Extracting User-Level Functions from Object-Oriented Code

Neil Walkinshaw, Marc Roper, Murray Wood

Department of Computer and Information Sciences,
The University of Strathclyde, Glasgow G1 1XH, UK
E-mail: {neil.walkinshaw, marc, murray}@cis.strath.ac.uk

Abstract

Several software (re)engineering, comprehension and verification tasks rely on the ability to extract the source code that is relevant to a user-level function. In an object-oriented system this code can be distributed across multiple classes, which makes it very difficult to trace. We have developed a tool-supported approach that addresses this issue. Given a set of landmark methods that must be executed in a given user-level function (or use-case), the tool uses slicing and call graph analysis to return a trail of the most relevant methods. The success of this approach hinges on the selection of landmark methods. This is a short position paper that presents our technique, some conclusions from a preliminary evaluation and some research avenues that we aim to pursue in the future.

1. Introduction

It is common for software (re)engineering and maintenance tasks to follow a policy of divide-and-conquer. As an example, in software inspections the system has to be divided into manageable chunks that can be inspected by individual inspectors [4]. As another example one of the goals of software re-engineering is to 'unbundle' a system into parts that can be marketed individually [1]. Dividing an object-oriented system on a structural basis (e.g. class-by-class) is simple but does not account for dependencies that may arise when the system is executed. Most tasks necessitate the isolation of source code that is relevant to a particular user-level system function, which may be spread across multiple classes.

Determining the behaviour of object-oriented code from a static presentation of the source code is very challenging and time consuming. Paradigm features such as inheritance, small methods, polymorphism and dynamic dispatch mean that the type of an object (the class containing the method of interest) can often not be determined statically. Tracing along every path in a non-trivial object-oriented system becomes practically infeasible because every permutation of run-time object types produces a different combination of paths that need to be taken into account. Figure 1 illustrates some of these problems.

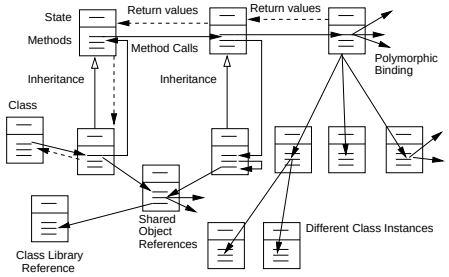


Figure 1. Challenges of Understanding the Behaviour of an Object-Oriented System

In an ideal situation specification documents could be used as a guide through the code, removing much of the manual overhead. In reality however they are rarely detailed enough to accurately map to the source code [1, 2]. They often only feature key interactions and are not maintained accurately as the system evolves. With legacy systems this problem tends to be amplified. Inaccurate specification documents place an enormous overhead on manually reading the source code because it is left to the reader to intuitively determine what source code is related to the program points they have been able to directly link with the specification.

We have developed a code-extraction technique for object-oriented systems [7] that uses a limited amount of information about the execution of a particular user-level function in the form of "landmark methods". These are methods (perhaps from the system specification) that *must* be executed at run-time. Based on these methods the system uses a combination of call graph analysis and slicing [8] to reduce the size of the call graph, focussing on edges that are particularly relevant.

1. **Select landmark methods in call graph**
2. **Identify direct paths between landmark methods:**
   (a) Induce hammock graphs on the call graph between every pair of landmark methods
3. **Identify paths that can influence and be influenced by the paths in the hammock graphs:**
   (a) Identify call statements for every edge in the hammock graphs
   (b) Generate intra-procedural slices, using call statements as slicing criteria
   (c) Mark all calls belonging to the slices
   (d) Follow all paths in the call graph originating from the marked call sites

Figure 2. Process of extracting code relevant to a particular aspect of system functionality

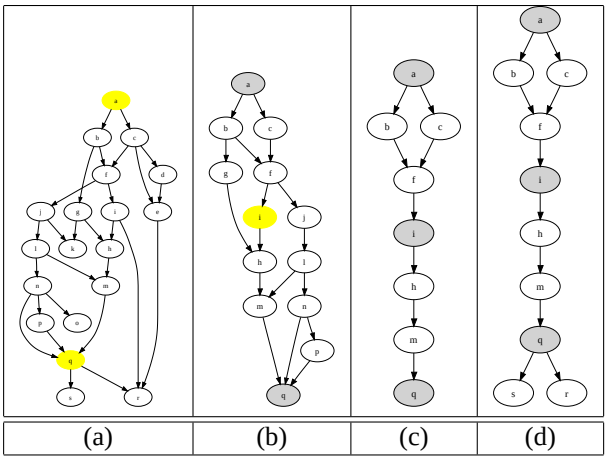


Figure 3. Trimming the call graph by inducing hammock graphs between landmark methods

2. Using Landmark Methods to Divide Object-Oriented Systems

Our technique returns a subgraph of the call graph that is relevant to a particular user-level function (or use case). Figure 2 contains an overview of our approach. The use case is characterised by a set of 'landmark methods', which *must* be executed. The resulting graph contains a set of edges that are particularly relevant to the execution of this set of landmark methods. Here we present an overview of how we obtain this graph. For a more in-depth description of the technique and its implementation the reader is referred to [7].

Once landmark methods have been identified, hammock graphs [3] are induced on the call graph[1]. Hammock graphs contain a single entry and a single exit node, and all of the paths in the graph lead from the entry node to the exit node. By identifying hammock graphs we are highlighting all of the edges in the call graph that belong to a direct path between two landmark methods.

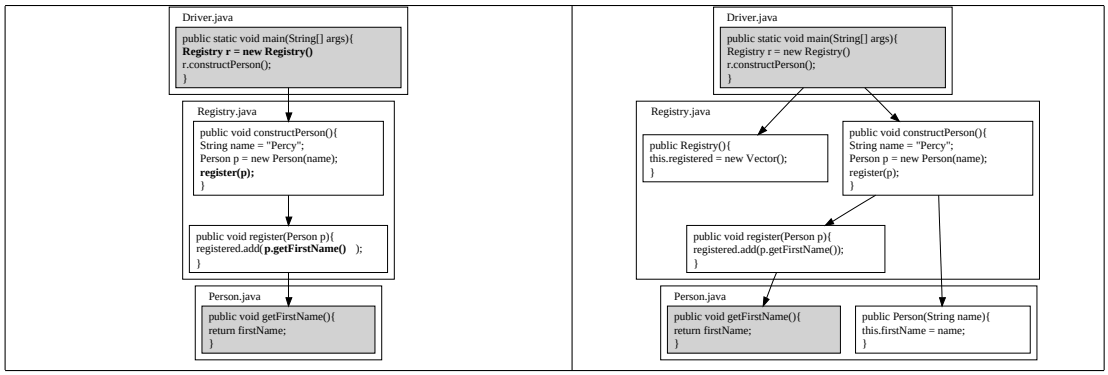
Figure 3 illustrates how to identify hammock graphs on a simple graph. Graph (a) shows the entire graph, with nodes a and q highlighted as landmark methods. A hammock graph between a and q is obtained by intersecting the set of edges that precede q and succeed a. The result is shown in figure (b). Figure (c) illustrates how the call graph can be split, by marking node i as a landmark method. Here the process is repeated by computing two hammock graphs, where i is the exit node for one and the entry node for the other. Because no landmark methods succeed q, we have to add all of q's successors to the list of calls to be read. The final result is represented in (d).

The edges contained in the hammock graphs currently identify the calls on the call graph that directly link landmark methods. Only following paths that *directly* connect landmark methods is not sufficient. Simply because there is not a direct path between two methods in the call graph does not mean that one cannot influence the execution of the other.

An example is provided in figure 4. The hammock graph between the methods main and getFirstName is shown in (a) (note that these methods belong to different classes). The information provided by the hammock graph alone is insufficient. To be thorough we would want to know how firstName is initialised in object p and how the Registry object is initialised. This would require the scrutiny of the Person and Registry constructors, which are not part of the hammock graph in (a).

We identify these relevant paths by using call sites in the hammock graphs as slicing criteria to identify call sites for relevant indirect calls (marked bold in (a)). A backwards slice on a slicing criterion (a statement and a set of variables in that statement) returns the set of statements that may influence the execution of the slicing criterion [8]. When slicing backward we use the call statement with the call arguments as slicing criteria (the destination object containing the called method counts as a call argument). Edges belonging to paths out of these call sites can be added to the final body of code to provide a self-contained unit. Any

[1] It should be noted that the initial node in the call graph is by default always a landmark method.



(a) hammock graph between main and getFirstName | (b) hammock graph from (a) with dependences

Figure 4. Adding dependences to hammock graphs

call sites that belong to the slices and are not the source of an edge in a hammock graph are marked. Marked calls are significant because we know that (a) they may be executed at run-time and (b) if they are executed, they can influence the execution of methods belonging to the hammock graph. If a marked call site is not succeeded by any landmark methods, we cannot restrict the path that would occur if it were executed. To provide a conservative estimate of the code that is relevant, all call graph edges that can be transitively reached by that callsite must be taken into account.

In figure 4, the call sites r.constructPerson(), register(p) and p.getFirstName() (the call sites that spawn edges on the hammock graph) are used as slicing criterion points. Variables representing parameters and destination objects are used as criterion variables (r and p). Intra-procedural slices on these criteria contain the calls Registry r = new Registry() in main, Person p = new Person() in constructPerson and registed.add(...) in register. If there is a call to a library method we do not add it to the paths to be inspected, because we currently treat library calls as being beyond our scope of interest. registered.add(...) is a library method (the Vector.add(Object) method in Java). The Person and Registry initialisers are however application methods so they need to be taken into account. If they were to call any further application methods (they do not in this example), these method calls would have to be traced through the call graph.

3. Implementation and Evaluation

We have developed a tool that can be used as a basis for evaluating our approach on sample software applications. It uses the Soot byte code analysis framework [5] to extract the call graph and dependence information. Soot operates on byte code, so slices have to be mapped back to the source code. Graphs are traversed and visualised using the Java Universal Network/Graph (JUNG) framework[2].

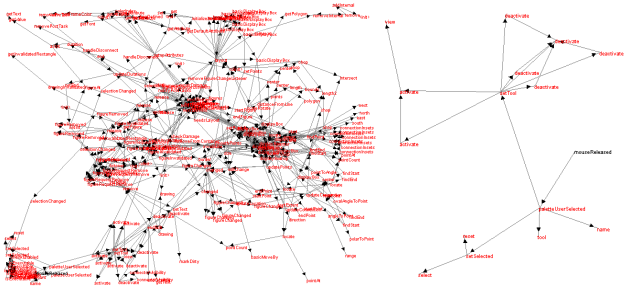


Figure 5. Call graphs from JHotDraw, left is unrestricted and right is restricted by four landmark methods

As an example, a JHotDraw[3] use-case specification accompanied by a set of sequence diagrams was produced by an individual who has expert knowledge of the system. It covers the selection of a drawing tool and commences when the user releases the mouse over a tool button. We use the mouseReleased method (implementing the mouseListener interface) as the entry point for the call graph. The unrefined call graph contains 251 vertices (methods) connected by 719 edges (potential method invocations) and is shown on the left in figure 5.

The problem of distributed functionality that has to be dealt becomes apparent when we consider that every method in the call graph is executed as part of at least one use case. Although it is possible to read the 251 methods individually, reading each method in every possible context of execution becomes infeasible. For this reason we need to apply our technique of reducing the call graph.

[2] http://jung.sourceforge.net/
[3] http://www.jhotdraw.org/

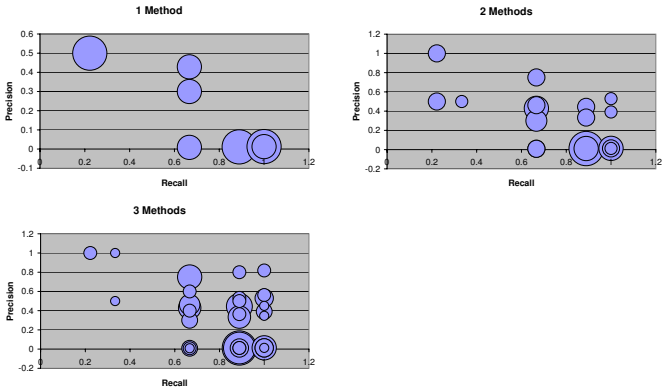


Figure 6. Landmark combinations for "select tool" use-case in JHotDraw

The reduced graph that corresponds to the supplied use-case is shown in on the right in figure 5 and was obtained by using four landmark methods. The graph is reduced to 16 vertices, connected by 20 edges, returning two superfluous edges because of a polymorphic call that is not restricted by landmark methods. This illustrates how the approach can be used to focus attention on a selection of interactions that may be of particular interest to the user.

Preliminary observations have shown that the quality of the results is dependant on the combination of landmark methods. The problem we are faced with is determining how to choose landmark methods to produce an accurate result. To gain an insight into what constitutes a "good" combination of landmark methods, we have modified our tool. By determining the set of edges that are relevant in advance, the tool automatically produces every possible combination of landmark methods (we determine the number of methods per combination). This allows us to compare the results returned by the automated combinations to our ideal graph.

Each graph produced by the automated landmark method combinations is compared to our ideal graph and evaluated in terms of precision and recall [6], an approach commonly used to evaluate information retrieval techniques. Precision denotes the proportion of retrieved edges that are actually relevant and recall denotes the proportion of relevant edges that are actually retrieved. For each use case these results can be visualised as a bubble chart, where the size of the bubble indicates the number of landmark method combinations that produce an identical precision and recall result. An example is shown in figure 6.

By using this approach to study several use-cases from two systems (JHotDraw and a hotel administration system developed for an undergraduate class), we have produced three observations for landmark method combinations:

1. Increasing the number of landmark methods can, depending on the use-case, substantially increase precision and recall.

2. Landmark methods can be used to reduce the number of candidate destinations for polymorphic calls.

3. If a landmark method is executed as the result of a condition, other method calls controlled by that condition will only be included if they are also marked as landmark methods.

4. Conclusions and Future Work

We have produced a technique that is very effective at focussing attention on a group of interactions between classes and their methods. By removing a large proportion of methods and method calls that are irrelevant, the substantial overhead involved in manually determining which method calls are relevant can be significantly reduced. We propose that this will be a particularly useful tool for any software (re)engineering task that necessitates the partitioning of an object-oriented system into smaller units, which preserve the context in which they are executed.

The effectiveness of our approach rests on the selection of suitable combinations of landmark methods. Our current research aims to produce a set of guidelines for their selection. We have produced a tentative set of observations about the properties of suitable combinations and hope to extend this in the future.

So far we have concentrated on the use of a set of landmark combinations that are selected a priori. During our preliminary evaluation we have realised that the use of landmark methods in a more interactive manner would be particularly useful for exploring source code on a more ad-hoc basis. To experiment with this idea the tool has been modified to enable both a priori landmark method selection and the interactive specification of landmark methods. As landmark methods are specified by the user, feedback is provided instantly by highlighting in yellow the calls that belong to the call-graph returned by our approach.

The underlying rationale for the development of this technique was to address the problem of inspecting code that corresponds to use-cases which is inherently distributed in object-oriented systems. To validate our approach with respect to this application we will need to qualitatively and quantatively compare it with other established code extraction and (re)engineering tools and techniques. Besides producing guidance for the placement of landmark methods, we will also need to produce an inspection-specific reading technique that will take the code produced by this approach and encourage the detection of faults in the code.

References

[1] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. ESPIRIT 21975, 1999.

[2] A. Egyed. A scenario-driven approach to traceability. *IEEE Transactions on Software Engineering*, 29(2):123–132, 2003.

[3] V.N. Kas'janov. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady*, 16(5):448–450, 1975.

[4] D. Parnas and M. Lawford. The role of inspection in software quality assurance. *IEEE Transactions on Software Engineering*, 29(8):674–676, August 2003.

[5] R. Vallee, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimisation framework. In *Proceedings of Cascon '99*, pages 125–135, 1999.

[6] C.J. van Rijsbergen. *Information Retrieval*. Butterworth-Heineman Newton, 1979.

[7] N. Walkinshaw, M. Roper, and M. Wood. Understanding object-oriented source code from the behavioural perspective. In *Proceedings of the International Workshop on Program Comprehension (IWPC'05)*, St. Louis, May 2005. IEEE.

[8] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.