SUPPORTING THE COMPREHENSION OF OBJECT-ORIENTED SOFTWARE SYSTEMS BY EXTENDED M-M GRAPH

Mario Luca Bernardi, Giuseppe Antonio Di Lucca
RCOST Research Centre on Software Technology
Dept. of Engineering
University of Sannio
Palazzo ex Poste, via Traiano, 82100 Benevento, Italy
dilucca/mlbernar@unisannio.it

ABSTRACT

Tracing user functional requirements to the source code components implementing them is a difficult, time consuming and expensive software comprehension task, specially when adequate documentation is absent. In object-oriented systems functional requirements are implemented by a chain of method activations started by an input event and terminating when an output event is generated. Such a chain may be modelled by the Method-Message Graph that has been used to effectively support the identification of sequences of method activations responsible of a given system behaviour and to recover use cases from object oriented code.

This paper presents some extension to the Method-Message Graph enriching it with information about the dependence on the control flow of each method call statement, and data exchanged by method activations. Such information allow to obtain more insight about relationships among method activations, as well as to define with less effort the relationships among the use cases that can be recovered by analyzing the M-M Graph. A case study involving a small sized system implemented in Java language was carried out to validate the approach; it confirmed the validity and usefulness of the proposed M-M Graph extensions and provide us valuable information for future work.

KEY WORDS
Software Engineering, Software Comprehension, Reverse Engineering, Source Code Analysis, UML

1. Introduction

A common task in software comprehension is to trace user functional requirements to the source code implementing them to allow the localization of the components that are involved in testing/maintenance operations. In Object-Oriented systems functional requirements are implemented by sequences of method activations linked by messages exchanged between objects. Indeed, the system external behavior is determined by activation sequences that are started by an input event and terminate with an output event. Localizing such an activation sequence may be a hard task when system documentation is absent, poor, or not suitable for the task the tester/maintainer has to fulfill. In this case reverse engineering techniques provide a fundamental support to accomplish software comprehension effectively.

UML use case diagrams and sequence/collaboration diagrams are used to model the behavior of an O-O system and several commercial tools are available to automatically reconstruct sequence/collaboration diagrams from source code. However, the diagrams produced by these tools usually are not actually useful because they may be too detailed and complex (i.e. difficult to read and comprehend). Moreover they often are too abstract with respect to the granularity of the needed information. This make difficult to detect those method activation sequences that are responsible for a given behavior. Thus alternative analysis and representation techniques to model method activation sequences are needed to effectively support the comprehension process.

Both static or dynamic analysis of the source code can be used to recover method activation sequences and both methods present some advantages and drawbacks. Static analysis allows a complete coverage of all activations statements in the source code but due to the presence of late binding, it could not be possible to know which method is really called at run-time. Dynamic analysis can resolve late binding problems but may be more expensive and time-consuming with respect to static one, because it requires an adequate set of code executions to be defined preliminarily. Moreover, it requires code instrumentation to be accomplished to be sure to have a full coverage of all method activation sequences. In literature some approaches addressing the polymorphism problems in static code analysis are proposed [9].

In [1] an approach to recover use cases from O-O code is proposed; the approach is based on the identification and analysis of method activation sequences recovered by code static analysis. The activation sequences are represented by a graph called the Method-Message Graph (M-M Graph) which is hence analyzed in order to identify notable sub-graphs (i.e. notable sub-sequence of method activations) in it. Subsequently use-cases are associated to those sub-graphs and use-cases relationships are defined according to some rules.

In this paper we propose to improve the static M-M Graph to better address the problem of identifying the type of relationships among the use cases associated to method activation sub-sequences. The proposed improvement mainly consist of adding new types of nodes to the M-M Graph. These nodes represent the control flow structures in which
a method activation is nested. Furthermore the edges of the M-M Graph are annotated with information about the parameters exchanged in a method activation. We call this graph Extended M-M Graph.

The remaining of the paper is structured as follows. Section 2 provides a synthetic background about the M-M Graph and the method to recover use cases and use cases relationships from it. Section 3 presents the new extended representation of the M-M Graph. Section 4 illustrates a case study, with respect to which the usefulness of the Extend M-M Graph is assessed and discussed. Conclusive remarks are, finally, presented in Section 5.

2. Background

2.1 Threads and M-M Graph in an O-O system

In [1] the analysis of method activation sequences is used for recovering a use case model of the analyzed object oriented system. The method is based on the concept of threads as derived from the definitions of M-M path and Atomic System Function (ASF) [4]. A thread is defined as a method activation sequence that starts with an input event and terminates with the production of an output event. Input/output events correspond to read/write statements used to control input/output devices (keyboards, pointing devices, sensors, displays, etc.) as well as to event handlers, i.e. those language constructs used by programming environments and languages (like Java, C++ and C#) to encapsulate the input/output event logic management. A graphical representation of the threads implemented in an O-O system is provided by the Method-Message Graph (M-M Graph). The M-M Graph represents the static activation relationships between the methods in the system and the static inclusion of input/output events in the methods. The input and output events are called respectively imports and exports. In the M-M Graph diamonds are used to represent import and export nodes while circles are used for methods. Directed edges are used to represent following relationships: (i) the inclusion of an import in the corresponding method; (ii) to qualify the destination node method as event handler for certain type of events; (iii) static activation between methods and (iv) inclusion of an export in the corresponding method. Figure 1 shows an example of a simple M-M graph.

The system threads are identified by searching for distinct paths on the M-M Graph connecting each import node with an export node. Table 1 shows the seven threads included in Figure 1. In the figure each node is associated with a unique label: Import/Export nodes are labelled by the "I" ("O") character followed by an integer, while method nodes are labelled by an integer. The M-M graph may include paths that start from an import node and do not reach any export node, such as the path <I1-16-17> in Figure 1. In this case, it is likely that the sub-path <16-17> does not yield any export because this method chain implements a service for the method 1; such a sub-path is collapsed in a single node to be included in the node activating the sub-path itself. Inheritance, polymorphism, dynamic binding,

overriding and overloading are taken into account and statically solved according to the rules defined in [1], when the M-M Graph is produced.

2.2 Use cases and M-M Graph

In [1] some rules are provided to analyze the M-M graph to identify use cases and the relationships among them. In the following, we report those rules synthetically. Each thread from the M-M graph may be assumed to define a distinct use case. Existing threads may not be completely disjoint since they may include common sub-paths (i.e. sub-threads) representing a common behavior shared by more threads. Hence as first step in use-cases recovering, the identification of such common parts is performed. The identification of sub-threads is obtained with a procedure that looks for fork nodes (i.e. a node with more exiting edges) and join nodes (i.e. a node with more entering edges) in a thread, and decomposes each thread in more parts, each of which may be:

- a sub-thread delimited by an import node and a fork/join node
- a sub-thread delimited by a fork/join node and an export node
- a sub-thread delimited by a fork/join node and another fork/join node.

As an example, with respect to the threads listed in Table 1, the following sub-threads can be identified: <i1, 1, 2, 3>, <4, 5, 6>, <9, 15>, <o1>, <o2>, <7, 8, o3>, <i2, 10, 11, 12>, <13, 14, o4>.

Each sub-thread is associated to a single use-case. To make more readable the M-M Graph, when a sub-thread forks in distinct sub-threads each including only an export node,
the forking sub-thread and all the outports will be associated with a unique use-case (with a different behavior scenario associated to each outport). In this way, with reference to Figure 1, we can identify the following set of use cases: U1: <1, 2, 3>, U2: <4, 5, 6>, U3: <9, 15, O1, O2>, U4: <7, 8, O3>, U5: <12, 10, 11, 12>, U6: <13, 14, O4>. In [1] relationships among the use cases are defined by applying the following heuristics:

- When two or more threads join in a node, the sub-thread starting with join node is shared by the joining threads. In this case, we can hypothesize that such a sub-thread provides a reusable sub-behavior, which can be modeled as a different use case included in the use cases associated with the joining sub-threads.

- When a thread forks in distinct sub-threads, the sub-thread terminating in the fork node (we call it as 'forked sub-thread') is shared by each of the forking sub-threads. In this case, a use case will be associated with the common initial part of the threads (i.e. the forked sub-thread), and for each forking sub-thread there will be a distinct use case. As far as the relationships between them is concerned, further deeper source code analysis is needed to define the correct «include» or «extend» relationship.

- Threads including one of the methods inherited from a virtual method may be associated with use cases that specialize a more general use case. Indeed, we abstract from these threads a general use case and consider each thread as a specialization of the general one.

From a first use case diagram directly recovered by the M-M graph, a hierarchy of use case diagrams at a higher level of abstractions can be built by applying the rules defined in [1]. In this paper, as far as the use case relationships is concerned, we focus only on the «include» and «extend» relationships.

2.3 Limits and drawbacks

The main limit of the M-M Graph proposed in [1] is that it does not contain any useful information to automatically support the definition of the «include» and «extend» relationships among the identified use cases, but further manual code inspection is required to define the correct type of relationship.

In particular, a source code analysis of the methods associated to join/fork nodes, for assessing if the activation of the joined/forking methods is alternatively or sequentially executed, is necessary to decide between «extend» and «include» alternatives, respectively.

As an example, let us consider the M-M Graph in figure 1 and the use cases U2, U3, and U4 we identified in it. The use cases U3 and U4 correspond to two sub-threads that fork from U2 (i.e. from the fork node 6 included in U2). To correctly define the relationships between U2 and U3 (respectively U2 and U4) we have to inspect the code of the method associated to node 6 in order to verify if the activation of the method associated to node 9 (respectively node 7) is depending on a predicate (i.e. the activation is nested in a selection or cyclic control structure) or not (i.e. it is nested in a sequence). In the first case we will define an «extend» relationship, in the latter an «include» one.

Thus, given a fork node in the M-M Graph, we can define the following rules:

- If the method activation is nested in a sequence, an «include» relationship is defined between the forked use case and the forking ones. In particular, the forked use case «include» the forking ones. The reason is because the forking use cases will be unconditionally always executed when the the forked one is executed.

- If the method activation is nested in a selective or cycle control structure, an «extend» relationships is defined between the use cases. In particular, the forking use cases «extend» the forked one. The reason is because the forking use cases will be conditionally executed, thus representing an optional behavior that may be inserted into a base use case (i.e. the forked one) [2].

Similar rules, that we do not report here for sake of brevity, are defined in the case of joining use cases.

Another limit of the M-M Graph is that it does not contain any information about the data exchanged in the activation statements. Such an information would provide a first useful support to identify, at a coarse grained level, the data dependency between the inport and the outport defining a thread. Also in this case further code inspections is needed to recover data dependency information.

The main drawback due to these limitations consist of a more effort to spend in carrying out an effective code inspection to recover information both about the nesting of method activations in the methods associated to the final nodes of a thread and the data exchanged between methods. To reduce this effort the M-M Graph can be improved by extending it by adding new elements representing the needed information. We call Extended M-M Graph this new graph.

3. The Extended M-M Graph

The M-M Graph does not include any information about the nesting of method activation statements in the control flow structures. As noted in section 2.3 this information is needed to define the correct relationships between use cases corresponding to the sub-threads in the M-M Graph. Thus the availability in the graph of this information would avoid the manual code inspection needed to define correct use case relationships. The E-M-M Graph overcomes this problem by additional nodes providing us a synthetic representation of the nesting of the control flow structures in which the method activations are placed. Two new types of node are defined:

- **Sequence node (\( / \)**) - corresponding to a sequence control structure in which one or more call statements can be nested and executed unconditionally.

- **Alternative node (\( + \)**) - corresponding to a predicable control structure (such as if-then, if-then-else, while-do,for statements) that will condition the execution of the call statement nested in it.
In the E-M-M Graph, Sequence and Alternative nodes are represented by a box labelled respectively by "/" and "+". These nodes will be added along the edges linking two method nodes in the graph. However to make the Graph more readable the new nodes are just reported along the edges linking the final node of a sub-thread with the first node of another successive sub-thread (i.e. just for the call statements in the final method of a sub-thread to the first method of a successive sub-thread). This because our attention is focused on the definition of the relationships among the use cases associated to each sub-thread. The series of Sequence and Alternative nodes between two method nodes will represent the nesting of the control structures where the call to a method is placed. As an example, let us consider the following excerpt of pseudo C++ code representing the method Method_C() of a class C in which there are some call to a method is placed. As an example, let us consider the following excerpt of pseudo C++ code representing the method Method_C() of a class C in which there are some call to a method is placed. As an example, let us consider the following excerpt of pseudo C++ code representing the method Method_C() of a class C in which there are some call to a method is placed. As an example, let us consider the following excerpt of pseudo C++ code representing the method Method_C() of a class C in which there are some call to a method is placed.

```cpp
public:
  void Method_C() { // id0
    A cA = new A(...);
    is(); // Inport I1
    if (pred_1) {
      while (pred_0) {
        if (pred_2)
          cA.Method5(); // id1
        else if (pred_3)
          cA.Method6(); // id2
        else cA.Method4(); // id3
          cA.Method2(); // id4
      }
    cA.Method1(); // id5
  }
  else {
    cA.Method20(); // id6
    if (pred_4)
      cA.Method21(); // id7
    else
      cA.Method22(); // id8
    cA.Method60(); // id9
    cA.Method61(); // id10
  }
  //end of Method_C()
```

In figure 2 the E-M-M graph corresponding to the above excerpt of pseudo C++ code is showed. In this graph the import and method nodes are labelled with the corresponding identifiers indicated as a comment in the pseudo code. Moreover, the "O" outport indicates a generic set of outports where the hypothetical sub-threads starting from the called methods, represented by the dashed lines, will terminate. In the graph we can identify a fork node, corresponding just to the Method_C() in the pseudo code, from which ten sub-threads start. According to the rules in section 2, we can associate a use case to each sub-thread, resulting in eleven use cases: the first one (UC0) composed by the node I1 and the node id0, while the other use cases (UC1, ..., UC10) will be formed by each remaining sub-thread rooted in the called nodes. Analyzing the E-M-M Graph we are able to say in which type of control structure (Sequence or Alternative) a method activation, linking two sub-threads, is nested and then if its execution depends on any condition. Indeed, just 'walking' along the path linking two sub-threads, if only Sequence nodes are crossed the method activation does not depend on a condition, while if at least an Alternative node is crossed the method activation is nested in at least a selective or cycle structure and then its execution depends on a condition. In the E-M-M Graph in figure 2 we have that just the execution of the methods corresponding to the nodes id9 and id10 is unconditioned (only "/" node are crossed from node id0 to nodes id9 and id10) while the executions of all the other methods are conditioned (at least a "+" node is crossed going from node id0 to all the remaining other nodes). These considerations allow us to define the following rules to define the type of relationship between the use cases:

- if, in the E-M-M Graph, along the path linking two sub-threads only Sequence nodes are crossed then an «include» relationship is defined between the two use cases associated to the two considered sub-threads;
- if, in the E-M-M Graph, along the path linking two sub-threads at least an Alternative node is crossed then an «extend» relationship is defined between the two use cases associated to the two considered sub-threads.

In this way just analyzing each path linking two sub-threads we can define the relationships between the use cases associated to those sub-threads and obtain a use case model consistent with the source code. In the case of the E-M-M Graph in figure 2, then we have that the use case UC0 «include» the use cases UC1 and UC2, while all the other use cases UC3, ..., UC10 «extend» UC0.

Another extension included in the E-M-M Graph regards the edge annotation: each edge in the graph is annotated with the data involved in the operation the edge represents. Then, an edge linking an Import (respectively an Outport) to a method node will be annotated with the names of the identifiers of the data read (written) in an input (output) operation. Similarly, an edge linking two method nodes will be annotated with the parameters exchanged by the call statement. This information will provide a first useful support, even if at a very coarse grain level, to begin the identification of the dependencies among the data flowing through a thread, from the import to the output. Of course, due to the static analysis of the source code the results are affected by all the limits of this kind of analysis; anyway also this simple support can reduce the effort to comprehend this aspect of the system by allowing an easier localization of the components to further analyze. Finally, also the Alternative nodes are annotated with the predicate characterizing the corresponding control structures they represent.

4. Case study

A case study has been carried out with the aim of assessing the better correctness and effectiveness of the E-M-M-Graph in recovering use case relationships, with respect to the M-M Graph. The object of the study was the ‘Mic1Sim’ software system, provided by Tanenbaum in his computer architecture textbook[3]. The ‘Mic1Sim’ system is a simulator of the example architecture described in Tanenbaum’s
textbook, coded by java programming language and made up of 50 methods included in 24 classes for a total size of about 3 KLOC. To recover the E-M-M Graph of 'Mic1Sim' a prototype tool-chain was developed. The static java code analysis was performed by using the BCEL library[8]. The data of interest for drawing the E-M-M Graph were abstracted and saved in a text file, according to a metadata model that has been defined to represent the E-M-M Graph. A JavaCC [10] parser for this representation is then used to instantiate the E-M-M Graph. It is worthwhile to note that it is possible both to save the data for representing an entire system or just a portion of it. As far as concern the problem due to polymorphism in static code analysis, this limitation was partially overcome by using a static class hierarchy analysis algorithm to determine the set of all the possible calls at run-time. As a consequence, the static-recovered E-M-M Graph contains a superset of the actual calls in the system. Another issue derives from the inputs and outputs identification. The metadata model defined for the E-M-M Graph representation allows to specify the event handlers and the basic I/O library functions to have to consider. On the other hand, the java analyzer has to be properly configured for the libraries accessed by the system to take them into account. Thus the reverse engineer is able to customize the tool as needed in order to identify and recover all the known I/O functions and event handlers in the analyzed system.

The E-M-M Graph recovered from the 'Mic1Sim' system was composed of 196 threads, with 46 subthreads, 8 inputs and 60 outputs. By applying the rules defined in §3, a use-case diagram was built and a concept, describing the behavior of the use case, was assigned to each use case. As an example, an excerpt of the Mic1Sim E-M-M Graph is depicted in figure 3, while the use-case model related to this portion of E-M-M Graph is showed in figure 4; this use cases correspond to the initial startup and configuration of the machine architecture. Starting from the first use case diagram directly recovered from the E-M-M Graph, a hierarchy of use case diagrams was derived by applying the rules defined in [1]. In this way a view at a higher abstraction level of the system was obtained, allowing to relate each high level use-case to more detailed use-cases that realize it and to the methods implementing those use cases.

To assess the better correctness of the use case diagram produced by using the E-M-M-Graph with respect to the M-M Graph, a software engineer was asked to produce a use case diagram for the Mic1Sim system by using the M-M Graph and the rules described in section 2. The latter diagram, in which the relationships among the use cases were recovered by human code inspection, was validated with respect to the system functional specification, and this diagram was assumed as to be the correct one. Then we compared the two use case diagrams: of course the two diagrams showed the same number of use cases, each one related to the same set of chain of method activations, while some dif-
ferences existed among the relationships defined in the two diagrams. In the diagram recovered by the M-M Graph, human code inspection recognised some 'generalisation' relationships among certain use cases. These relationships were not recognised by using the E-M-M Graph just because they were not considered in this approach, that just recovered 'include' or 'extend' relationships among the same use cases (however, usually the generalisation relationship is based on semantic aspects that are difficult to capture automatically). In all the other cases the differences were about relationships defined as 'include' ones by using the M-M Graph, while in the E-M-M Graph they corresponded to 'extend' relationships. Talking with the software engineer involved in the recovering the use case diagram by the M-M Graph he said that in some case he preferred to define an 'include' relationship instead of an 'extend' one because the use cases were 'conceptually' included in the base use case, even if the call statements were depending on a predicate. This was actually true in some cases, for example when activations of the same method were both in the 'true' and 'false' branches of an 'if-then-else' statement, but with different values of the exchanged parameters. As a consequence the recovered use case model by the E-M-M Graph has to be submitted to a 'semantic' analysis to validate the correctness of the recovered 'extend' relationships.

The study proved that the most of the relationships recovered by the E-M-M Graph were the same of those ones recovered by the M-M Graph, but the usage of M-M Graph required a heavy manual code inspection of the code to identify the type of control flow structure a method activation was nested in. Then, the E-M-M Graph allowed to reduce significatively the effort needed to recover a use case diagram with correct relationships, where only some 'extend' relationships has to be validated.

All that showed us the actual effectiveness of the E-M-M Graph in supporting the software comprehension task. In particular the new rules for automatically defining the relationships among the use cases allowed a greater correctness when building use case diagrams at higher abstraction level and, thus, a greater precision in tracing them to the source code with a sensibly reduction of the needed effort.

5. Conclusions

An extended version of the M-M Graph has been proposed in this paper. The extensions are mainly concerned with the introduction of new types of nodes, Sequence and Alternative nodes, taking in account the nesting of the method call statements in the control flow structures. This make easier and cheaper to define, according to some rules defined in the paper, the relationships between the use case associated to the sub-thread identified in the graph. Other extensions are concerned with the annotation of the M-M-Graph edges, with the data exchanged between the nodes linked by an edge, and of the Alternative nodes, with the condition in the predicate defining the control structure such a node represents. These annotations are a valuable initial support in tracing the data dependencies along a thread and in testing operations aiming to cover all the threads in the graph.

Moreover, future work will be devoted to exploit these annotations to recover sequence diagrams depicting different scenarios of the use cases recovered from the E-M-M Graph itself. Future work will be devoted also to dynamically analyze a system and to integrate the results from static and dynamic analysis to overcome some problems of the static one. A case study was carried out to assess the validity and correctness of the proposed M-M Graph extensions in supporting the definitions of the relationships among the recovered use cases: the results were more than encouraging.

References


[10] SUN, Java compiler-compiler (JavaCC) (http://javacc.dev.java.net)