Static and Dynamic C++ Code Analysis for the Recovery of the Object Diagram

Paolo Tonella and Alessandra Potrich
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{tonella, potrich}@itc.it

Abstract

When a software system enters the maintenance phase, the availability of accurate and consistent information about its organization can help alleviate the difficulties of program understanding. Reverse engineering methods aim at extracting such kind of information directly from the code. While several tools support the recovery of the class diagram from object oriented code, no work has insofar attacked the problem of statically characterizing the behavior of an object oriented system by means of diagrams which represent the class instances (objects) and their mutual relationships.

In this paper a novel static analysis algorithm is proposed for the extraction of the object diagram from the code, based on a program representation called the object flow graph. Partial object diagrams can be associated dynamically to the system by executing and tracing the program on a set of test cases. The complementary nature of these two views will be discussed, and a novel approach to object oriented testing will be derived from such a comparison. The usefulness of the proposed technique is illustrated on a real world, public domain C++ system.

1 Introduction

During maintenance, the most reliable and accurate description of the behavior of a software system is its source code. Such a valuable information repository cannot directly answer all questions about the system. Reverse engineering techniques provide a way to extract higher level views of the system, which summarize some relevant aspects of the computation performed by the program statements. Reverse engineered diagrams can support program understanding activities, as well as restructuring interventions and traceability.

The class diagram [7] can be easily recovered from an existing object oriented system, by means of a syntactic analysis of the classes declared in the program and of the type of the class fields. When a class field is another class of the system, an association can be drawn between the two. Although such a view is the basic one for program understanding, it is not much informative of the behavior that the program will exhibit at run time, being focused on the static relationships between classes. On the contrary, the *object diagram* represents the instances of the classes that are created dynamically, and the related inter-object relationships. This program representation provides additional information with respect to the class diagram on the way classes are actually used. Further diagrams that can be derived from the object diagram, such as the collaboration and the sequence diagrams [7], characterize the identified objects in terms of the messages that are exchanged.

In this paper two techniques for the automatic extraction of the object diagram are investigated. The first technique is based on a static analysis of the source program. It exploits flow analysis to propagate information about the allocated objects up to the object fields, so that the inter-object relationships mediated by the object fields are approximated statically in a conservative way. Consequently, this analysis reports a safe superset of the relationships that are expected to hold during any program execution. The second technique considered in this paper is based on the execution of the program on a set of test cases. Each test case is associated to the object diagram depicting the objects and the relationships that are instantiated when the test case is run. The diagram can be obtained as a postprocessing of the program traces generated during each execution (following the suggestions in [7] for the construction of the *UML* object diagram). These two program views are complementary, in that the first is safe with respect to the objects and the relationships represented, but cannot provide precise information on their actual multiplicity, nor on the actual memory shape associated with the objects that are allocated. The dynamic view is clearer about number of instances and memory layout, but is (by definition) partial. Therefore, it is useful to contrast it with the static one to determine the por-

Proceedings of the International Conference on Software Maintenance (ICSM'02)
0-7695-1819-2/02 $17.00 © 2002 IEEE

tion of the overall view that was explored with the available test suite. Such a comparison can be conducted for program understanding purposes, as well as for testing purposes. As a by product of our analysis, we propose two novel testing criteria – object and inter-object relationship coverage – that are well suited for object oriented systems.

The difficulties in reverse engineering and maintenance of object oriented applications are discussed in [4] and [10]. Available commercial design tools (such as Rational Rose and Together) allow the static extraction of the class diagram from the code, while information about class instances is collected at run-time only by research prototypes [3, 5, 6, 9]. In these works, creation of objects and inter-object message exchange are captured by tracing the execution of the program on a given set of scenarios. On the contrary, to the best of the authors' knowledge, no work considered the problems related to reverse engineering of the object diagram statically, from the code.

The core analysis algorithm we propose for the static inference of the referenced objects is based on the type inference techniques described in [2]. While in the original formulation of the analysis, sets of *types* are attached to variables, and are updated according to the statements encountered in a flow insensitive visit of the program, in our approach, variables are associated to the *objects* they reference, and such information is propagated inside the object flow graph, a novel program representation designed specifically for the static analysis we propose. In a previous paper [8], a similar flow analysis algorithm was proposed for the recovery of the class diagram in presence of weakly typed containers, which make the field types unusable for reverse engineering, when fields are containers.

The paper is organized as follows: the next section presents static and dynamic techniques for the extraction of the object diagram, discusses their relationship, and derives two object oriented testing criteria from them. Section 3 describes the experimental results obtained on the C++ application *Gutebrowser*. The last section is devoted to conclusions and future work.

2 Recovery of the Object Diagram

The object diagram represents the set of objects created by a given program and the relationships holding among them. The elements in this diagram (objects and relationships) are *instances* of the elements (classes and associations, resp.) in the class diagram. The difference between object diagram and class diagram is that the former instantiates the latter. As a consequence, the objects in the object diagram represent specific cases of the related classes. Their fields are expected to have well defined values and the relationships with other objects have a known multiplicity. For each class in the class diagram there may be several objects instantiating it in the object diagram. For each relationship between classes in the class diagram there may be object pairs instantiating it and pairs not related by it.

The usefulness of the object diagram as an abstract program representation lies in the information specific to the instantiation of the classes that it shows. While the class diagram summarizes all properties that objects of a given class may have, the object diagram provides more details on the properties that specific instances of each class possess. Different instances may play different roles and may be involved in different relationships with other objects. While this is not apparent in the class diagram, the object diagram represents such kind of information explicitly. For example, there may be one `Node` class in the class diagram for a *BinaryTree* program, with two auto-associations named `left` and `right` for the two children, while a possible instance represented in an object diagram may include three objects of type `Node`, playing three different roles (e.g., tree root, left child and right child). The relationships between these three elements are compliant with those in the class diagram, but provide more information on the layout of the related instances by showing a specific scenario (where the root references two children which have no further descendants). Moreover, the object diagram is the starting point for the construction of the interaction (collaboration and sequence) diagrams, where information about the message exchange between objects is added to the class instances, thus focusing the view on the dynamic behavior of a set of cooperating objects (a *collaboration*, in the UML terminology).

In the following, two techniques are described for the recovery of the object diagram. The first exploits only static information and approximates the set of objects created in the program by analyzing the allocation (`new`) statements and propagating the resulting objects by means of a flow analysis algorithm. The second considers a set of execution traces, associated with the test cases available for a given program, and computes the object diagram by analyzing the actual memory areas allocated during execution, and the addresses of the pointer fields. These two techniques have advantages and disadvantages and it is therefore desirable to be able to compute and integrate both of them.

2.1 Static analysis

The static computation of the object diagram we propose exploits flow propagation to transmit information about the objects that are created up to the fields that reference them. The data structure on which flow propagation takes place is called the Object Flow Graph (OFG) and contains as nodes the program locations which may hold a reference to an object, while its edges connect two locations when there is a program statement through which an object referenced by the first location can be assigned to the second.

The OFG cannot be derived directly from the code. Some of its nodes can be created only when (partial) results of the flow propagation are available. In fact, program locations associated to fields of classes are mapped to OFG nodes with the containing objects as prefixes. When a class field is assigned a value, it is not known a priori the object which the field belongs to, and, at the same time, the computation of the objects referenced inside methods is done while the analysis progresses. Consequently, an incremental construction of the OFG has to be conducted, intermixed with the propagation of flow information about the allocated objects.

```
objectAnalysis()
1   N ← 0, change ← true, graph ← ∅
2   while change
3       graph ← incrBuildObjectFlowGraph(graph)
4       change ← fixpoint(graph)
5   end while

incrBuildObjectFlowGraph(graph: Graph)
1   for eac hs: Statement in PROGRAM
2       addEdges(graph, s)
3       addGEN(graph, s)
4   end for
```

Figure 1. *Main loops of object analyzer and object flow graph builder.*

Fig. 1 shows the main loop of the algorithm. It contains two main steps (lines 3 and 4). In the first, the OFG is incrementally built (i.e., at each step some edge may be added, if not present), and in the second step the flow information is propagated until the fixpoint is reached. Such steps are repeated as long as some information changes in the OFG (either due to incremental graph construction or to flow propagation). The procedure for the incremental construction of the OFG (*incrBuildObjectFlowGraph*) examines in turn each program statement and, if necessary, adds new edges or new flow information (*GEN* sets of the OFG nodes) according to the type of statement under consideration.

The flow information inserted into the *GEN* sets of the OFG consists of class names (e.g., `A`) suffixed with a unique object identifier (an integer value, in the following, as in $\mathtt{A}_N$). The equations used for flow propagation in the *fixpoint* procedure are the usual flow analysis equations [1] (with the union as meet operator and empty *KILL* set):

$$IN[n] = \bigcup_{p \in pred(n)} OUT[p] \quad (1)$$

$$OUT[n] = GEN[n] \cup IN[n] \quad (2)$$

where n is an OFG node and *pred(n)* is the set of its predecessors in the OFG. $GEN[n]$ contains the set of objects directly assigned to node n in an allocation statement (such as, `n = new A()`), while $IN[n]$ and $OUT[n]$ contain incoming and outgoing information computed dynamically by the fixpoint routine according to the equations above.

```
addEdges(graph: Graph, stmt: 'lhs = rhs')
1   out1 ← expand(rhs)
2   out2 ← expand(lhs)
3   for each x in out1
4       for each y in out2
5           addEdge(graph, (x, y))
6       end for
7   end for

addGEN(graph: Graph, stmt: 'lhs = new A()')
1   out ← expand(lhs)
2   N ← N + 1
3   for each x in out
4       addGENToNode(x, {A_N})
5   end for
```

Figure 2. *Procedures to incrementally build the object flow graph and fill-in the GEN sets of its nodes.*

The incremental construction of the OFG is described by the pseudocode of the procedure *addEdges* in Fig. 2, which considers the case where the statement to be examined is an assignment with right hand side *rhs* and left hand side *lhs*. Since object references can be propagated from *rhs* to *lhs* due to this statement, an edge has to be added in the OFG between the location associated with *rhs* and the location associated with *lhs*. In case *rhs* or *lhs* are class fields, the related locations can be obtained by expanding their prefix into the set of objects having them as fields (procedure *expand*), as described in Fig. 3.

Statements different from assignments can be either reduced to the case of the assignment in Fig. 2, or require no edge addition. For example, a method call such as `q = p->f(ac)`, where `p` is of type `A*` and class `A` contains a method `f`, with formal parameter `fm`, can be processed as equivalent to the following assignments:

```
D::g::q    = A::f::return
A::f::fm   = D::g::ac
A::f::this = D::g::p
```

That is, a `return` location, with `A::f` as scope, is introduced to represent the returned value. Actual parameters are assigned to formal parameters, and the object on which the method invocation is issued is assigned to the `this` pointer of the invoked method. Edges in the OFG are created for each of these three assignments. Variables `q`, `ac` and `p` have been assumed to be local to a method `g` of a class `D`. When they are class fields, a preliminary expansion operation is required, similarly to *lhs* and *rhs* in Fig. 2.

Incremental generation of flow information is described by procedure *addGEN* in Fig. 2 for an assignment statement

whose right hand side is an allocation expression. An object identifier A_N is created and added to the *GEN* set of the (possibly expanded) left hand side. A_N represents the object of type A instantiated at the line of code under examination (identified by N). This implies that each line of code containing an allocation expression is associated with an object possibly created by the program during some of its executions.

Statements different from assignments which involve an allocation expression can be easily reduced to the assignment. For example, the allocated object may be passed to a method as an actual parameter, as in: `p->f(new B)`. If the formal parameter of method `f`, assumed to belong to class `A`, is `fm`, this method invocation can be handled as equivalent to the assignment:

```
A::f::fm = new B()
```

```
expand(x: Var):Set<Location>
1   A ← class(scope(x))
2   f ← method(scope(x))
3   if x is a field of A
4       S ← ∅
5       for each y in OUT[A::f::this]
6           S ← S ∪ {y.x}
7       end for
8   else
9       S ← {x}
10  end if
11  return S
```

Figure 3. *Class fields are expanded into locations prefixed by the names of the objects they belong to.*

Object identifiers are propagated along the OFG by the *fixpoint* procedure, according to the flow equations given before. The result is that the *OUT* set of each OFG node contains the set of objects possibly referenced by that node. Therefore, the *expand* procedure, aimed at prefixing a class field with all objects that may own it as a field, exploits the *OUT* set as the currently available set of objects referenced by the `this` pointer (`A::f::this` in Fig. 3).

If, for example, the statement `p = q` is encountered in the program inside method `f` of class `A`, and `p` is a class field, while `q` is a local variable, the *expand* procedure will take all objects in *OUT*[`A::f::this`], say B_1 and B_2, and will generate the two locations B_1`.p` and B_2`.p` as expansions of `p`. Then, edges from `A::f::q` to B_1`.p` and B_2`.p` will be created.

Construction of the object diagram is a straightforward post-processing of the computation described above. Every node in the OFG associated to an object – i.e., having a prefix A_N, where `A` is a class and N an integer, and a suffix `.p`, where `p` is a field of class `A` – generates a node in the object diagram, identified by the location prefix (A_N for A_N`.p`). The *OUT* set of the original OFG node (i.e., *OUT*[A_N`.p`]) gives the set of objects reachable from the current one along the association implemented through the field `p`. Such an association can thus be given the name of the field, `p`.

Fig. 4 (left) shows the code of a program implementing a binary search tree. It consists of a class `Node`, whose private fields `left` and `right` reference left child node and right child node respectively (if any). Its public methods allow inserting and retrieving child nodes. The second class of this program, `BinaryTree`, declares a pointer to a tree node, giving access to the tree root. Among the other public methods of this class, method `build` has the responsibility of constructing the tree. It first allocates the root (line 13) and set it as the current node (variable `curNode`, line 14). Then, after some computation (ellipsis at line 15), according to some condition `c` (line 16) it appends a new node as left or right child of the current node and it updates `curNode`. The `main` function of the program creates a `BinaryTree` object and builds a tree (lines 26, 27).

When the object analysis algorithm of Fig. 1 is applied to the code in Fig. 4, the main loop which invokes the incremental OFG construction and the fixpoint procedures is traversed 4 times. The OFG produced during such 4 iterations is depicted in Fig. 4 (right). During the first iteration, an edge between `main::bt` and `BinaryTree::build::this` is added to the initially empty OFG, due to the method invocation at line 27. Moreover, an object, identified as `BinaryTree-26` (the line number was used as integer suffix for clarity), is added to the *GEN* set of `main::bt`, because of the allocation performed at line 26. The method invocations at lines 17, 18, 20, 21 are the reasons for the edges from `BinaryTree::build::curNode` to the `this` pointers of the called methods, while the assignments involving the returned values at lines 18, 21 justify the edges from the two related `return` locations to `BinaryTree::build::curNode`. Two *GEN* sets containing respectively `Node-17` and `Node-20` are associated with the formal parameter `n` of `addLeft` and `addRight`, due to the allocations at lines 17 and 20.

During the first iteration, the expansion of `left` and `right` to be performed when processing lines 4, 5, 6, 7 gives an empty set, since the *OUT* of the `this` pointer associated with the respective methods is empty (the index of *GEN* and *OUT* sets gives the iteration when they are filled-in). After flow propagation there is a change of the *OUT* set of node `BinaryTree::build::this`, which then contains `BinaryTree-26`. In the second iteration, the field `root` of class `BinaryTree` is expanded into `BinaryTree-26.root` when lines 13 and 14 are analyzed. Consequently a new node

```
1   class Node {
2      Node *left, *right;
3   public:
4      void addLeft(Node *n) {left = n;}
5      void addRight(Node *n) {right = n;}
6      Node* getLeft() {return left;}
7      Node* getRight() {return right;}
8   };

9   class BinaryTree {
10     Node *root;
11  public:
12     void build() {
13        root = new Node();
14        Node* curNode = root;
15        ...
16        if (c) {
17           curNode->addLeft(new Node());
18           curNode = curNode->getLeft();
19        } else {
20           curNode->addRight(new Node());
21           curNode = curNode->getRight();
22        }
23     }
24  };

25  void main() {
26     BinaryTree *bt = new BinaryTree();
27     bt->build();
28  }
```

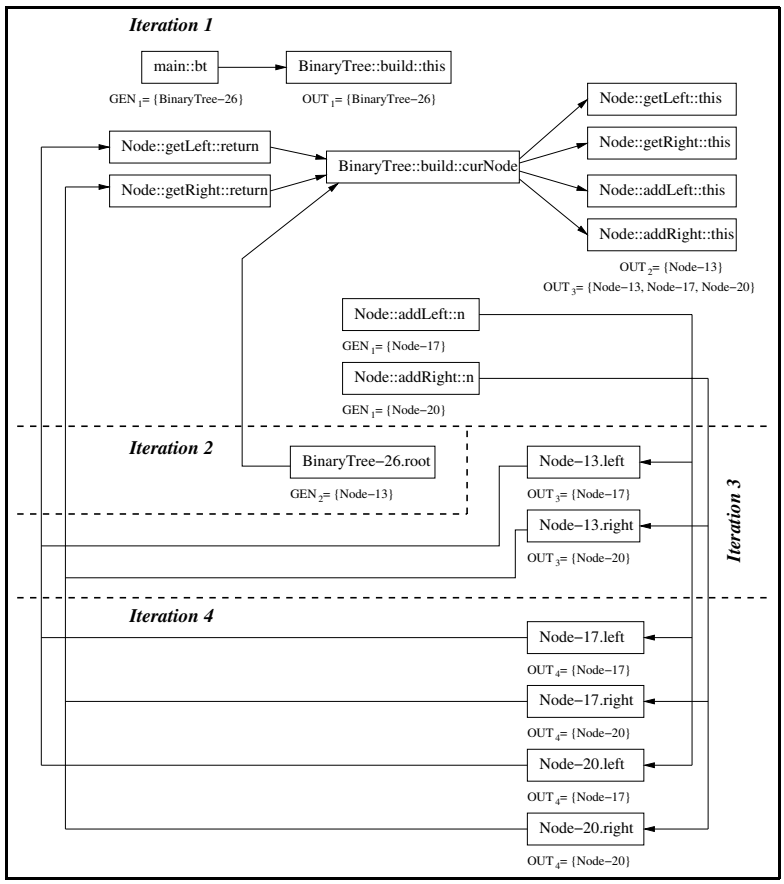

Figure 4. *C++ code implementing a binary tree (left) and flow analysis iterations on the related OFG (right).*

`BinaryTree-26.root` is added, linked to `BinaryTree::build::curNode` and with a *GEN* set containing `Node-13`, the object allocated at line 13. Such an object reaches the `this` pointer of `getLeft, getRight, addLeft, addRight` thanks to the *fixpoint* procedure which propagates the flow information.

At the next iteration, the expansion of the fields `left` and `right` at lines 4, 5, 6, 7 results in `Node-13.left` and `Node-13.right`, giving rise to two new nodes connected to the `return` of `getLeft` and `getRight` respectively and reachable from the formal parameter `n` of `addLeft` and `addRight`. These new connections allow the objects `Node-17` and `Node-20` to be propagated during the fixpoint to the *OUT* sets of respectively `Node-13.left` and `Node-13.right` and to the *OUT* sets of `Node::getLeft::this`, ..., `Node::addRight::this`.

In the last iteration, fields `left` and `right` are expanded again during the analysis of lines 4, 5, 6, 7, originating four new nodes, `Node-17.left`, `Node-17.right`, `Node-20.left`, `Node-20.right`, with similar connections as `Node-13.left` and `Node-13.right` and, after fixpoint computation, with similar *OUT* sets. Then, any attempt to add new edges or generate new flow information produces no change in the current situation, so that the algorithm stops and the analysis is complete.

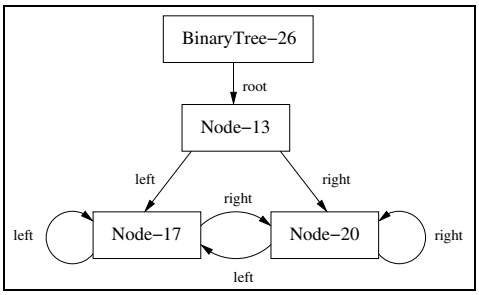

Figure 5. *Static object diagram for the binary tree program in Fig. 4 (left).*

The postprocessing necessary to produce the object diagram from the OFG obtained after the completion of all iterations involves determining the objects and the inter-object relationships. The objects identified during OFG

construction are in the *OUT* sets of the OFG nodes. In our example, 4 objects are present: `BinaryTree-26`, `Node-13`, `Node-17`, `Node-20`. They correspond to the 4 lines of code where classes are instantiated (allocations at lines 13, 17, 20, 26). Inter-object relationships can be recovered by considering the *OUT* set of the OFG nodes which have an object name as prefix and a class field as suffix (e.g., `Node-17.right`). The field name can be used to label the relationship, while the objects in the *OUT* set (e.g., `Node-20` for `Node-17.right`) are the target of an association starting at the object prefixing the OFG node. The object diagram obtained after the analysis of the example in Fig. 4 is shown in Fig. 5. The `BinaryTree` object allocated at line 26 references a tree `Node`, allocated at line 13, through its field `root`. The root node in turn may reference a `Node` allocated at line 17 as left child, while the right child is a node allocated at line 20. These two latter objects may in turn have left and right children, which are still allocated at lines 17 and 20 respectively. It can be noted that the associated class diagram is much less informative, in that the three elements `Node-13, Node-17, Node-20` are collapsed into a single element (`Node`) with two auto-associations (`left` and `right`).

2.2 Dynamic analysis

The dynamic construction of the object diagram can be achieved by executing the target program on a set of test cases with a tracer program activated. The tracing facilities required are basically the possibility to inspect the address of the current object and the content of its fields each time a method is invoked on an object and its statements are executed. Available tracing and debugging tools typically provide such kind of functions.

Each program execution is thus associated with an execution trace, the analysis of which produces an object diagram. Consequently, the outcome of the dynamic analysis is a set of object diagrams, each associated with a test case, providing information on the objects and the relationships that are instantiated with each test case. Their construction from the execution trace is straightforward. The address of each object (content of `this`, dumped into the trace) is an identifier of the objects created during execution. The content of the fields of pointer type is directly the address of the referenced objects, so that the values of the pointer fields dumped into the execution trace give directly the identifiers of the objects referenced by the current object.

With reference to the code sketched in Fig. 4, let us assume that class `Node` has an extra field, `key`, of type `char*`, on which the binary tree is kept sorted. It is possible to execute the binary tree program with some alternative sequences of keys as inputs. For example, the three sequences `("a", "b", "c")`, `("b", "a", "c")`, `("c", "b", "a")` can be associated to three test cases. The analysis of the related execution traces produces the three object diagrams depicted in Fig. 6. In the first case all child nodes are added on the right. In the second case the tree is balanced, while in the last case only left children are present.

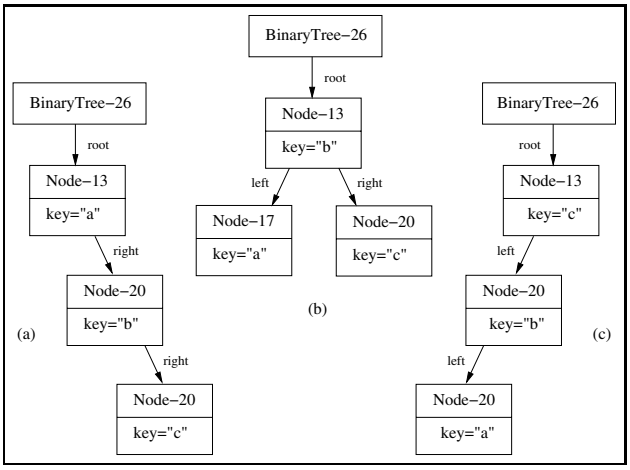

Figure 6. *Dynamic object diagrams produced in three executions of the program in Fig. 4 (left).*

2.3 Discussion

Static and dynamic extractions of the object diagram produce different but complementary information about the instantiations of the classes performed by the program. The static object diagram gives a conservative view of the objects that are possibly created by the program and of the relationships that may exist between the objects. The number of objects reflects the number of program locations where an allocation statement is present. If such a statement is executed multiple times, the actual multiplicity of the related object is greater than the multiplicity indicated in the static object diagram (i.e., 1). The presence of a relationship between two objects in the static object diagram indicates that there is some path in the program along which the first object may reference the second one (through some of its fields). The existence of a path in the program does not imply that such a path is traversed in every execution. As a consequence, the relationships between objects indicated in the static object diagram are a conservative superset of those actually instantiated at run time. Moreover, it may happen that some of these relationships are associated to paths that can never be followed, for any input value. This is typical of the static analyses: the solution is conservative, but may include infeasible parts, due to mutually exclusive conditions on the input values.

The dynamic object diagram complements the static one,

in that objects are replicated in it each time a same allocation statement is re-executed, thus giving a better idea on their actual multiplicity. However, such a diagram is always partial, being based on a limited and necessarily incomplete set of test cases. An indication of the parts of the object diagram not yet explored can be obtained by contrasting it with the static object diagram. Objects and relationships in the static object diagram that are not represented in the dynamic one are associated respectively to allocation statements and execution paths not exercised by the available test cases.

As depicted in Fig. 5, the example program has a static object diagram with 4 nodes and 7 edges. The first test case executed on it (Fig. 6.a) instantiates its objects in 3 out of the 4 locations identified statically. Allocation of a `Node` at line 17 is not exercised in the first test case. Consequently, the two edges leaving `Node-17` in the static object diagram and the two incoming edges are not represented in the first dynamic object diagram. However, the first dynamic object diagram provides some additional information on the multiplicity of the object `Node-20`, which appears to be greater than 1. On the contrary, a unitary multiplicity seems to be confirmed for `BinaryTree-26` and `Node-13`.

The second test case generates a dynamic object diagram (Fig. 6.b) in which all objects in Fig. 5 are represented. The last test case (Fig. 6.c) reveals that the multiplicity of `Node-17` can also be greater than 1.

The comparison of the diagrams in Fig. 6 with that in Fig. 5 highlights the different and complementary nature of the information they provide. The actual shape of the allocated objects (a tree) becomes clear only when the dynamic diagrams are considered. However, they cannot be taken alone, since they do not represent all possible cases that may occur in the program. Inspection of the static object diagram allows detecting portions of the code not yet exercised, which are relevant for the construction of the objects and of the inter-object relationships, and therefore could contribute to the understanding of the object organization in the program.

2.4 Testing

In addition to reverse engineering, the extraction of static and dynamic object diagrams can be conducted in the context of object oriented testing. Two novel object oriented testing criteria can be defined, as a sub-product of the analyses for the recovery of the object diagram:

Object coverage: Every object in the static object diagram is instantiated during the execution of at least one test case.

Inter-object relationship coverage: Every relationship in the static object diagram is instantiated during the execution of at least one test case.

Coverage of all objects in the static object diagram is always desirable, since it ensures that every object creation that may occur in some execution has been tested. This coverage criterion is subsumed by the statement coverage testing criterion. Actually, it is a weaker criterion requiring statement coverage limited to the allocation statements.

Coverage of all relationships that may exist between pairs of objects is also desirable, since it enforces the execution of all paths responsible for the assignment of object references to some fields of a given object. This stronger criterion is not directly comparable to any of the classical coverage criteria (branch, def-use, etc.), except, of course, for the all path coverage which ensures execution of all program paths, but which becomes impractical if infinite paths (due to loops) are present. Therefore, the inter-object relationship coverage criterion is complementary with respect to the traditional coverage criteria. Being based on a radically different view of the program, it leads to exercising the program differently, thus increasing the possibility of revealing different kinds of defects. It is particularly suited to object oriented programming, since it considers object creation and it enforces a deep examination of the relationships that may exist between the created objects.

By contrasting the dynamic diagrams in Fig. 6 with the static diagram in Fig. 5, it is clear that object coverage has been achieved (actually, the second test case alone is sufficient for it). As regards inter-object relationship coverage, 2 out of the 7 relationships in Fig. 5 are not covered by any test case. They are the edge from `Node-17` to `Node-20` labeled with `right` and the edge from `Node-20` to `Node-17` labeled with `left`. Two possible sequences of key values that force their construction are: `("c", "a", "b"), ("a", "c", "b")`. Execution of the program also in these cases provides a higher confidence on its ability to properly handle also these inter-object relationships. Other, more traditional, coverage criteria, such as branch coverage and all-uses coverage, are reached without the need of the two extra test cases, since the first three test cases are sufficient. Consequently, in this example traditional white box testing may not reveal defects associated to some specific inter-object relationships, that are necessarily examined if our testing criterion is adopted.

3 Case study

This work is part of a collaboration between ITC-irst and CERN, the research center for Nuclear Physics in Geneva. The collaboration aims at studying methodologies and tools to improve the quality of the code developed at CERN. In this context, the tool **RevEng** was developed, for the extraction of a set of UML diagrams from C++ code, among which the object diagram.

Recovery of the static and dynamic object diagram was

Lines of code	12223
Implementation files	40
header files	19
Classes	40

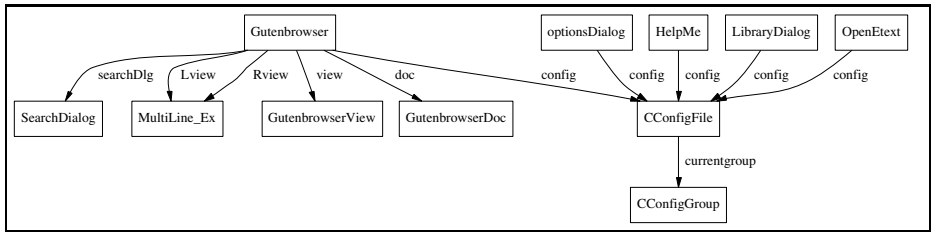

Figure 7. *Features of the C++ program* Gutenbrowser *(left) and portion of its class diagram (right).*

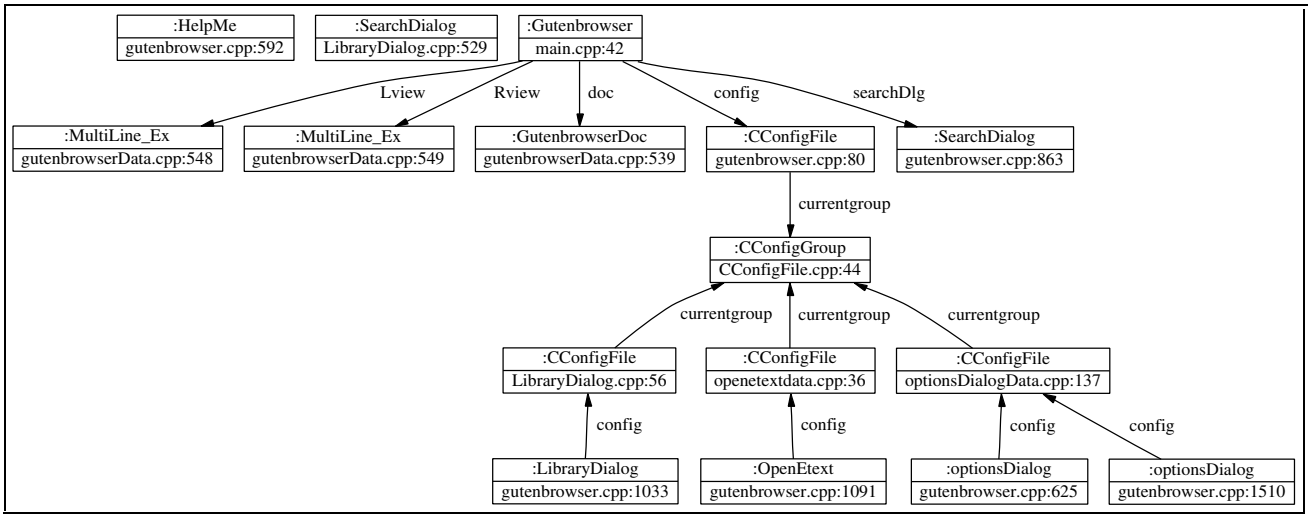

Figure 8. *Static object diagram of the Gutenbrowser application.*

experimented on a real world case study, the C++ application *Gutenbrowser*. **RevEng** was employed for the static analysis, while the program traces necessary for the dynamic analysis were obtained by wrapping the *gdb* debugger. The *gdb-wrapper* automatically defines a set of breakpoints, associated with every block of statements within methods and with every allocation statement, it starts execution and it sends a sequence of continuation requests to *gdb* until execution terminates. When reaching a breakpoint, the wrapper dumps information about the current object, such as its address and the content of its fields.

Gutenbrowser is a medium size C++ program (see Fig. 7, left) of approximately 12 kLOC (Lines Of Code), to easily search, download and read free classic literature, in the form of electronic E-texts. Classic books republished electronically by the Project Gutenberg can be read with *Gutenbrowser*. *Gutenbrowser* is free software, distributed under the GNU General Public License. It can be obtained from the Web site http://sourceforge.net/.

Fig. 7 (right) shows a portion of class diagram of the program *Gutenbrowser*, as produced by **RevEng**. The main class of this program, *Gutenbrowser*, references some graphical widgets such as a dialog window to search books (class *SearchDialog*) and a scrollable text viewer for the display of the book pages (class *MultiLine_Ex*). Other graphical elements are not directly accessed through fields of class *Gutenbrowser*. They are created locally to some method and deleted when their processing is finished. Examples of such classes are *LibraryDialog*, handling the selection of a book from a library, and *OpenEtext*, used to issue a request of opening an E-text. The related objects are expected to have a shorter life time than those referenced by class *Gutenbrowser*.

Fig. 8 contains the object diagram of *Gutenbrowser* resulting from the static analysis performed by **RevEng**. It is interesting to note that class *MultiLine_Ex* is instantiated in two places (lines 548 and 549 of *gutenbrowserData.cpp*), the two related objects being referenced by the fields *Lview* and *Rview* of class *Gutenbrowser*. They are associated to the two book pages that are simultaneously displayed to the user, one on the left and the other on the right of the main window. Class *optionsDialog* is also instantiated in two different places. One instance (line 1510 of *gutenbrowser.cpp*) handles the specification of some user options, such as appearence of the windows, ftp list and http browser, while the other instance (line 625 of *gutenbrowser.cpp*) allows downloading a list of books via ftp/http. A configuration object of class *CConfigFile* is instantiated by several classes (*Guten-*

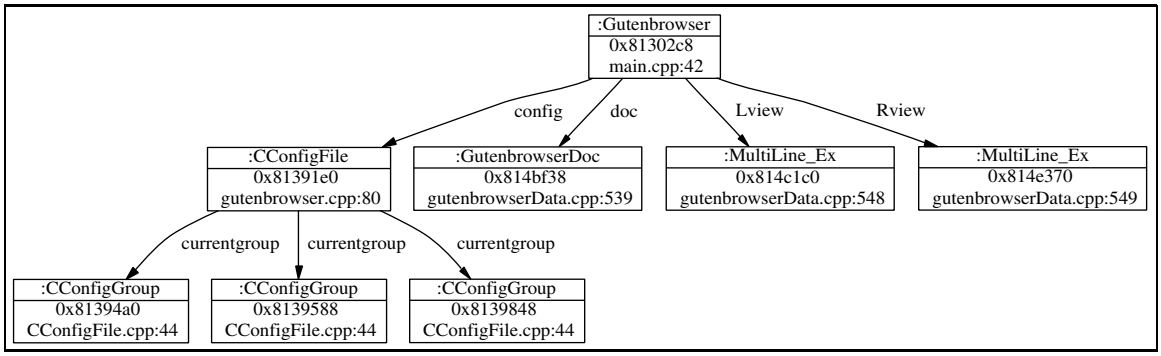

Figure 9. *Dynamic object diagram of the Gutenbrowser application: common part.*

browser, LibraryDialog, etc.) in different places. In turn, it instantiates another configuration object of class *CConfigGroup* at line 44 of *CConfigFile.cpp*.

Some interesting facts emerge by contrasting object diagram and class diagram. The class *GutenbrowserView* is referenced by a field of *Gutenbrowser* in the class diagram, while no object instantiates it in the object diagram. Actually, there is no program statement allocating any *GutenbrowserView* object, which thus seems to be not in use. Although class *HelpMe* has a field (named *config*) referencing a *CConfigFile* object in the class diagram, such a field is assigned nowhere a value in the code. This is the reason for the disconnected object of type *HelpMe*. While class *SearchDialog* plays one role in the class diagram, being referenced by a field of *Gutenbrowser*, it appears to play a twofold role in the object diagram. In fact, a *SearchDialog* is created at line 863 of *gutenbrowser.cpp* and is referenced by class *Gutenbrowser*, but another disconnected *SearchDialog* object is created at line 529 of *LibraryDialog.cpp*. This latter object is temporarily created when the user looks for a book in the library and decides to perform a search.

Three test cases were executed to produce three dynamic versions of the object diagram. The first test case consists of the following user actions: after starting the program, an E-text is selected and opened. Then, some pages of the book are scrolled. In the second test case, the window containing library information is opened, a book is selected and then downloaded from a remote site. In the third test case, the window with the configuration options is opened and some switches are activated/deactivated.

The object diagrams resulting from the analysis of the three execution traces contain a common part which is shown in Fig. 9. For the sake of readability the number of instances of class *CConfigGroup*, reachable from a *CConfigFile* object via the link *currentgroup*, has been reduced from 13 to 3. This applies also to the following diagrams. For each object, the memory address reported by *gdb* is indicated above the file name/line number.

The common part of the object diagram contains all objects referenced by the instance of *Gutenbrowser* that are shown in the static diagram (Fig. 8), except for the *SerachDialog* object, which is not created since the searching functionality was not executed. The multiplicity of the instantiated objects becomes now clearer. While for most of them it corresponds to that in the static diagram, the object of class *CConfigGroup* is allocated several (actually, 13) times, because of several invocations of the method *setGroup* which contains the allocation. In fact, each time a configuration file is read (method *read* of *CConfigFile*), a new *CConfigGroup* object is created.

The portions of the object diagram specific to the three test cases are shown in Fig. 10. They contain a test case specific object – respectively, of class *OpenEtext*, *LibraryDialog* and *optionsDialog* – with the same configuration structure used by the *Gutenbrowser* class object: a *CConfigFile* object referencing 13 *CConfigGroup* objects.

The dynamically computed object diagrams provide important information on the actual object creation that occurs in specific run-time conditions. The mechanism related to the multiple creations of *CConfigGroup* objects becomes apparent. The role of objects such as *OpenEtext*, *LibraryDialog* and *optionsDialog* can be better understood, since it can be associated with a particular functionality exercised by the test case (respectively, reading an E-text, consulting the library and configuring the tool). Moreover, the object diagram views specific to a given test case can be contrasted against the static object diagram, to locate the individual functionality in the framework of a global, conservative view of all instantiated objects.

Object diagram computation was so far regarded in the framework of program understanding, but it is also possible to consider it as a guide to testing. If the object coverage criterion is considered, it is clear from the comparison of Fig. 9, 10 and Fig. 8 that some objects are not instantiated in any of the three test cases (the two *Search Dialog* objects, the *HelpMe* object and the *optionsDialog* object allocated at line 625 of *gutenbrowser.cpp*). To reach complete object coverage, some additional test cases have to be run,

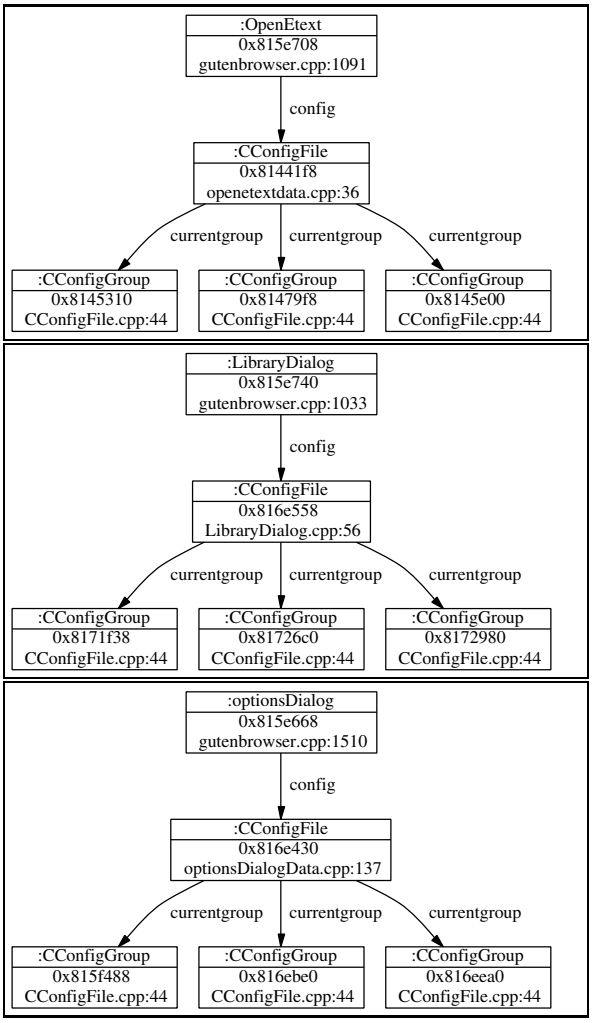

Figure 10. *Dynamic object diagrams of the Gutenbrowser application: test cases 1 (top), 2 (middle) and 3 (bottom).*

which exercise the two searching facilities (from the main window and from the library dialog window), which require help and which download a list of books via ftp. With this additional set of test cases object coverage is achieved, as well as inter-object relationship coverage.

4 Conclusions and future work

The static and dynamic analysis techniques for the extraction of the object diagram presented in this paper have been applied to the C++ application *Gutenbrowser*. The resulting diagrams helped us in the process of recovering information about the system, by providing additional details on the inter-object relationships that are expected to occur at runtime. The presence of classes that are instantiated multiple times/in multiple places and play different roles in the system is not apparent from the class diagram, while it is revealed by the static and dynamic object diagrams. Moreover, the association of the dynamic diagrams with test cases allowed enriching the class instances and the associated relationships with semantic information, obtained as the functionality exercised by each test case.

Future work will be devoted to improving the usability of the recovered information, by providing the user with an interface which supports filtering the extracted information, browsing the results, and searching for specific data. A wider empirical validation on the usefulness of the reverse engineered diagrams is also desirable. Finally, the testing criteria based on the object diagram will be further investigated and compared with other object oriented testing techniques both theoretically and experimentally.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Phd Thesis, DIKU, University of Copenhagen, 1994.

[3] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active test for illustrating object-oriented programs. In *Proc. of International Conference on Software Engineering*, pages 366–375, Berlin, Germany, March 25-29 1996.

[4] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[5] W. D. Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In *Proc. of ECOOP'94 – Lecture Notes in Computer Science*, pages 163–182. Springer-Verlag, July 1994.

[6] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. of the International Conference on Software Maintenance*, pages 13–22, Oxford, England, 1999.

[7] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language – Reference Guide*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[8] P. Tonella and A. Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In *Proc. of the International Conference on Software Maintenance*, pages 376–385, Firenze, Italy, 2001.

[9] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–283, Vancouver, British Columbia, Canada, October 18-22 1998.

[10] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.