HI-C: Diagnosing Object Churn in Framework-Based Applications

Marc Fisher II
University of Memphis
Memphis, TN, USA
marc.fisher@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

In prior work we have developed an escape analysis to help developers identify sources of object churn (i.e., excessive use of temporaries) in large framework-based applications. We have developed HI-C, an Eclipse plug-in that allows users to visualize, filter, and explore analysis results to aid them in diagnosis of object churn and in program comprehension in general.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*

General Terms

Measurement, Performance

1. INTRODUCTION

In prior work, we developed ELUDE, a tool that performs a blended escape analysis [4]. For a particular analysis run, ELUDE produces hundreds of megabytes of complex XML data. This data represents information at two levels of granularity – calling relationships between methods during the execution and reference relationships between objects within these methods; each level refers to the other in complex ways. For example, each object[1] includes information about the method(s) in which it was allocated or captured or from which it escaped. This information is intended to guide the user in the process of identifying locations of object churn (i.e., excessive creation and initialization of temporary objects) so that the performance of the corresponding code can be improved.

Due to their size and complexity, manually sifting through these results is infeasible. Therefore, we used a combination of specialized scripts and manual inspection to explore the analysis results. These scripts produce static views of the data such as ranked lists of potentially interesting methods and objects, graphs and basic statistics. Identifying interesting objects or understanding how objects flow through an execution using such reports required manually jumping between them and mentally managing links between the different summaries, something we found to be difficult.

In order to ease this burden of exploring and understanding analysis results, we built HI-C, an Eclipse plugin that provides interactive, dynamic visualizations. HI-C presents two views: (i) a calling context tree (CCT) representing calling relationships between methods [1] and (ii) a connection graph (CG) representing reference relationships between objects within a particular method [4]. Unlike previous reports, these views provide mechanisms for quickly identifying how the objects represented in the CG flow through the execution represented in the CCT. Thus, HI-C provides an example of how to help a developer understand and explore complex dataflow information.

[1] We use the term *object* to refer to the static representation of a run-time object in the dataflow formulation and *instance* to refer to a run-time object. For our analysis, an object is an allocation site within a particular method.

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
ACM 978-1-60558-791-2/10/11.

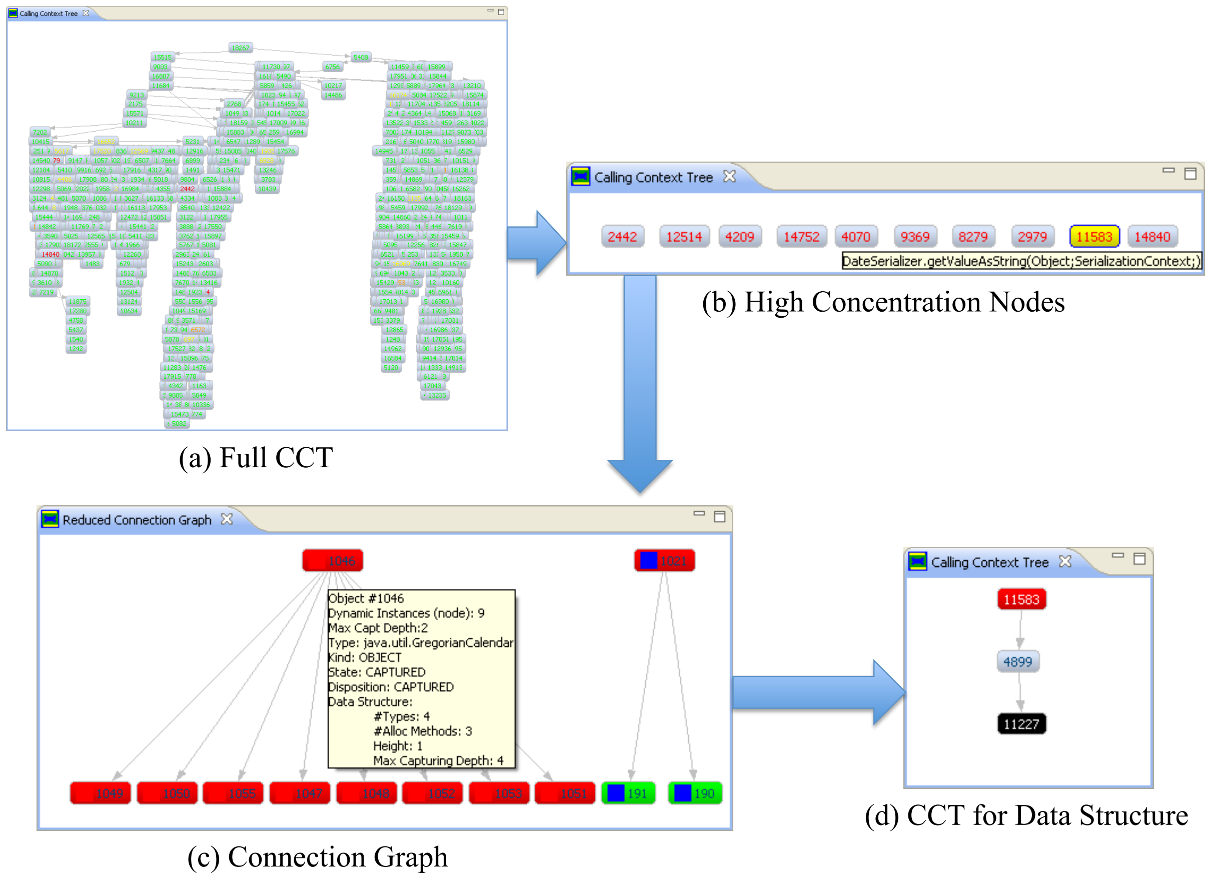
2. TOOL DESCRIPTION

HI-C is the last stage in a workflow that uses a variety of analysis tools to identify areas of high churn within an application. This workflow begins when the user identifies a transaction that is performing poorly. The user then uses JINSIGHT [3] to collect a dynamic trace of the execution, ELUDE [4] to perform the escape analysis, and CHURNI to merge the output of ELUDE with allocation information from JINSIGHT to produce a CCT and CGs.

HI-C, implemented as an Eclipse plug-in, allows exploration of the output produced by CHURNI. When started, HI-C displays the CCT produced by CHURNI as shown in Figure 1(a). Each node in the CCT represents a set of calls to a method (a context) and each edge indicates calling relationships between these contexts. Within the CCT, each node is color-coded to indicate the relative number of instances captured within its corresponding context, with red indicating the capture of many instances, orange or yellow, an intermediate number of captured instances, and green, few or no captured instances.

As can be seen in Figure 1(a), the CCT generally includes too many nodes to effectively navigate and identify the contexts of interest. Therefore, the user can press a button to limit the display to only those contexts that capture large numbers of instances as shown in Figure 1(b). Prior research has shown that, in most cases, relatively few contexts are responsible for explaining the majority of captured instances [4]. Tooltips over the nodes in the CCT indicate the method corresponding to each node.

When the user selects a node in the CCT, the corresponding CG is displayed in another pane as shown in Figure 1(c). Each node in the CG corresponds to an object used in the escape analysis and each edge indicates a reference relationship between these objects. Within the CG, the nodes are color-coded to indicate the escape status of the corresponding object. Specifically, the square within the node indicates the local escape status (i.e., is the object captured in or does it escape from the current context), with red indicating captured within this context (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 (i.e., is there any context where the object globally escapes), either red for captured (e.g., node 1021) or green for escaping (e.g., node 191). The tooltips for the nodes in the CG in-



DateSerializer.getValueAsString(Object;SerializationContext;)

Object #1046
Dynamic Instances (node): 9
Max Capt Depth:2
Type: java.util.GregorianCalendar
Kind: OBJECT
State: CAPTURED
Disposition: CAPTURED
Data Structure:
#Types: 4
#Alloc Methods: 3
Height: 1
Max Capturing Depth: 4

(a) Full CCT

(b) High Concentration Nodes

(c) Connection Graph

(d) CCT for Data Structure

Figure 1: Example Use of HI-C

clude additional information about the objects, including type, the number of instances, and the maximum capture depth of the object.

When the user selects a node in the CG, the CCT pane changes to show the contexts where the corresponding object is visible as shown in Figure 1(d). In this reduced CCT, the context that allocated the selected object is black (e.g., node 11227), the contexts that capture the object are red (e.g., node 11583), and any contexts where the object globally escapes are colored green.

More details about HI-C are available in reference [6].

3. RELATED WORK

While many people have visualized data structures or algorithms for pedagogical purposes (e.g., [5]) or created visualizations to identify specific types of problems in programs (e.g., [7]), 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 [8]. The graphs displayed within the snapshots are similar to the connection graphs that HI-C shows, while the sequences of the snapshots themselves provide temporal data similar to that provided by the CCT.

Bohnet and Döllner have created visualizations for exploring the dynamic call graph corresponding to the implementation of a particular feature [2]. 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 were not the same due to the different goals of the visualization.

Acknowledgements.

The authors would like to thank Gary Sevitsky and Bruno Dufour who were involved in the development of blended escape analysis and of many of the tools used in this work. This work was supported in part by NSF Grants CCF-0811518 and CCF-0964870 and by an IBM Open Collaboration Award.

4. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Conf. on Programming Language Design and Implementation*, pages 85–96, 1997.

[2] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Symp. on Software Visualisation*, pages 95–104, September 2006.

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

[4] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Inter. Symp. on the Foundations of Software Engineering*, 2008.

[5] A. S. Erkan, T. J. VanSlyke, and T. M. Scaffidi. Data structure visualization with LaTeX and prefuse. *ACM SIGCSE Bulletin*, 39(3):301–305, 2007.

[6] M. Fisher II, L. Marrs, and B. G. Ryder. Visualizing the results of a complex hybrid dynamic-static analysis. Technical Report TR-10-07, Virginia Tech, April 2010.

[7] C. Parnin, C. Görg, and O. Nnadi. A catalogue of lightweight visualizations to support code smell inspection. In *Symp. on Software Visualisation*, pages 77–86, September 2008.

[8] S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *Inter. Conf. on Program Comprehension*, 2006.