Understanding Object-Oriented Source Code from the Behavioural Perspective

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

Comprehension is a key activity that underpins a variety of software maintenance and engineering tasks. The task of understanding object-oriented systems is hampered by the fact that the code segments that are related to a user-level function tend to be distributed across the system. We introduce a tool-supported code extraction technique that addresses this issue. Given a minimal amount of information about a behavioural element of the system that is of interest (such as a use-case), it extracts a trail of the methods (and method invocations) through the system that are needed in order to achieve an understanding of the implementation of the element of interest. We demonstrate the feasibility of our approach by implementing it as part of a code extraction tool, presenting a case study and evaluating the approach and tool against a set of established criteria for program comprehension tools.

Keywords: *slicing, hammock graphs, behavioural comprehension*

1. Introduction

Many software engineering activities such as maintenance, testing and inspection rely heavily on the use of effective comprehension techniques. As an example, if a programmer has to perform a maintenance task on a software system (which they may not have written themselves), they need to understand it to determine which element(s) of the system are relevant. If the task results in a change, they also need to understand how it will affect the rest of the system.

In object-oriented systems, the object is the primary and sole unit of decomposition. Objects model important entities within a system and consist of data tightly bound with the methods that manipulate them. The system executes by passing messages (invoking methods) between objects. A consequence of this decomposition strategy is that function and structure are no longer coincidental: functionally related code is distributed over many different objects.

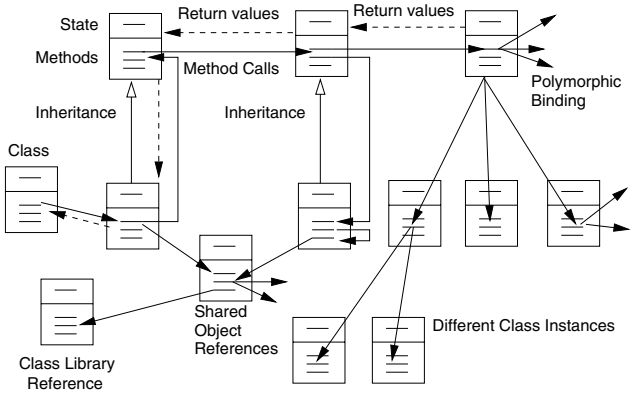


Figure 1. Challenges of Understanding the Behaviour of an Object-Oriented System

Understanding 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. The problem is summarised succinctly by Gamma *et al.* [10]: *"In fact, the two structures [run-time and compile-time] are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals"*. Figure 1 illustrates some of these problems.

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 [5]. 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.

Manually comprehending object-oriented systems from the dynamic perspective has been investigated by Robillard *et al.* [17], who investigate comprehension with respect to a software change task, Dunsmore *et al.* [4] and Thelin *et al.* [20], who both investigate reading techniques to support comprehension for software inspections. Robillard *et al.* conclude that a systematic approach is required to compensate for the fact that code artefacts related to a single change task are spread across the system. In their previous work [18] they provide a good example of the distribution of functionally related code, stating that trying to modify the conditions under which logging occurs in Jakarta Tomcat would entail the consideration of 32% of the system's Java source files. Dunsmore *et al.* stress the need to reduce the manual overhead involved in tracing relevant source code elements across classes via some form of tool support. This is the rationale for the work presented in this paper, which aims to significantly reduce the expense inherent in reading through functionally related code by removing the need to manually determine what is relevant.

We present an approach that uses program slicing to determine method calls that are relevant to a (potentially limited) set of 'landmark methods' that must be executed for a given use-case scenario that could be traced from the specification. A use-case is a description of a set of sequences of actions, including variants, that a system performs that yields an observable result of value to an actor [3]. A scenario is a particular instantiation of a use-case. Our solution provides a manageable amount of code to read and understand for a given use-case. We restrict the call graph to provide only those methods and invocations that are related to the use-case (or use-case scenario) under consideration. We trim the source code base to make it more manageable for the code reader as follows:

1. Elicit points, called landmark methods, from the specification that we know will be executed for a given use-case and map them to the source code.

2. Induce hammock graphs on the call graph between these points (hammock graphs are defined in section 2.2).

3. Use the calls in the hammock graphs as slicing criteria in order to mark further calls whose executions influence the computation of methods belonging to the hammock graphs.

4. Expand paths from calls marked by step 3.

The next section provides an overview of code slicing and provides definitions. Section three presents our approach. Section four contains an evaluation and details of our implementation. Section five surveys related work and section six provides conclusions and directions for future work.

2. Background

2.1. Slicing

Slicing is a technique used to highlight statements that are relevant to a particular computation. Since it was introduced by Weiser in his thesis in 1979 [24], it has grown into an active field of academic research. In abstract terms, a slice presents a fragment of the program, consisting of statements that are semantically related to some slicing criterion specified by the user.

There exists a plethora of slicing approaches to suit different software maintenance tasks. The approach we propose uses static intra-procedural backward [25] and forward [12] slices. For a comprehensive introduction on how to compute these slices, the reader is referred to either Tip's [21] or Binkley and Gallagher's [1] slicing overview.

A static *backward* slice answers the question: *"What statements can affect the behaviour of a variable v at a point p?"*. The standard criterion format for a slice is denoted <*p,v*>, where *p* denotes the point of interest in the program and *v* denotes the variable. Usually it is assumed that *v* is either defined or used at *p*.

A static *forward* slice answers the question: *"What statements contain variables that can be affected by a variable v at a point p?"*.

There are two popular approaches to computing a slice. Weiser's original approach is an iterative algorithm that computes the slice as the solution to a set of data-flow equations [25]. The other approach is based on representing the program as a graph where vertices represent expressions and edges represent different types of inter-dependences [8]. Our approach uses dependence graphs to represent individual methods (the graphs are referred to as Method Dependence Graphs or MDGs).

We avoid the problematic issue of calling-context that can cause inter-procedural slices to be imprecise [12] because we use *intra*-procedural slices. To construct an accurate intra-procedural slice within an object-oriented system, we do however still rely on *inter*-procedural data flow information which is necessary to compute transitive dependences caused by method calls. In procedural systems, a transitive dependence occurs when the argument to a method affects the value it returns [12]. In object-oriented systems a transitive dependence also occurs if the call modifies the state of the target object. A transitive dependence is represented (on the dependence graph) by a summary edge

between the actual-in and actual-out vertices of a call. Reps *et al.* [16] provide an efficient approach for their computation.

2.2. Definitions

A *digraph* (or *directed graph*) D is a structure $< N, E >$, where N is a set of nodes and E is a set of edges. A *path* from *n* to *m* of length *k* is a list of nodes $p_0, p_1, ..., p_k$ such that $p_0 = n$, $p_k = m$ and for all i $1 \leq i \leq k-1$, (p_i, p_{i+1}) is in E. A node n_x *precedes* a node n_y (and n_y *succeeds* n_x) if there exists a path from n_x to n_y. If they are adjacent then n_x *directly precedes* n_y. The set of nodes preceding n_y on a graph D is denoted as $Pre(n_y, D)$ and the set of nodes succeeding n_x is denoted as $Succ(n_x, D)$.

A *flow graph* F is a structure $< N, E, n_0 >$, where $< N, E >$ is a digraph and n_0 is a member of N such that there is a path from n_0 to all other nodes in N. We will refer to n_0 as the initial node. If m and n are two nodes in N, m *dominates* n if m is on every path from n to n_e.

An object-oriented call graph C is a flow graph that represents the call relationships between methods. The node set N represents the set of methods (every method is represented by a single node), E represents the set of calls between methods and n_0 is the entry point to the system. A call edge e takes the form $e = c \rightarrow m$, where c is the call site (statement where the call originates) and m is the target method.

The original definition of a hammock graph is provided by Kas'janov [13]. In the context of static program analysis it is commonly defined with respect to the control flow graph of a method (e.g. Weiser [25] and Ferrante *et al.* [8]). In our work, Kas'janov's notion of a hammock graph remains the same, but instead of referring to hammock graphs in the context of CFGs, we refer to them in the context of call graphs. Our definition for hammock graphs is the same as Weiser's.

A *hammock graph* H is a structure $< N, E, n_0, n_e >$ such that $< N, E, n_0 >$ and $< N, E^{-1}, n_e >$ are both flow graphs, where $E^{-1} = \{(a, b) | (b, a) \in E\}$.

We will refer to the information we garner from the specification in terms of a set of landmark methods $M = \{m_0, ..., m_n\}$. Methods can only belong to M if they can be traced from the specification to the source code.

3. Using Dependence and Specification Information to Produce a Reduced Call Graph

Because of the large edge-to-vertex ratio in object-oriented call graphs, it is difficult to determine the context in which a method might be called during the execution of a given scenario. Here we suggest an approach that uses any available information about the execution of the scenario to limit the number of contexts in which a set of methods may be called. This divides the call graph into self-contained segments that can be read individually, following a 'divide and conquer' policy. The process of extracting relevant code is outlined in figure 2.

1. **Trace landmark methods from specification to call graph**
2. **Identify direct paths between traced methods:**
   (a) Mark methods traced from the specification on the call graph
   (b) Induce hammock graphs on the call graph between every pair of traced 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

We identify relevant paths through the system by making use of information from the specification. Our approach caters for the realistic situation that the specification may be incomplete, but that any scenario contains a set of landmark method invocations M that can be traced to the source code. We create a chain of hammock graphs between the methods in M, isolating calls on direct paths between the methods. We then use their call sites (see section 2.2) as slicing criteria to detect calls that do not belong to paths in the hammock graphs but can still influence or be influenced by their execution.

3.1. Obtaining Hammock Graphs

A hammock graph H induced on a call graph C between methods l and m is denoted $H(C, l, m)$ where $H(C, l, m) =< N, E, l, m >$ and $H(E) \subseteq C(E)$, where $H(E)$ denotes the edges belonging to H and $C(E)$ denotes the edges belonging to C. $H(C, l, m)$ is a vertex-induced subgraph of C, which contains the nodes that belong to the

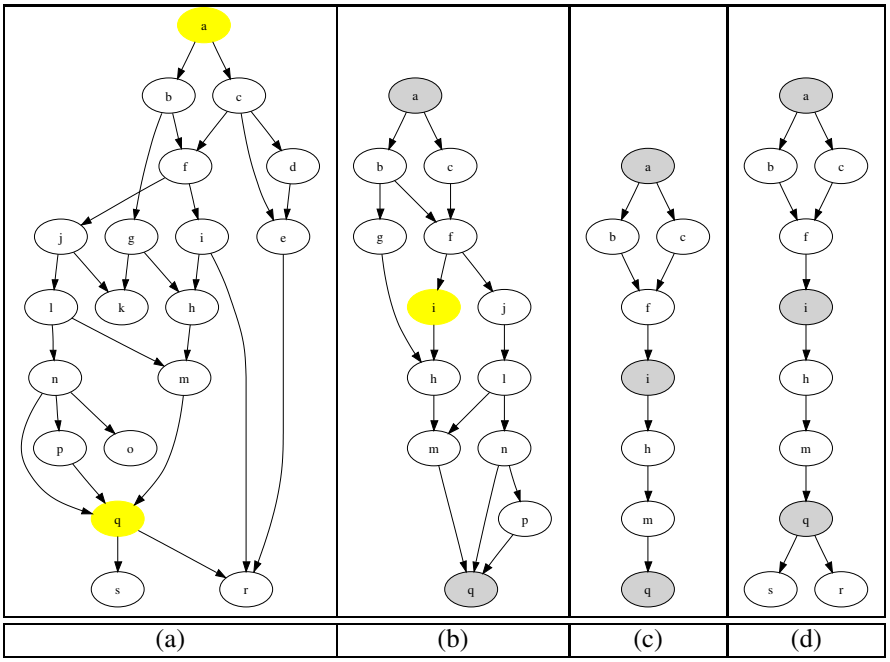


Figure 3. Trimming the call graph by inducing hammock graphs between landmark methods

intersection of $Succ(l, C)$ and $Pre(m, C)$ (see section 2.2) and the edges that connect them in C.

We use the set of landmark methods M garnered from the specification to eliminate as many superfluous call graph edges as we can. Methods elicited from the scenario specification act as landmarks in the call graph; every path to be understood must pass through them. A hammock graph $H(C, l, m)$ contains all call edges that may belong to a path from l to m. To restrict paths through the call graph to those that execute landmark methods, we produce a hammock graph for every pair of methods in $M \cup m_0$ where m_0 is the entry method to the system.

An example of how to induce a hammock graph on a call graph is provided in figure 3. Here the nodes $C = \{a, ..., s\}$ represent the vertices of the entire call graph, $a = m_0$ and $\{q, i\} \in M$. Figure (a) shows the entire call graph, with nodes a and q highlighted. Figure (b) shows the call graph obtained by inducing the hammock graph $H(C, a, q)$. Figure (c) shows the hammock graphs that are obtained when we divide the graph from (b) by using node i as an additional landmark method. Because i succeeds a and precedes q on the call graph, we can split the graph $H(C, a, q)$ into $H(C, a, i)$ and $H(C, i, q)$.

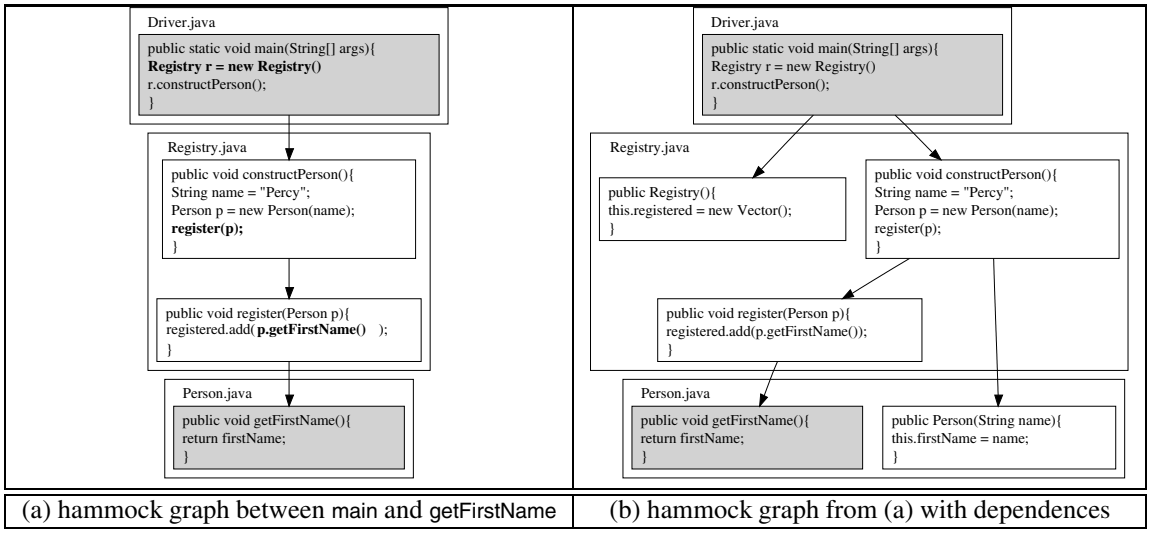
Assuming we do not know what happens after the execution of method q, we have to add all of q's successors to the list of calls to be inspected. The final result is represented in (d). This is what is used as the basis for computing the path dependencies, as described in the following section.

3.2. Computing Dependencies of Hammock Graph Paths

The edges contained in the hammock graphs currently identify the calls on the call graph that directly link methods belonging to $M \cup m_0$. Following paths that only *directly* link landmark methods is not sufficient. Simply because there is not a direct path between two methods in the call graph does not mean that they cannot be executed as part of the same execution.

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 slicing criterion consists of a program point (usually a statement) and a set of variables [25]. When slicing backward, we use the arguments for the call as variables for the slicing criterion. When slicing forward we use variables that are assigned a value by the call and variables representing objects that have had their state changed by the call for the criterion. Edges belonging to paths out of these call sites can be added to the



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

Figure 4. Adding dependences to hammock graphs

final body of code to provide a self-contained unit.

A callsite can be derived from a call graph edge because a call takes the form $callsite \rightarrow method$ (see section 2.2). For every callsite belonging to an edge in a hammock graph, we produce intra-procedural backward and forward slices. Any 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 or be influenced by 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 (e.g. actual-in vertices belonging to these call sites in the MDG) 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.

3.3. Restricting Path Dependencies

By taking both forward and backward slices from the hammock graph call sites into account, we are answering the following question: *"What are the paths on the call graph that can either affect or be affected by a set of methods M?"*. Depending on the system being analysed and the set of landmark methods, the code base that is extracted using this approach may still be too large to be of practical use.

Given the same hammock graph, we can further restrict the code base by being more specific about the dependencies that are computed. By using only backward slices, the code base extracted answers the following question: *"What are the paths on the call graph that can affect the execution of a set of methods M?"*. By slicing forward from call sites instead of backward, using any value returned by the call instead of the call arguments as slicing criteria, the question is rephrased to *"What are the paths on the call graph that can be affected by the execution of a set of methods in M?"*.

Whether to use forward slices, backward slices or both depends on the comprehension task. If, for example, the task is to determine the effect a maintenance fix will have on the rest of the system, the appropriate choice would be to use forward slicing, choosing landmark methods on the paths that execute the code containing the fix. Code inspections on the other hand often require knowledge about how a particular variable could have obtained its value. Using a conventional inter-procedural backward slice would not be concise enough (not restricting the slice to a set of paths or a single path of interest), so we can use our approach with backward slicing to restrict the results.

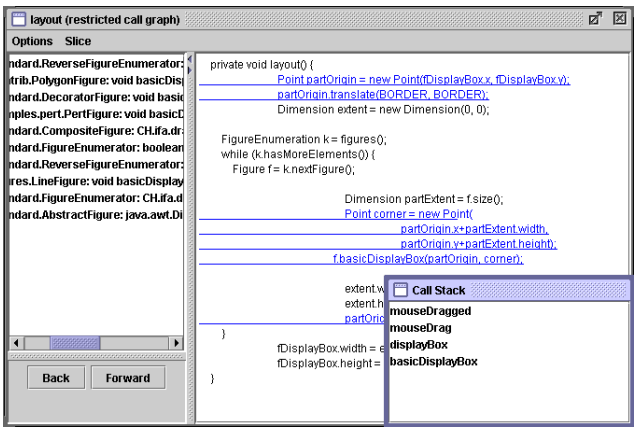


Figure 5. Call graph navigator with window to show call stack

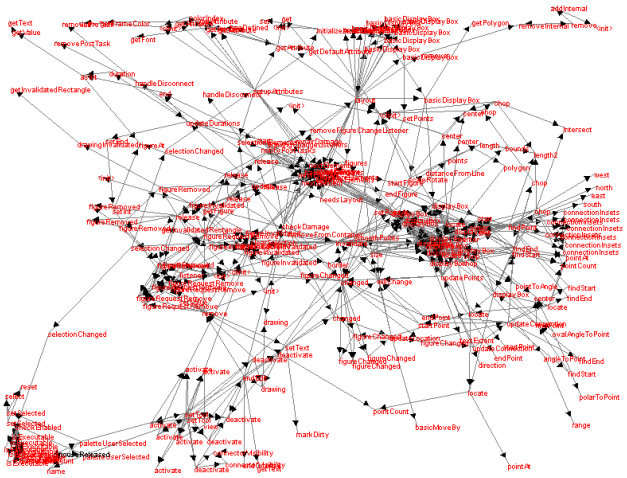


Figure 6. Call graph from PaletteButton.mouseReleased **method in JHotDraw**

4. Evaluation

To assess the feasibility of our approach, it has been implemented as a code extraction tool for Java systems. The tool is currently being evaluated on a range of systems, to understand the effect that different landmark methods have on the results produced. To evaluate its usefulness with respect to comprehension we refer to Storey *et al.*'s set of guidelines [19] that tools for program understanding should aim to satisfy.

4.1. Implementation and Example

Upon loading Java classes the tool produces a call graph. This can be refined by selecting methods that belong to the specification and applying the approach outlined in the previous section. We use the Soot byte code analysis framework[1] as a basis for extracting dependence information and the call graph. 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].

As the call graph is trimmed, it can be navigated as shown in figure 5. Outgoing calls are listed in the pane on the left. When selected, the text-pane reloads with the source code belonging to the selected method. Intra-procedural slices can be highlighted.

As a basis for our example, we have chosen a use-case from JHotDraw[3], a framework for drawing editors. It contains about 120 classes and uses a variety of object-oriented paradigm features such as polymorphism and inheritance. It has evolved into a mature system (its initial Smalltalk implementation was produced in the Eighties) and its design has been developed by using typical framework construction techniques such as design patterns.

The use-case, 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 in figure 6.

The problem of distributed functionality that has to be dealt with 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.

The approach was evaluated in the context of a software inspection task, so the tool was modified to use only backward slices at call sites. The reduced graph that corresponds to the supplied use-case is shown in figure 7 and was obtained by using four landmark methods. The graph is reduced to 16 vertices, connected by 20 edges, returning two edges that are superfluous because of a polymorphic call that is not restricted by landmark methods.

The choice of landmark methods is going to be crucial, so as part of our initial evaluation we are investigating its precision and recall with respect to different method combinations. We have added a function to our tool that allows

[1] http://www.sable.mcgill.ca/soot/
[2] http://jung.sourceforge.net/
[3] http://www.jhotdraw.org

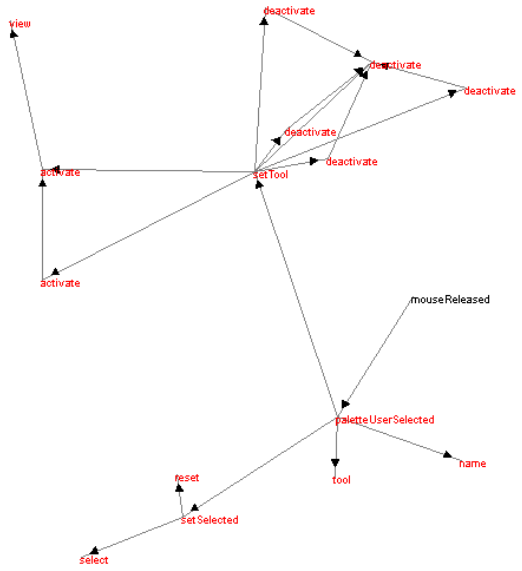


Figure 7. Reduced Call Graph

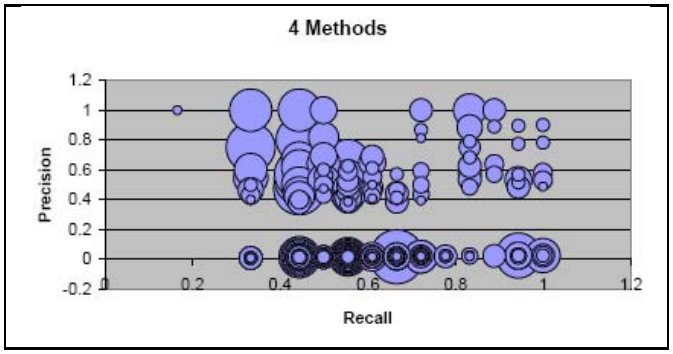


Figure 8. Precision-recall results for the sample use-case, using 4 landmark methods

the user to specify a set of relevant edges R and the number of landmark methods to be considered l. The tool then proceeds to generate every combination of l landmark methods in R. For every combination it produces the reduced call graph and calculates its precision and recall. Precision measures the proportion of retrieved edges that are actually relevant and is calculated by dividing the number of relevant edges retrieved by the total number of edges retrieved. Recall measures the proportion of relevant edges that are actually retrieved and is calculated by dividing the number of relevant edges retrieved by the total number of relevant edges.

It should be emphasised our evaluation of this work is still on-going. We have shown that the approach performs very well, given a good combination of landmark methods (the combination that produced the graph in figure 7 scored 90% precision and 100% recall. Several of the four-method combinations however also produced very poor precision-recall results (the worst case produced 1% precision and 33% recall). A sample of the output produced by the tool is provided in figure 8. The results are displayed in the form of a bubble graph, where the size of the bubble represents the number of landmark method combinations that produced that precision-recall result.

The initial precision-recall results vary significantly from one use-case to the other. By comparing the properties of the landmark methods that produced poor results to those that were useful, we have so far managed to highlight two key features of successful method combinations: (1) wherever possible they should specify the destination of a polymorphic call (otherwise the call graph follows all destinations) and (2) if a scenario contains multiple branches that execute independently of each other (i.e. the call to one branch would not belong to the slice from the call to another branch), each branch must contain a landmark method. Future work will explore further the criteria that make for successful landmark methods.

4.2. Tool Evaluation

Storey *et al.* [19][4] propose a list of fourteen cognitive design elements that should be supported by a software comprehension tool if it is to be effective. Out of these, it was found that our approach can assist on the following nine points:

Improve program comprehension

Enhance bottom-up comprehension

- *Indicate syntactic and semantic relations between software objects:* Edges contained in the reduced call graph produced by our approach link the methods that are relevant to a given execution.
- *Reduce the effect of delocalized plans:* This is the key aim of our approach. Our reduced call graph pulls together elements of code from across the system that may affect or be affected by a particular method.

Enhance top-down comprehension

- *Support goal-directed, hypothesis-driven comprehension:* Using our approach, the hypothesis is expressed in terms of the landmark methods that should be executed if a scenario executes.

[4]Their work carries a strong bias towards software visualisation, but the comprehension aspect is just as important for our work.

Reduce maintainer's cognitive overhead

Facilitate navigation

- *Provide directional navigation:* As shown in figure 5, navigating the call graph before or after it has been reduced is simply a matter of selecting methods from the list and using the 'forward' and 'back' buttons.

Provide orientation clues

- *Indicate the maintainer's current focus:* The current focus is the source code for the method in the pane.
- *Display the path that led to the current focus:* This is represented by the call stack, shown in the lower window in figure 5.
- *Indicate options for further exploration:* The list of methods to the left of the pane indicate methods called by the current method and can be selected for further exploration.

Reduce disorientation

- *Reduce additional effort for user-interface adjustment:* The interface provided by our tool is minimal and very intuitive, only requiring that the maintainer choose the destination method from a list and navigate forwards or backwards.
- *Provide effective presentation styles:* This refers to the need for effective visual feedback from the tool. Our tool presents the call graph and updates it every time an additional landmark method is added (see figures 6 and 7), this is particularly effective because it immediately presents the user with a visualisation of the impact that different landmark methods have on the final call graph.

Our approach is particularly successful with respect to the design elements aimed at reducing the maintainer's cognitive overhead, where it satisfies six out of the seven points listed. The one unsatisfied point states that a tool should also provide arbitrary navigation, which was not implemented in our approach.

With respect to improving program comprehension our approach satisfies three of the seven elements listed. Storey *et al.* list that a tool should also: provide abstraction mechanisms, provide an adequate overview of the system architecture at various levels of abstraction, support the construction of multiple mental models and cross-reference mental models. Their model is however intended for software exploration tools that support program comprehension. Ours is not a software exploration tool, but a code extraction tool, so dealing with different levels of abstraction and different mental models is outside of its scope.

4.3. Code Reading

To explore the nature of the code returned by our approach and the practical issues that arise when trying to read and understand it, we analysed the information produced by the tool for a smaller system [23]. Calls in the reduced call graphs were highlighted in the source code. Classes were printed out individually, and a reverse engineered class diagram was provided to help identify class data members. Methods not belonging to the reduced call graph were removed. Method calls representing edges on the reduced call graph were highlighted, and marked with an identifier of their target method[5]. The following issues were highlighted:

1. Understanding code without further tool support is problematic because it is difficult to maintain the calling context in which a method is being read. It is difficult to remember the caller of a method, so when mentally executing a chain of calls, it is difficult to 'descend' back down the call chain.
2. Although the reduced code set appears to be useful, a clear reading strategy is needed to mark out the important paths through the code.
3. It would be useful to be aware when a method call crosses a class boundary.
4. An approach is needed to handle multiple instances of different objects (possible of the same type) from a static perspective.

Issue 1 can be partially addressed when using a tool. As mentioned previously, the tool displays the call stack that has led to a given method and provides a 'backwards' button to skip back to the previous method (see figure 5). The solution is only partial, because a mental abstraction of the functionality produced by the chain of calls is needed. Issue 2 clearly needs to be addressed and is an important area of future work. Issue 3 is another feature that could be provided by tool support. Issue 4 needs to be taken into account by any static object-oriented comprehension approach. Different objects of the same type may have different states. This is a clear challenge when trying to understand the code from the dynamic perspective and must be taken into account when addressing issue 2.

5. Related Work

Given a complete system, the challenge is to identify the code belonging to a set of use-cases. The key problem

[5] A postscript file can be downloaded from: http://www.cis.strath.ac.uk/~nw/occupantInforScenario.ps

with reverse engineering use-cases (or particular scenarios) is that computing them statically is generally accepted to be an unsolvable problem. Static approaches tend to be necessarily conservative and include a large number of superfluous method invocations. Various static approaches exist that attempt to minimise the amount of irrelevant information (see section 5.2).

Although dynamic approaches are based on an exact trace of the methods that are invoked at run-time, they are unsuitable if we need to understand software without executing it (as is the case with inspections). Another challenge with respect to dynamic approaches is that they rely on the determination of a suitable set of test cases that are exhaustive and representative with respect to the use-cases. They also rely on the availability of a system that is executable in the first place. In object-oriented systems it is particularly common that incomplete systems (such as frameworks) are created, which can then be used in different contexts. Our approach presents a compromise, where dynamic information pertaining to a given set of executions can be used for the analysis in the form of a set of landmark methods.

5.1. Dynamic Approaches

Bojic and Velasevic [2] propose an approach for using run-time information to reverse-engineer use-cases. Their approach is based on relating run-time information obtained from a set of test cases corresponding to a use-case to a concept lattice constructed using formal concept analysis.

El-Ramly *et al.* [7] propose another dynamic approach that records user interaction with the system. Based on these interactions, data mining and pattern matching techniques are applied. Any frequently occurring interactions are used as a basis for use-case models.

Egyed [5] proposes an approach that uses run-time information to produce traces between scenarios, model elements and the system. The user supplies a series of representative test cases and an executable version of the system. The system is executed and a 'footprint graph' is constructed. This is used as a basis for automatically generating further traces.

Eisenbarth *et al.* [6] combine dynamic analysis with formal concept analysis to map program features to procedural source code. They produce an execution profile of scenarios that are of interest and use formal concept analysis to produce a mapping between the features that are invoked by the scenarios and the source code. Their work emphasises the source code units that map to a feature, whereas our work concentrates on the call relationships between units that implement a feature (Eisenbarth *et al.*'s definition of a feature is invoked by a set of scenarios).

5.2. Static Approaches

Di Lucca *et al.* [14] propose an approach that is based on the premise that a scenario starts with a system-level input and ends with a system-level output. They represent the message sequences in the form of a Method-Message Graph (MMG). 'Threads' of message invocations are extracted from the graph and collated to form use-cases.

Tonella and Potrich [22] provide a reverse-engineering approach for interaction diagrams from C++ code. Acknowledging that a purely static approach is over conservative, they use two mechanisms called partial analysis and focussing to ensure that the average size of a graph is small enough to be of use. They validate their approach by applying it to a substantial real-world project.

Qin *et al.* [15] propose an approach based on constructing a call graph-based abstract representation of the subject program called the Branch-Reserving Call Graph (BRCG). This represents calls between methods and retains control dependence information, so that predicate statements that control the execution of a given procedure call are integrated. Because no prior use-case information is used and the approach is static, it returns all possible execution scenarios of the system. This can be alleviated by pruning nodes using a graph-based 'importance metric'.

6. Conclusions and Future Work

We show how to restrict the call graph to contain only methods and calls that may be relevant to the execution of a particular use-case or scenario. A strength of this work is that it allows for the context to be restricted without relying on a fine-grained specification or a dynamic execution trace. This makes it extremely flexible and applicable to the realistic situation of understanding how a system executes given only a high level specification.

The results of our analysis depend on the methods chosen as landmarks, the scenarios in which they are analysed and the system being inspected. These are three variables that need to be investigated in order to determine the relationship between the information garnered from the specification and the resulting code base. This should give us a good idea of how much specification is required to produce a useful code base with respect to a given use-case and how the technique scales to larger systems.

Although this paper has shown how to extract the source code related to aspects of system behaviour, it does not address the problem of how to read through, comprehend and verify it. This area is a potent field for further research, particularly with respect to code inspections, where reading techniques have a major influence on their success. Although there has been a significant amount of research in comprehension and reading techniques for procedural code,

there is still a void in corresponding object-oriented techniques.

With respect to the output generated by our program, a 'degree-of-interest' (DOI) [9] function would guide the code reader to calls that are particularly important by assigning a 'weight' to different edges in the graph. On a call-graph level, it would be useful to now enable the exclusion of methods (and relevant edges) because we know that they *won't* be executed. On a method statement level, we need to investigate techniques such as Harman *et al.*'s Key Statement Analysis algorithm [11], to emphasise statements that contribute to the computation of principal variables belonging to a method.

The reduced call-graph produced by our approach provides benefits beyond software comprehension. It also has the potential to significantly economise further static analysis. Static *inter*-procedural slices of object-oriented programs tend to be very large because the context in which methods are called is not restricted. Our approach would be a suitable pre-processing step to produce a cut-down dependence graph (reducing the significant overhead involved in constructing dependence graphs) and should produce more precise results with respect to the given scenario.

References

[1] D. Binkley and K. Gallagher. *Advances in Computers*, volume 43, chapter Program Slicing. Academic Press, San Diego, CA, 1996.

[2] D. Bojic and D. Velasevic. Reverse engineering of use case realisations in UML. In *Proceedings of the ACM Symposium on Applied Computing (SAC'00)*, pages 741–747, 2000.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[4] A. Dunsmore, M. Roper, and M. Wood. The development and evaluation of three diverse techniques for object-oriented code inspections. *IEEE Transactions on Software Engineering*, 29(8):677–686, August 2003.

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

[6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[7] M. El-Ramly, E. Stroulia, and P. Sorenson. Mining system-user interaction traces for use case models. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pages 21–29, 2002.

[8] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[9] G. Furnas. Generalized fisheye views. In *Proceedings of Human Factors in Computing Systems (CHI'86)*, pages 16–23, 1986.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1999.

[11] M. Harman, N. Gold, R. Hierons, and D. Binkley. Code extraction algorithms which unify slicing and concept assignment. In *9th IEEE Conference on Reverse Engineering (WCRE '02)*, Richmond, Virginia, USA, 2002.

[12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

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

[14] G. A. D. Lucca, A. R. Fasolino, and U. D. Carlini. Recovering use case models from object-oriented code: A thread based approach. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'00)*, pages 108–117, 2000.

[15] T. Qin, L. Zhang, Z. Zhou, D. Hao, and J. Sun. Discovering use cases from source code using the branch-reserving call graph. In *Proceedings of the Tenth Asia-Pacific Software Engineering Conference (APSEC'03)*, pages 60–67, 2003.

[16] T. Reps, H. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 1994.

[17] M. Robillard, W. Coelho, and G. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, December 2004.

[18] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, May 2002.

[19] M.-A. Storey, F. Fracchia, and H. Mueller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems, special issue on Program Comprehension*, 44, 1999.

[20] T. Thelin, P. Runeson, and C. Wohlin. An experimental comparison of usage-based and checklist-based reading. *IEEE Transactions on Software Engineering*, 29(8):687–703, August 2003.

[21] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[22] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *Proceedings of the International Conference on Software Maintainence (ICSM'03)*, pages 159–168, 2003.

[23] N. Walkinshaw. Statically partitioning object oriented code for use-case driven code inspections. Technical Report EFoCS-55-2004, The University of Strathclyde, December 2004.

[24] M. Weiser. *Formal, Psychological, and Practical Investigation of an Automatic Program Abstraction Method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

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