called FlyBy, which realizes a special case of the above scenario. FlyBy deals with complaints about an application’s interactive performance, in particular about its responsiveness. The “Complain!” button is called “Was Slow!”, and the behavioral information collected by FlyBy corresponds to periodic call stack samples of the application’s threads. At the time a user complains, FlyBy packages the call stack samples received over the last few seconds, and ships them to the developer’s server. Instead of developers manually classifying complaints to produce bug reports, FlyBy clusters complaints based on the calling context tree (CCT) produced from their call stack samples. It allows developers to visually explore the hierarchy of clusters in the form of a dendrogram, and for each cluster it produces and visualizes a representative CCT that helps the developer to understand the application’s behavior during its unresponsiveness.

At the core of the FlyBy tool is a framework for visualizing, comparing, clustering, and intersecting context trees. This framework, Trevis\footnote{http://sape.inf.usi.ch/trevis}, is extensible in several important dimensions: the tree model, the visual properties used for rendering trees, the distance measure used for comparing trees, the agglomeration method used for clustering trees, and the algorithm used to compute a representative exemplar from a set of trees. Trevis comes with a set of default implementations for each of these extension points. We implemented Trevis in Java, based on Swing.

In Section 2 we introduce the prior work on which our ideas are based. The remainder of the paper consists of two parts. The first part describes Trevis: Section 3 describes the context tree model, Section 4 presents the context tree visualization framework, Section 5 explains the context tree comparison approaches, Section 6 shows how we cluster context trees, and Section 7 describes our approach to tree intersection. The second part of the paper evaluates the Trevis framework by using it to implement FlyBy: Section 8 introduces the FlyBy system and the idea of end-user per-
formance failure reporting, and Section 9 shows a case study of the use of FlyBy. Finally, Section 10 concludes the paper.

2. RELATED WORK

Prior work [14, 4] has demonstrated the usefulness of collecting and clustering failure reports in the field. Our work extends this idea to performance failures and bases the reports on transparently collected CCTs. Unlike in prior systems, which file a failure report as a result of a fatal crash or hang, in our FlyBy system, the end user can press a button to file a complaint at any time. This means that we can gather complaints about problems that are not classically considered failures. In particular, we can collect information about behavior that exhibits perceptibly long latency. In prior work, with the help of a profiler that measures interactive response time [5], we have found perceptible latency to be a common problem in interactive Java applications [1].

Our FlyBy profiler collects and analyzes CCTs, a structure that is commonly used when analyzing application performance, because it provides an intuitive mapping of performance metrics to source code. Prior research specifically studied CCT differencing between program runs [19] or program versions [10], the detection of repeated patterns in CCTs [2], as well as the visualization of large CCTs using calling context tree ring charts (CCRC) [9, 8].

CCRCs, which are at the basis of the visualization component of our Trevis framework, are based on the general idea of space-filling visualizations of tree data structures [16], and more specifically, on the idea to use concentric rings [18]. Unlike the above approaches, which visualize a single tree, TreeJuxtaposer [12] enables the visual comparison of two phylogenetic trees (where internal nodes have no labels) using a focus+context approach.

The goal of our Trevis framework is to enable users to study a large number of trees, for example, the set of CCTs collected using FlyBy complaints. Trevis thus combines visualization with the various forms of analysis necessary for the automatic comparison, clustering, and intersection of trees.

3. CONTEXT TREE MODEL

A ContextTree is a tree consisting of ContextTreeNode subclass. A NodeAttribute can, given a ContextTreeNode, produce a value representing some attribute of that node. NodeAttributes can be quantitative (numerical values) or qualitative (textual values). A numerical NodeAttribute can be inclusive or exclusive. Inclusive means that the value of the NodeAttribute of a parent node is greater or equal to the value of the NodeAttribute of each of its children. For a tree to be a context tree, it has to fulfill the following three properties:

1. Each node contains a label (which we call a “frame”)
2. The frames of sibling nodes must differ.
3. There is at least one inclusive NodeAttribute.

The main use of our framework is the representation of CCTs, where nodes are labeled with the stack frame (activation record) of the method call they represent. Our framework is not limited to representing CCTs, though. It can operate on any tree fulfilling the above properties.

The second property allows the children of a node to be brought into a canonical order, and thus the tree to be processed as an ordered tree. The third property is necessary for the successful layout of the tree. It is trivially fulfilled by the class DescendantCountAttribute, which, given a node, evaluates to the number of its descendants.

Users can subclass the ContextTree class to model their own kinds of trees. Normally, a subclass would add a constructor that builds a tree (consisting of objects of a ContextTreeNode subclass) from some external source of information, such as a trace or profile. We have produced several different instantiations of this model:

File system. A context tree represents a folder on a hard disk. The label is the unqualified file name, and the inclusive attribute is the inclusive size of that folder.

Bytecode program structure. A context tree represents the nested structure of a Java application, starting at the top level packages, nested packages, classes, and methods. The label is the unqualified name and signature of the program element. The inclusive attribute is the number of bytecode instructions.

Java hprof profile. The context tree represents the sampled CCT produced by Java’s built-in hprof profiler. The label is the class name, method name, Java source file name, and line number of a node’s call stack frame. The inclusive attribute corresponds to the number of call stack samples hprof took while a given method was somewhere (not necessarily on the top) on the call stack.

NetBeans profile. The context tree represents the filtered CCT produced by the NetBeans Java profiler. The label is the class name, method name, and method signature of a node’s call stack frame. The inclusive attribute represents the amount of time the program spent in the given method or any of its callees.

FlyBy profile. This is the model we use in the remainder of this paper. The context tree corresponds to a sampled CCT representing a short period of activity corresponding to a performance failure. The label is the class and method name of a node’s call stack frame. Like for the hprof profile, the inclusive attribute corresponds to the number of call stack samples.

4. CONTEXT TREE VISUALIZATION

Originally, Trevis could visualize trees only through the graph layout and rendering package GraphViz [3]. However, GraphViz visualizations do not scale easily to large trees and are limited in their interactivity. Our second rendering approach is based on Moret et al.’s [9] calling context ring charts (CCRC). For both rendering approaches, our framework provides extension points for developers to map node attributes to visual properties of the rendered trees.

Angle. For CCRCs, the angle of a ring segment is determined by any quantitative inclusive attribute of the corresponding context tree node.

Line width. For GraphViz visualizations, the line width of a node is determined by any quantitative attribute of the corresponding context tree node.
**Saturation.** The saturation of a ring segment is determined by any quantitative attribute of the corresponding context tree node. This attribute does not necessarily have to be inclusive.

**Hue.** The hue of a ring segment is determined by any qualitative attribute of the corresponding context tree node. The framework will scan the context tree to determine all unique values of that attribute, and it will produce a hue map of equidistant hues (e.g. for an attribute with three possible values, the hues will be red, green, and blue).

**Filter threshold.** This property is most useful when rendering via GraphViz. It is determined by a quantitative attribute of the corresponding context tree node. It specifies a threshold value, below which a subtree is omitted. This helps prune unimportant nodes for large trees, which allows a user to focus on the big picture.

**Info.** The CCRC visualization overlays information about the ring segment under the cursor. The GraphViz visualization renders that info inside each tree node. That information is provided by a configurable list of quantitative or qualitative node attributes.

![Figure 1: GraphViz Rendering of Example CCT](image)

**4.1 Angle**

The primary visual property of our context tree visualization is the angle of ring segments. Figure 2 shows the effect of using two different attributes to determine the angle. The top tree uses the number of descendants of a node. This attribute does not depend on the specific context tree model used, because the number of descendants can be computed in any model. It renders each leaf node with the same angular size, and sizes internal nodes according to their subtree’s size. The bottom tree uses the inclusive sample count to determine a node’s angle. This sizes nodes according to their method’s hotness: a method that executes for a long time will be represented with a bigger ring segment, and methods that do not affect execution time much are drawn as small segments.

**4.2 Color**

Figure 3 shows the effect of using a different saturation attribute. The upper tree uses the inclusive sample count (the same attribute we use for the angle), while the lower tree uses the exclusive sample count.

The small table in Figure 4 shows the hue map, based on the component\(^1\) to which a corresponding context tree node belongs. Our model provides a `NodeAttribute` that determines the component based on the node’s label (the

\(^1\)Our context trees represent performance problems reported by a user of Eclipse. A `component` roughly corresponds to an Eclipse feature (a set of related plugins).
first few components of the package name of the node’s method). Figure 4 shows the same context tree, but colors the nodes according to component. The top tree uses a constant saturation, while the saturation in the bottom tree corresponds to the inclusive sample count. This combination of two distinct attributes (component and inclusive sample count) onto one visual property (color) could be problematic. However, given that we define the colors by hue and saturation, and given a relatively small number of hues, the resulting color still allows a user to map back to the two attributes. For example, the gray ring segments towards the bottom left area of the tree represent nodes with low saturation (low inclusive sample count, and thus low importance with respect to performance). Nodes with higher saturation levels have a higher relevance to performance, and allow the clear distinction between the different hues (application components).

4.3 Node Selection

Our context tree visualization is interactive. Figure 5 shows the result of hovering over a ring segment with the mouse. The visualization always highlights the ring segment under the mouse by changing its brightness. Given that we already use the hue and saturation components of a color, using the brightness for yet another purpose can be problematic. However, we only use two levels of brightness: dark for selected nodes and bright for all other nodes. In our own use of our tool we found this use of hue, saturation, and brightness for the representation of these three different aspects (a category, a numerical value, and a boolean value) to work reasonably well, as long as the hue only used a small number of distinct values.

To select a ring segment, the user needs to position the mouse cursor over that segment. This is not easy given the circular shape of a ring segment. To support the user in this task, we also render a polar-coordinate cross-hair cursor, consisting of a ray originating from the center and a 10 degree circular segment.

The nodes in our CCRC visualization are too small to include any textual information. Thus, to provide details about the selected node, the visualization includes a header area rendering multiple lines of text. The contents of that area is entirely determined by a configurable list of node attributes. In the FlyBy CCT visualization tool, the header area shows two lines, one for the class and one for the method name. Moreover, the header area also shows the name of the given tree (first line, in bold), which is particularly important in a tool that visualizes multiple trees.

Finally, a footer area in the visualization contains a legend for the specific angle, hue, and saturation of the selected node. It shows the values of the specific attributes and the visual representation of those values. For the hue and saturation attributes, the visual representation corresponds to a circle filled in a color. For the angle attribute, the representation shows a circle with a highlighted segment of the corresponding angle.

4.4 Handling Large Trees

It is not rare that the CCTs FlyBy gathers from realistic applications have heights greater than 100. This is a direct consequence of the call stack depths observed in those
programs. For event-based applications, which includes any GUI applications or any server-based applications, we found two reasons for such excessive stack depths. First, the stack of the event dispatch thread, when it reaches the point where it can start dispatching events, often already is several dozen frames deep. Second, modern applications are built on rich frameworks that provide many layers of abstraction, and each abstraction introduces additional indirections. Each indirection corresponds to an additional method call, and thus to an additional frame on the call stack.

Our visualization framework offers two mechanisms to deal with large trees: scaling and focusing. Figure 6 shows the results of scaling (top) and focusing (bottom).

Figure 5: Interactive Node Selection

Figure 6: Scaling (Top) and Focusing (Bottom)

Scaling. To scale a tree visualization, we just reduce the thickness of a ring segment, which leads to a reduction of the diameter of the visualization. By default we render rings that are 5 pixels thick, and we add one pixel as a white separator between rings. By reducing the thickness, we can fit a much higher tree into the same amount of screen real estate (a tree of height 100 will produce a visualization with a 200-pixel diameter, plus a few extra pixels for rendering some information in the center of the ring). Unfortunately, the absence of the white separators between ring segments greatly reduces the power of the visualization: It becomes impossible to distinguish between nested ring segments. This problem is alleviated if ring segments are colored, for example by their component. Then nested segments in the same component appear as a contiguous area, but segments from different components are visually distinct. A scaled visualization with a ring thickness of 1 is good to get an overview of a tree. For understanding the details, and investigating individual nodes, scaling to 5 pixels is much more useful.

Focusing. The usefulness of scaling is limited. If we scale the ring thickness to less than one pixel, we lose a significant amount of information. Thus, we also provide a means to focus the visualization of a subtree rooted in a specific node. While a user focusing on a subtree will lose context, we at least use the center ring to indicate that the current visualization is focused on a subtree. Focusing on a node allows us to render visually discernible rings (e.g. with the default size of 5 pixels), which means that the user can identify individual nodes.

We have observed that FlyBy CCTs often have a very particular shape. The root node often has only a single child, and that child also only has one child, and so on. This chain of single-child nodes can be over a dozen nodes long before reaching a node with more than one child (which we call “top-level split node”). That chain from the root is often not interesting to a performance analyst (it usually stays the same for all trees representing the behavior of a given application, because it represents the context of that application’s main event loop). For this reason we added a navigation command that directly focuses on the top-level split node.

5. CONTEXT TREE COMPARISON

Trevis can compare two context trees by computing the distance between them. It supports pluggable distance measures, and it can visualize the result of comparing multiple trees in the form of a similarity matrix.

Table 1: Taxonomy of Distance Measures

<table>
<thead>
<tr>
<th>Node</th>
<th>Existence set</th>
<th>Context multiset</th>
<th>Hotness weighted multiset</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSD</td>
<td>TED, MTD, RTD*</td>
<td>WTED*, WMTD*, WRTD</td>
<td></td>
</tr>
<tr>
<td>NMD*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WNMD</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Taxonomy of Distance Measures

Trevis already includes a set of distance measure plugins. Table 1 classifies those context tree distance measures along two dimensions: The rows distinguish whether a measure considers a tree’s structure, and the columns distinguish measures by how they deal with node weights.

A distance measure may consider the structure of the trees, or it may compare trees based solely on information about nodes (and thus ignore the parent-child relationships between nodes). Given that trees can be large (it is not uncommon for CCTs to grow to hundreds or thousands of

4Measures marked with a * are under development.
nodes), structural measures can become costly to compute. Node measures reduce a tree to a bag of nodes, and thus allow more efficient comparisons. Moreover, when comparing CCTs, node measures can be useful when we care more about which methods were called, but care less about the specific caller-callee relationships.

Distance measures also differ in their weighting approach. Existence measures compare trees merely based on the similarity of their node label sets. They do not take into account how often a specific node label appears in a tree. For example, a node-based existence measure considers all trees containing any number of nodes with labels ‘A’, ‘B’, and ‘C’, but no other nodes, as equivalent. For CCTs, using an existence measure is a way to compare their method coverage. Context measures take the number of occurrences (resp. the number of contexts in which a label appears) of a given label into account. They often are implemented based on multisets of node labels. For example, a node-based context measure considers all trees containing 2 nodes with a label ‘A’ and 3 nodes with a label ‘B’, but no other nodes, equivalent. Hotness measures take node weights into account. In Trevis’ context tree model, the weight of a node corresponds to an exclusive attribute of the node. For example, a node-based hotness measure considers all trees for which the weights of the ‘A’ nodes add up to 15 and the weights of the ‘B’ nodes add up to 4, and which contain no other nodes, as equivalent. In CCTs that represent performance profiles, labels correspond to code regions (e.g. methods), and weights correspond to a performance metric (e.g. execution time). Hotness measures are thus useful to distinguish between trees representing similar behavior with different performance.

The measures from Table 1 are defined as follows.

**NSD - Node Set Distance.** This measure computes the Jaccard distance \(d(A, B) = \frac{|A \cap B|}{|A \cup B|} \) between the two trees’ node label sets \(A\) and \(B\). This is probably the most straightforward distance measure for comparing two labeled trees. The NSD is based on the measure used by TreeJuxtaposer [12] to compute the similarity of internal nodes of phylogenetic trees (where internal nodes have no labels).

**NMD* - Node Multiset Distance.** Instead of node label sets, this measure uses node label multisets. A node label multiset can contain multiple occurrences of the same label. Instead of using set union \(A \cup B\) and intersection \(A \cap B\), we use the corresponding operations for multisets [11] \(A \cup B\) and \(A \cap B\). For CCTs, the NSD essentially represents a comparison of method coverages, while the NMD also considers the number of distinct contexts a given method was executed in.

**WNMD - Weighted Node Multiset Distance.** This metric is also based on multisets, but it includes a node’s weight. While the NMD counts the number of nodes with a given label, the WNMD sums up the weights of all nodes of a given label. In the domain of CCT comparison, unlike the NSD and the NMD, the WNMD also takes a method’s hotness (execution time, sample count) into consideration. Using the WNMD on a sampled CCT (such as the ones produced by the FlyBy profiler), is equivalent to comparing the flat call profiles in a traditional profiler. The weighted node multisets correspond to a list of methods, with total exclusive time spent in each method, independent of the calling context of that method.

**TED - Tree Edit Distance.** The tree edit distance [17] is the standard measure for comparing labeled trees. Unfortunately its algorithmic complexity does not allow us to use it for large trees. Moreover, the standard TED does not consider node weights.

**MTD - Multiset Tree Distance.** Molina et al.’s MTD [11] computes the average of the NMD and a subtree multiset distance. The subtree multiset distance is equivalent to the NMD, but instead of a multiset of nodes, it uses a multiset of subtrees. By including subtrees, this distance measure also accounts for structural differences. For example, the swapping of caller and callee (e.g. method \(m\) delegates to method \(x\), instead of method \(x\) delegating to method \(m\)) would not affect the NMD, but it would affect the MTD.

**RTD* - Rooted Tree Distance.** The RTD is a distance measure based on common tree matching [19]. Given two trees, we use common tree matching to build their common tree (which we call intersection tree). We also build a corresponding union tree, by merging them starting from the roots. The RTD corresponds to the number of nodes in the intersection tree divided by the number of nodes in the union tree. The RTD has a significant weakness when comparing two trees that contain possibly large similar subtrees but where the corresponding subtrees have different paths to the root. This is because of the common tree matching approach that matches trees starting from the root. Adding a single indirection, e.g. by introducing a façade, decorator, or proxy method, somewhere close to the root in the call tree, will lead to a large RTD, even for otherwise identical trees.

**WTED* - Weighted Tree Edit Distance.** This measure is based on the TED, but the cost of each tree edit operation is multiplied by the weights of the nodes involved.

**WMTD* - Weighted Multiset Tree Distance.** This is equivalent to the MTD, but each node is weighted by an exclusive attribute, and each subtree is weighted by the corresponding inclusive attribute evaluated on the subtree’s root.

**WRTD - Weighted Rooted Tree Distance.** This is similar to the RTD, however we compute the ratio based on weights: we divide the inclusive weight of the intersection tree by the inclusive weight of the union tree.

Trevis can be extended with further distance measures, such as those used for AST-based code clone detection [15]. A distance measure computes the distance between two context trees. Trevis can visualize the distances between any number of trees using a dissimilarity matrix. Figure 7 shows an example of such a matrix for three trees. The three context trees represent three FlyBy profilers we gathered for the same user request in three different versions of an interactive application. The matrix shows the MTD between those trees. The row and column headers visualize the
context trees of Figure 7. We used the multiset tree dis-
sion method based on the Lance-Williams dissimilarity up-
senting all of the context trees. Trevis can use any of the
a larger cluster, until it ends up with a single cluster repre-
with a set of clusters, one cluster for each individual tree.
It then always combines the two most similar clusters into
larger numbers of trees, the matrix is scrollable, similar to
spreadsheet, and the headers are always visible.

6. CONTEXT TREE CLUSTERING

When analyzing multiple context trees, it is often useful
to be able to group similar trees together. Trevis classifies
tree based on their distance, using hierarchical agglom-
agglomerative clustering [13]. This clustering approach starts
with a set of clusters, one cluster for each individual tree.
It then always combines the two most similar clusters into
a larger cluster, until it ends up with a single cluster repre-
senting all of the context trees. Trevis can use any of the
available tree distance measures to determine the similar-
ity between individual trees, and it can use any agglomera-
tion method based on the Lance-Williams dissimilarity up-
date formula [6], to compute the similarity between clusters.
These standard agglomeration methods include single link-
age (nearest neighbor), complete linkage (diameter), group
average, median, centroid, and Ward (minimum variance).

The result of hierarchical agglomerative clustering is a
dendrogram, a tree with a leaf node for each context tree,
and an internal node for each cluster. Trevis visualizes this
result using a dendrogram table. This visualization combines
a visual dendrogram with a table of context trees. Each
row in the table corresponds to a context tree and is aligned
with the corresponding leaf in the dendrogram. The columns
shown in the table are configurable, they correspond to a list
of tree node attributes which are evaluated on the context
tree’s root node.

Figure 8 shows a small dendrogram table for the three
context trees of Figure 7. We used the multiset tree dis-
tance measure and the group average agglomerative method
to perform the clustering. The four columns of the table
present the label, inclusive sample count, height, and num-
ber of descendants of each tree’s root node. The inclusive
sample count, for a sampled CCT, represents the number of
call stack samples used to build the tree. The number of
descendants of the root node corresponds to the number of
nodes in the tree, minus one. The dendrogram shows, in a
more compact form, the same findings we observed in the
similarity matrix.

The dendrogram table visualization is interactive. A user
can move a dissimilarity threshold, represented by a thin ver-
tical line in the dendrogram. This line cuts the dendrogram
into a fixed number of clusters such that the dissimilarity
within each cluster is below the given threshold. In Figure
9, the dendrogram is cut into two clusters: the red cluster
contains the trees in the top two rows, and the blue
singleton cluster contains the bottom tree. This interac-
tivity is a significant advantage of the hierarchical cluster-
ning approach over traditional k-means clustering (where the
number of clusters has to be predetermined). With our visu-
ization, a user can adjust the threshold until the number of
clusters makes sense. She can determine whether clusters
make sense by looking at the information shown in the
dendrogram table, or by clicking on any row in the table to
visualize the corresponding context tree. Moreover, given
that the dendrogram represents a clustering, which groups
similar trees together, our visualization has the advantage
that the context trees in the table are implicitly arranged
by similarity.

7. CONTEXT TREE INTERSECTION

In the previous section we described how a user can split
a dendrogram into clusters by adjusting the dissimilarity
threshold. Reducing a large number of context trees to a
small number of clusters is an effective way to summarize
information. For example, if we want to analyze several
hundred context trees submitted as FlyBy complaints, it is
important to be able to cluster them to end up with a few
clusters of similar complaints, ideally with each cluster cor-
responding to a different bug. Trevis’ clustering approach
can produce those clusters. However, we also need an effec-
tive way to succinctly visualize a given cluster. Visualizing
all of the many individual trees contained in a cluster is not
an option, because such a visualization would not scale. The
dissimilarity matrix in Figure 7 gives an idea of the problem
of succinctly presenting multiple context trees. To overcome
this problem, Trevis provides an approach to compute a re-
presentative exemplar of a set of context trees. We have im-
plemented two mechanisms to compute such an exemplar:
context tree union and context tree intersection. Union and
intersection are defined for sets, and have been extended for
multisets [11]. We base the corresponding operations for
context trees on the notion of common tree matching [19].
Intersection. We intersect two context trees by traversing them concurrently, starting at the root nodes. The intersection of two unweighted trees A and B is a context tree C, where the child set \( \text{children}(c) \) of a node \( c \) in tree \( A \) corresponds to the intersection of the child sets of the corresponding nodes in trees A and B: \( \text{children}(c) = \text{children}(a) \cap \text{children}(b) \). For context trees, where nodes have weights, we model the children as a multiset (a child’s weight determines how many times it appears in the multiset). Thus the intersection of a context tree node’s children is defined with the multiset version of the intersection operation \( \text{children}(c) = \text{children}(a) \cap \text{children}(b) \).

Union. The union of two context trees is computed analog to the intersection: the union of a node’s children is defined as: \( \text{children}(c) = \text{children}(a) \cup \text{children}(b) \).

These two operations are fast. They are linear in the trees’ sizes, because the trees are traversed in a lock-step fashion, as long as the child multisets of a node are available in a canonical order. Our tree implementations fulfill this requirement by ordering a tree node’s children with respect to their labels at tree construction time.

Figure 10: Union (Top) and Intersection (Bottom)

Figure 10 shows the union and the intersection exemplars of a set of CCTs, gathered by repeating similar operations. The figure shows that, obviously, the union exemplar is heavier, with 327 samples (the number in the center represents the tree’s weight in terms of samples), than the intersection exemplar with 76 samples. The union includes all behavior of all trees, while the intersection only includes common behavior. For this reason, the intersection is particularly useful as an exemplar representing a cluster of trees: any behavior that differs between the trees in the cluster will be eliminated, and only behavior that occurs in all trees is present.

8. PERFORMANCE FAILURE ANALYSIS

While Trevis is a general context tree visualization and analysis framework, we have built it with a specific application in mind. We want to use it to classify performance bug reports submitted by users of interactive applications. With the increasing prevalence of interactive applications, automatic failure reporting systems like Windows Error Reporting [4], which collect failures in the field and report them back to the developers, are becoming more popular.

FlyBy, the tool we developed for this case study, is a failure reporting tool for the collection and analysis of performance failure reports. FlyBy is targeting Java applications. It consists of three components. (1) The profiler consists of a JVMTI [7] agent, a native library dynamically linked to the Java virtual machine. It is continuously running below a monitored interactive Java application. It periodically captures call stack samples of all threads running in the virtual machine, and it keeps a ring buffer containing the last few seconds of stack samples. The sampling rate and the buffer size are configurable.

(2) The end-user interface just consists of the “Was Slow!” button. We implement that button in a small Java library, which we link into the application. Depending on the application, we call our library to create the button from within the application’s main method, or we provide an application-specific plug-in to add the button to the application’s user interface. Whenever the user feels that the application behaved sluggish, she can press that button to report it. When the user presses the button, the library performs a JNI native call to notify the FlyBy profiling agent, which stores the ring buffer of samples into a file or sends it back to the developers’ server. (3) The performance failure report classification tool is based on the Trevis framework. It visualizes and clusters the submitted reports. Its goal is to group a large number of submitted bug reports (CCTs) into a small number of clusters, where each cluster corresponds to a specific performance bug. Moreover, Trevis’s intersection approach then summarizes the numerous CCTs in a given cluster into a single representative CCT that the developer can use to identify the cause of the bug.

9. EVALUATION

We configured our FlyBy profiler with a sampling rate of 10ms and a buffer size of 1000 samples, leading to failure reports containing the past 10 seconds of runtime behavior. As our application with perceptible performance problems, we picked one of the most complex interactive Java applications we are aware of: the Eclipse IDE, which consists of over 30000 classes. We produced performance failure reports for three different versions of Eclipse (3.3.2, 3.4.1, and 3.5.2), running on Java 1.5 on Mac OS X Leopard. We installed the FlyBy Eclipse plugin, which provides the “Was Slow!” button, into each of those Eclipse versions. Given that Eclipse, like many other interactive applications, exhibits performance issues especially when it operates on larger amounts of data, we prepared an Eclipse workspace (the data) by importing a Java project containing 40000 lines of code. In this case study, FlyBy measures the performance of the Eclipse IDE, not of the program being developed.

We have not noticed any degradation of perceptible performance due to always-on stack sampling. Periodically capturing call stack traces via JVMTI is relatively cheap, as long as they are not translated from their internal binary representation into a human-readable form. We only need to translate the stack traces that are in the buffer at the time the user submits a bug report. All other stack traces do not need to be translated.
oped in the IDE. However, the IDE’s performance depends on the complexity and size of the project being edited.

In our case study we show how Trevis classifies FlyBy failure reports automatically into clusters that differ by the activity a user was performing. We performed two different activities: normal editing of Java source code, and renaming of classes using the rename refactoring. The latter activity exposes a severe performance bug: Eclipse becomes truly sluggish when typing in rename refactoring mode.

We also performed a case study where we show that FlyBy will cluster reports by application version, if the behavior significantly changed between versions, but we had to omit that study from this paper for space reasons.

9.1 Tree Characteristics

Table 2 quantifies the size of the CCTs produced by the FlyBy profiler in our experiments. The table is based on 40 reports for each activity (40 for renaming, 40 for typing) gathered across three different version of the Eclipse IDE. We can see that the trees are relatively small, thanks to our use of sampling, and our focus on problematic phases of behavior (as determined by us pushing the “Was Slow!” button). These small tree sizes, compared to the several orders of magnitude larger complete CCTs of Moret et al. [9], allow us to perform the computationally expensive analyses provided by Trevis.

The table also shows that the number of “useful” samples per report are well below the 1000 samples maintained in the buffer. This is because Eclipse is an interactive application, which is idle most of the time. We ignore stack samples when the GUI thread was idle. Moreover, in this experiment we focus exclusively on the GUI thread, which performs the vast majority of work.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Renaming</th>
<th>Typing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Useable Samples</td>
<td>Max: 460</td>
<td>Max: 269</td>
</tr>
<tr>
<td>Mean: 377</td>
<td>Mean: 273</td>
<td></td>
</tr>
<tr>
<td>Min: 214(*)</td>
<td>Min: 266</td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>141</td>
<td>141</td>
</tr>
<tr>
<td>Nodes</td>
<td>2569</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>2124</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>1971(310*)</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>948</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>513</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>405</td>
<td>74</td>
</tr>
</tbody>
</table>

Table 2: FlyBy CCT Characteristics

9.2 Activity Recognition

In this case study the user performs two different kinds of activities in the most recent version of Eclipse (3.5.2). He presses the “Was slow!” button to produce a complaint after working on a given activity for roughly 10 seconds. We want to see whether the context trees from the complaints allow us to group the complaints by activity. The first activity is to perform a rename refactoring of a class name. Eclipse supports renaming directly inside the editor, where every occurrence of the specific class name is updated while the user is typing. The second activity is the simple typing of text, where the user implements a new method in an existing Java class. Note that these activities are relatively similar; in both cases the user is typing keys, Eclipse is updating the Java source code and rendering the code to the screen. The difference is that with the rename refactoring, multiple source code locations are updated while typing. We repeated each of these two activities 10 times, and clicked the “Was slow!” button each time.

6Measures marked with a * represent a single outlier detected during the experiments.

Figure 11 shows the dendrogram table Trevis produces when using the Weighted Multiset Node Distance (WMND) measure and the Group Average agglomeration method. We interactively adjusted the dissimilarity threshold to cut the dendrogram into the two main clusters. Given that the activities are quite similar, we were surprised about how clearly the clustering approach can separate between them. The dendrogram shows that the last rename context tree (E3.5-RenameWithBug-01) is quite different from the other rename trees. The reason for this is that this specific tree represents the first occurrence of this activity in the run of the application, which means that it includes initialization activity that is only executed once.

Figure 12 shows the dissimilarity measure has a noticeable impact on the clustering. For this figure, we added a third activity, which is a minor variation of the rename refactoring: During renaming, Eclipse continuously shows a small tooltip. This tooltip slows down the rendering of the text. However, it can be moved away, which eliminates that slowdown. Our third activity (RenameNoBug) consists of renaming, but without that tooltip. Figure 12 compares two different dissimilarity measures in terms of their ability to separate RenameNoBug from RenameBug. We see that WMND (a node distance) mixes RenameNoBug and RenameBug trees. MTD (a structural distance), which considers caller-callee relations, is better able to separate the
two activities. However, this separation comes at the cost of splitting the Typing activities into two distinct clusters. In both cases, though, the clustering shows that the Typing activity is clearly separated from the Rename activities.

Figure 13: Intersection of RenameBug Trees

The dendrogram at the bottom of Figure 12 shows four clusters, each consisting of up to five calling context trees. In a real-world deployment of FlyBy, a cluster might represent hundreds or thousands of user-submitted reports (and thus trees). As a developer, we would like to see one representative tree per cluster, so that we do not have to look at all submitted trees individually. Figure 13 shows such a representative tree for the RenameBug cluster: it corresponds to the intersection of the five trees of that cluster. We can now use this visualization, which represents the common behavior across all trees of the RenameBug cluster, to study the calling contexts where the program spent most time. We find that Eclipse spends a significant amount of time packing the rename tooltip, which corresponds to the cause of the performance problem. Thus, Trevis’ clustering, intersection, and visualization features help us to summarize information from multiple user-submitted bug reports into one compact visualization of the problematic calling contexts.

9.3 Limitations and Threats to Validity

In our case study, we were running the application and filing the complaints. In the real-world usage of FlyBy, the activities would be more diverse, and the clusters might be less clearly separated. Also, our union and intersection approaches do not work well for trees that differ somewhere near the root. Such a difference can occur when comparing trees from an evolving software systems, where methods often are refactored. This is especially true if entire classes or packages are renamed.

10. CONCLUSIONS

Trevis is an extensible framework for visualizing, comparing, clustering, and intersecting trees. It lies at the core of our FlyBy performance bug report classification and visualization tool. FlyBy benefits from Trevis’ ability to compactly summarize a large number of trees by clustering them and by presenting just one exemplar for each cluster.

11. REFERENCES


