Visualizing the Results of a Complex Hybrid Dynamic-Static Analysis

Marc Fisher II
Virginia Tech
Blacksburg, VA, USA
fisherii@cs.vt.edu

Luke Marrs
Virginia Tech
Blacksburg, VA, USA
lmarrs@vt.edu

Barbara G. Ryder
Virginia Tech
Blacksburg, VA, USA
ryder@cs.vt.edu

***Abstract*—Complex static or hybrid static-dynamic analyses produce large quantities of structured data. In the past, this data was generally intended for use by compilers or other software tools that used the produced information to transform the application being analyzed. However, it is becomingly increasingly common for the results of these analyses to be used directly by humans. For example, in our own prior work we have developed a hybrid dynamic-static escape analysis intended to help developers identify sources of object churn within large framework-base applications. In order to facilitate human use of complex analysis results, visualizations need to be developed that allow a user to browse these results and to identify the points of interest within these large data sets. In this paper we present HI-C, a visualization tool for our hybrid escape analysis that has been implemented as an Eclipse plugin. We show how HI-C can help developers identify sources of object churn in a large framework-based application and how we have used the tool to assist in understanding the results of a complex analysis.**

I. INTRODUCTION

Complex static analyses are used for a variety of tasks in program optimization, software testing, and software maintenance. These analyses produce lots of data that is primarily intended for use by other tools that, for example, automatically transform the program or produce test cases. Increasingly the intended consumer of the static analysis results is a programmer or software engineer, who is expected to use the results of the analysis to inform their actions.

Since these analyses produce large quantities of data, it is difficult for users to effectively use the analysis results. Additionally, the type of data produced by these analyses are poorly structured and generally span the entire body of code, a problem exacerbated when analyzing framework-based applications, such as web or service-oriented applications.

In order to make use of the analysis results, effective methods of displaying, searching, and browsing the results is necessary. Such a tool should display the results in a fashion that allows the user to quickly focus in on the important information and connects back to the underlying source code.

In prior work, we have developed ELUDE, a blended escape analysis. For a particular analysis run, ELUDE can produce hundreds of megabytes of XML data that encodes both information about the calling relationships between methods and the reference relationships of objects within each of these methods. The information is intended to guide the user in the process of identifying locations of object churn (excessive creation of temporary objects) so that the corresponding code can be optimized to reduce the number of temporary objects being created and thereby improve performance.

Manually sifting through hundreds of megabytes of XML is obviously infeasible, therefore in the past we have created scripts that attempt to identify the interesting pieces of the analysis results. These scripts filter the analysis results using various metrics such as the number of dynamic instances captured within a particular method context. However, even with these results understanding how the different objects move through the application being analyzed is difficult, time-consuming, and error-prone.

Therefore we have developed HI-C, a tool for exploring the results of our blended escape analysis. The visualization consists of two main views: a representation of the calling structure that provides a high-level view of the temporal aspects of the execution and a representation of the the data structures within the methods displayed in the calling structure.

This paper has the following contributions:

- A description of HI-C, a tool for the exploring the results of a complex blended escape analysis.
- An example that demonstrates how HI-C fits into an analysis work-flow to diagnose a source of object churn within a large framework-based application.
- A description of our use of HI-C to understand and debug our blended analysis.

The remainder of this paper is organized as follows. Section II describes details about HI-C and shows how it fits into an example work-flow. Section III presents details about how we have used HI-C to understand and debug the escape analysis. Section IV describes related work. Section V presents conclusions and future work.

II. EXAMPLE WORK-FLOW

Figure 1 presents an overview of the workflow required to identify object churn using our tool chain. This workflow begins when the user identifies a transaction that is performing poorly. This transaction can be identified from log information, from customer complaints, or through testing. The user is then responsible for constructing an input to the application using information from the identified problem transaction.

The second step of our workflow uses a modified version of JINSIGHT [3], a dynamic profiling tool created by IBM.

Fig. 1. Escape Analysis Workflow

JINSIGHT is able to collect a trace of an execution that includes method calls and returns and object allocations. Conceptually, this dynamic trace can be viewed as a *call tree*, a graph representation that uses a node for each dynamic invocation of a method with edges between nodes that indicate the calling relationships between these invocations . Call trees are generally very large, even for relatively short program runs. More concise representations of the calls can easily be obtained by aggregating nodes in the call tree. For our analysis we use *Calling Context Trees (CCTs)* [1], a popular way to represent dynamic calls. CCTs offer a much more precise representation of the calls in a program than a basic call graph, but without the prohibitively high cost of full call trees. The key idea behind CCTs is to differentiate between calls of a method based on the full invocation stack that resulted in the call. While techniques exist to collect CCTs directly at runtime (e.g., [10]), it is easy to generate them from an existing call tree by aggregating nodes that share the same method sequence from the beginning of the trace. Our extension to JINSIGHT produces a CCT with additional information about object allocations within the nodes of the graph. We then execute the application on the user created input while collecting profiling information with JINSIGHT and output the results of that execution to a CCT file.

The third step of our workflow uses ELUDE [4], [5] to perform a blended escape analysis. ELUDE is based on an escape analysis algorithm developed by Choi *et. al.* [2]. Their escape analysis is similar to reference (i.e., points-to) analysis in that it computes the set of static objects that may be pointed to by each reference in the analyzed program region(s). Escape analysis additionally associates an *escape state* with each static object. The escape state of a static object at a method m indicates if the object is only reachable during executions of m (*captured*), or if it escapes m through a global reference (*globally escaping*) or through parameters and return values (*argument escaping*). The escape analysis within ELUDE differs from the static escape analysis of Choi *et. al.* in that is implemented as a blended analysis. A blended analysis is a static analysis that is focused on a set of executions of interest. In ELUDE this set of executions is represented by the dynamic CCT constructed by JINSIGHT.

As output, ELUDE produces a connection graph for each node in the CCT. The connection graphs show the reference relationships between the the objects visible within the context of a CCT node. Additionally, for each object in the connection graph, it shows the objects escape state within that CCT node.

CHURNI, the fourth step of the workflow, matches the dynamic instance data collected by Jinsight to the results of the static analysis performed by ELUDE. After mapping the dynamic instances to the static objects, reduced connection graphs are built for each context in the CCT. Reduced connection graphs are simple object-to-object graphs that include only those static objects that were mapped to dynamic instances. Additionally, a reduced CCT is also constructed by removing nodes that do not have any captured or escaping instances.

The final step of the workflow is HI-C. HI-C is implemented as an Eclipse plugin. When started HI-C displays the reduced CCT produced by CHURNI as shown in Figure 2(a). Within the CCT, the nodes are color coded to indicate the relative number of dynamic instances captured within the node, with red indicating the node captures large numbers of dynamic instances, orange or yellow an intermediate number of captured instances, and green few or no captured instances.

As can be seen in Figure 2(a), the CCT generally includes too many nodes to effectively navigate and identify the contexts of interest. Therefore, the user presses a button to limit the display to only those nodes that capture large numbers of dynamic instances as shown in Figure 2(b). Prior research has shown that, in most cases, relatively few contexts are responsible for explaining the majority of captured instances [5], [7]. In this case, there are 10 high capturing nodes. Tooltips over the nodes in the CCT indicate the method that each node corresponds to.

When the user selects node 11583, `DateSerializer.getValueAsString()`, in the CCT, the corresponding reduced connection graph is displayed in another pane as shown in Figure 2(c). Within the reduced connection graph, the nodes are color coded to indicate the local and final escape status of corresponding static object. Specifically, the square within the node indicates the local escape status, with red indicating captured within this method (e.g., node 1046), blue indicating argument escaping (e.g., node 191) and green indicating globally escaping. Similarly, the color of the rest of the node body indicates the final escape status, either red for captured (e.g., node 1021) or green for escaping (e.g., node 191).

The user hovers over node 1046 in the connection graph to view information about the corresponding object, including the type (`java.util.GregorianCalendar`), the number of dynamic instances (9), and the maximum capture depth (2). In addition, if the node is the root of a data structure in the context, additional information about the data structure is displayed including the number of types in the data structure (4) and the number of different methods within which the

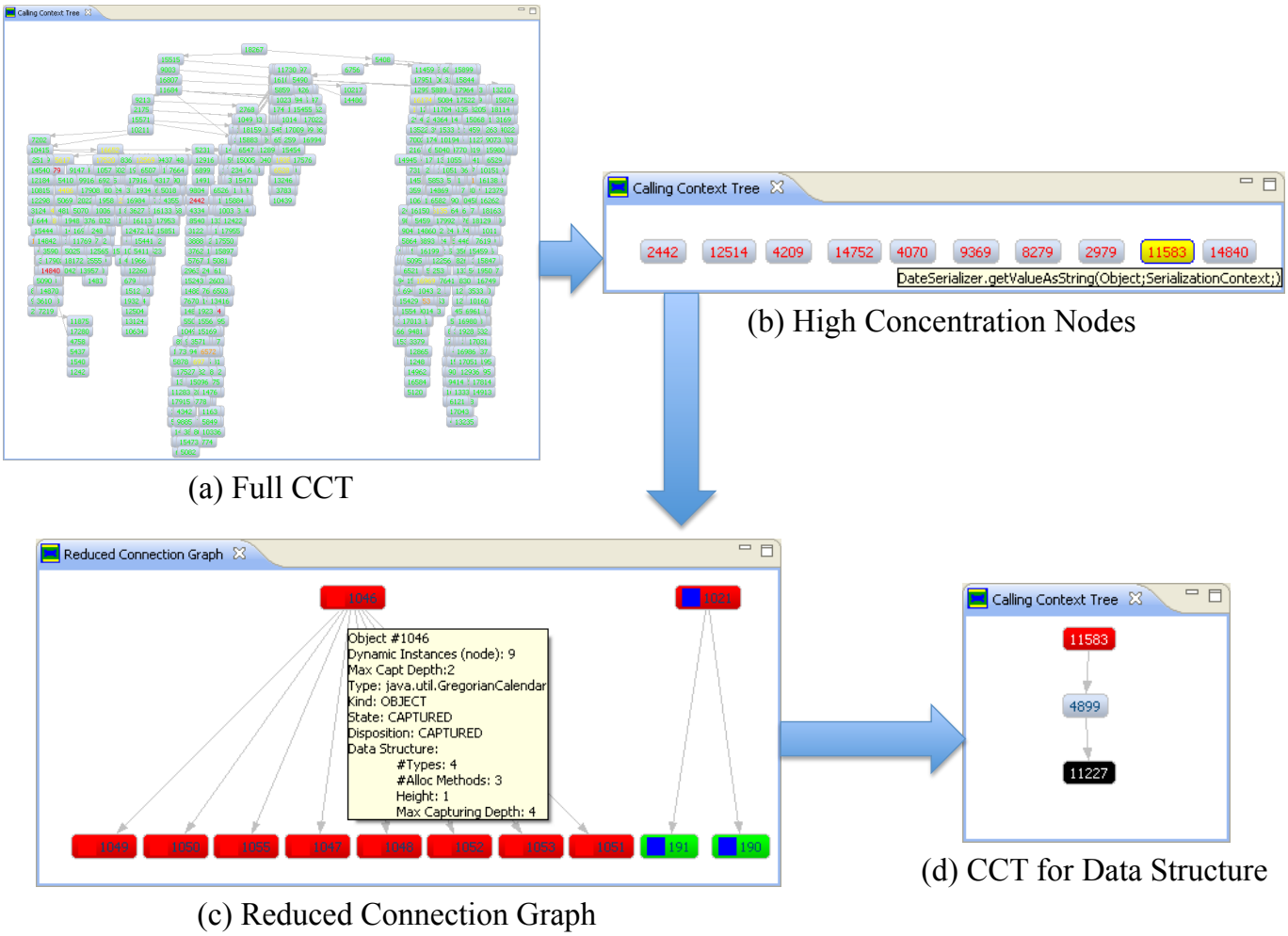

Fig. 2. Example Use of Visualizer

the objects in the data structure are allocated (3). In total 9 `java.util.GregorianCalendar` instances were allocated, which resulted in an additional 108 allocated object instances, mostly `boolean` and `int` arrays.

When the user selects node 1046 in the connection graph, the CCT display changes to show the contexts where the corresponding object is visible as shown in Figure 2(d). In this reduced CCT, the context that allocated the selected object are black (node 11227), the contexts that capture the object are red (node 11583), and any contexts where the object globally escapes are colored green.

In this example, the information shown in HI-C and an understanding of the `java.util.GregorianCalendar` allows the user to determine that by caching the calendar object in the `DateSerializer.getValueAsString()` method they can significantly reduce the number of allocated object instances.

III. Our Usage of Visualizer

In recent work we extended ELUDE and CHURNI to use CCTs as input instead of context-insensitive call graphs (CGs) [7]. The goal of this change was to make the analysis more precise. Additionally, in the process of doing this, we found that although the potential worst case running time of ELUDE was greater for CCTs than CGs, in practice the simpler structure of the CCTs actually improved the running times. However, as we tested these changes and attempted to measure the impact of the changes on the precision of ELUDE, we found several anomalies.

These anomalies were primarily cases where we expected precision to increase or stay the same with a change to the algorithm, but instead it decreased. For example, in one case some dynamic objects had their disposition change from captured to escaped (a decrease in precision) when an optimization called pruning was enabled. While the use of a dynamic calling structure in blended analysis allows us to focus on a subset of the interprocedural control flows, the analysis still processes all intraprocedural control flows, even

though some of these control flows can be shown to not have executed on the run of interest. Pruning allows us to remove some of these intraprocedural control flows by identifying method dispatches with out matching calling structure edges and new statements without matching dynamic instances. For each of these identified statements, pruning removes the smallest enclosing basic block and any control flow edges that lead to that basic block.

With the way pruning was implemented, it should always be the case that any dynamic instance that was identified as captured by an unpruned run should also be identified as captured by a pruned run. However, on a small number of instances we found this to not be the case. We attempted to use HI-C to diagnose this problem. As shown in Figure 2(a), the full CCT tends to be very large. For the particular case we were looking at, the reduced CCTs produced by CHURNI have approximately 2140 nodes. This means that if the node of interest is not one of the high capturing nodes, identifying a specific node in the CCT is very time consuming and difficult.

IV. Related Work

There is a large body of visualization work that could be potentially related to our work on HI-C. For example, many people have visualized data structures or algorithms for pedagogical purposes (e.g. [6]) or created visualizations to identify specific types of problems in programs (e.g. [8]). However there has been relatively little work on providing visualizations of complex static or dynamic analyses targetted toward application developers.

Pheng and Verbrugge's work on dynamic data structure analysis visualizes the changes to the heap during execution as a series of snapshots [9]. These snapshots show the objects on the heap as nodes in a graph with the references between the objects shown as edges in the graph. Within the graphs they show different information about the objects, including age, type, origin (library vs. application code) and reachability, using shapes, colors and labels. This visualization shows information similar to that presented by HI-C. The transitions from one snapshot to the next represent the temporal component of the execution, while we use a CCT as an abstraction for the same thing. The graphs displayed within the snapshots are similar to the connection graphs that HI-C shows. However, the underlying source of the information is somewhat different; HI-C display connection graphs based on a blended analysis that approximates the actual state of the heap during execution while Pheng and Verbrugge's visualization has access to the precise state of the heap from the execution.

Bohnet and Döllner have created visualizations for exploring the dynamic call graph corresponding to the implementation of particular feature. These visualizations group the functions according to the system architecture of the application and attempt to identify the functions that are important with respect to the feature the user is trying to understand. The dynamic call graphs being visualized are similar to our CCTs, and in both cases, due to the size of the graphs, it was important to provide mechanisms to identify interesting nodes in the graphs. However, the criteria used to identify these methods or functions was different due to the different goals of the visualization.

V. Conclusions and Future Work

Complex static analyses produce large amounts of complex, poorly structured data. Extracting the relevant information from this data is difficult, time-consuming, and error-proned. Therefore tool support is required to ease the burden of using these types of analyses. In this paper, we presented HI-C, an Eclipse plugin for visually exploring the results of a complex hybrid static-dynamic escape analysis. We showed how this tool could be used to help developers identify object churn within their applications. We also show how we have used HI-C in our own work debugging the escape analysis.

While using the tool, we have found a number of ways in which the tool can be improved and have learned some general lessons that we feel can be applied to these types of tools. First, the current version of the tool has only limited interactions with the Eclipse IDE, primarily using Eclipse for selecting files and as a basic visualization platform. Increasing the level of interaction with Eclipse, in particular linking the contexts in the CCT view to the corresponding methods in a project and connecting the static objects in the connection graphs to the corresponding allocation sites within the Java source or byte code would make understanding the results of the analysis quicker and easier.

Second, HI-C was designed with the goal of diagnosing object churn in mind, and as such has browsing tools focused on identifying the sources of significant temporary usage. However, when using the tool to debug the escape analysis, it was often the case that problem areas of the analysis lied in corner cases that did not occur frequently, and therefore did not correspond to significant object churn. Adding more robust search features to the tool would have aided in its use in exploring these uncommon cases.

We have several actions that produce reduced version of the CCTs or connection graphs. For these, we selected the set of nodes of interest and displayed only edges that connected these nodes. Unfortunately this had the effect of making it difficult to contextualize the information being displayed as it did not readily map back to the full CCT or connection graph. These reduced displays could be improved by providing additional contextual information.

References

[1] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997, pp. 85–96.

[2] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff, "Stack allocation and synchronization optimizations for Java using escape analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 6, pp. 876–910, 2003.

[3] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of Java programs," in *Software Visualization*, ser. LNCS, 2002, vol. 2269, pp. 151–162.

[4] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analalysis (ISSTA)*, 2007, pp. 118–128.

[5] ——, "A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[6] A. S. Erkan, T. J. VanSlyke, and T. M. Scaffidi, "Data structure visualization with latex and prefuse," *ACM SIGCSE Bulletin*, vol. 39, no. 3, pp. 301–305, 2007.

[7] M. Fisher II, B. Dufour, S. Basu, and B. G. Ryder, "Exploring the impact of context sensitivity on blended analysis," Virginia Polytechnic Institute and State University, Tech. Rep. TR-10-06, April 2010, submitted to ICSM 2010.

[8] C. Parnin, C. Görg, and O. Nnadi, "A catalogue of lightweight visualizations to support code smell inspection," in *Proceedings of the Symposium on Software Visualisation (SoftVis)*, September 2008, pp. 77–86.

[9] S. Pheng and C. Verbrugge, "Dynamic data structure analysis for Java programs," in *Proceedings of the International Conference on Program Comprehension (ICPC)*, 2006.

[10] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, "Accurate, efficient, and adaptive calling context profiling," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 263–271.