klase

Reverse engineering of actor-based systems

ITC-irst

Abstract. Flow propagation inside the object flow graph has been employed to detect the presence of actors (objects) instantiated in the system and communicating with other actors. The message exchange among the identified actors has been represented in the form of interaction diagrams. The algorithm is based on a static, conservative flow analysis, that approximates the behavior of the system in any execution and for any possible input. Applicability of the approach to large software is achieved by means of two mechanisms: partial analysis and focusing.

Document Identifier	Deliverable D6.1
Project	MIUR-FIRB project RBNE0195K5 “Knowledge Level Automated Software Engineering”
Version	v1.0
Date	October 31, 2006
State	Final
Distribution	Public

Copyright © 2006 ITC-irst

Acknowledgements.

This document is part of a research project funded by the FIRB 2001 Programme of the “Ministero dell’Istruzione, dell’Università e della Ricerca” as project number RBNE0195K5.

The partners in this project are: Istituto Trentino di Cultura (Coordinator), Università degli Studi di Trento, Università degli Studi di Genova, Università degli Studi di Roma “La Sapienza”, DeltaDator S.p.A..

Executive Summary

In actor-based systems, functionalities result from the interactions (message exchanges) among the actors (objects) allocated by the system. While designing actor interactions is far more complex than designing the class structure in forward engineering, the problem of understanding actor interactions during code evolution is even harder, because the related information is spread across the code.

In this document, a technique for the automatic extraction of UML interaction diagrams from C++ code is described. The algorithm is based on a static, conservative flow analysis, that approximates the behavior of the system in any execution and for any possible input. Applicability of the approach to large software is achieved by means of two mechanisms: partial analysis and focusing. Usage of this method on a real world, large C++ system confirmed its viability.

Contents

1 Introduction **1**

2 Reverse engineering **3**
2.1 Interaction diagrams 3
2.2 Recovery algorithm 4
2.3 Dealing with incomplete systems 8
2.4 Focusing and numbering 11

3 Case study **16**

4 Conclusions **20**

5 History of the Deliverable **21**
5.1 1st year 21
5.2 2nd year 21
5.3 3rd year 22
5.4 4th year 22

Chapter 1

Introduction

One of the most challenging tasks in actor-based systems is understanding the behavior emerging from a society of communicating actors. In fact, the instructions associated with an overall functionality may be not concentrated in a single place: when several classes contribute to a functionality, the related code is spread among the methods of these classes. The UML (Unified Modeling Language) interaction diagrams [BRJ98] aim at describing the message exchange that occurs among different actors (objects) involved in a given computation. They provide a graphical representation of the control transfers that take place when the overall functionality emerges from the interaction among several actors. These diagrams are useful to describe the expected behavior of a system, during forward engineering, but their availability during software maintenance is potentially even more important. In fact, during maintenance the activity of program understanding is central, and for actor-based systems it may be remarkably difficult to understand the behavior of the objects instantating the actors [LMR92, WH92]. Reverse engineering of the interaction diagrams can support such a task.

In this document, an algorithm is described for the automatic extraction of the interaction diagrams from C++ code. The sequence of method dispatches is considered and their ordering is represented in the two forms of the interaction diagrams [OMG01, BRJ98]: either in *collaboration diagrams*, which emphasize the message flows over the structural organization of the objects, or in *sequence diagrams*, which emphasize the temporal ordering. This work extends a previous paper [TP02], which was focused on a structural view of the objects. While in [TP02] the objects allocated in the code are identified and represented in a diagram, together with their mutual relationships, in the present work not only the objects (actors) are identified, but their message exchange is analyzed to determine their mutual interactions.

A *dynamic* analysis of the interactions among the actors in an actor-based system is a relatively easy task, which requires a tracer to save information about sources and targets of the dispatched methods during execution. However, the diagrams that can be reverse engineered in this way are extremely partial. They hold for a single, given execution

of the program, with given input values, and they cannot be easily generalized to the behavior of the program for any execution with any input. Moreover, dynamic analysis is possible only for complete, executable systems, while in actor-based systems it is typical producing incomplete sets of classes that are reused in different contexts. On the contrary, a conservative *static* analysis produces results that are valid for all executions and for all inputs. We investigated a static code analysis to recover the interaction diagrams, which works also for incomplete systems.

The algorithm described in this document has some properties that makes it applicable to real world, large, industrial systems. It is based on a flow insensitive algorithm that was experienced to be a scalable technique, when used for a different purpose: pointer analysis [And94, Ste96]. Moreover, the proposed algorithm can be applied to incomplete systems and its output can be focused on a computation of interest. In this way, understanding a behavior of interest in a large system can be approached by isolating the components that potentially contribute to it and focusing on the related computations.

Although some works in the literature [KM96, PKV94, RD99, WMFB+98] allow collecting information about class instances and message exchange at run-time, to the best of the authors' knowledge no work tried to attack the problem of reverse engineering the interaction diagrams *statically*, from the source code instead of the execution traces. In [RD99] static information limited to method invocations (call graph) can be combined with execution traces, thanks to a common representation of both data in a single database of logic facts, from which views are created through queries. In [KG88] the call graph is animated by highlighting the currently executing methods. In constructing the Object Relation Diagram, a points-to analysis is employed in [MRR02] to improve the accuracy of a purely type-based computation of the associations among classes. Construction of call graphs for Object Oriented programs and their accuracy are considered in [GC01, TP00].

Chapter 2

Reverse engineering

2.1 Interaction diagrams

Interaction diagrams are used to model the dynamic aspects of an object oriented system [BRJ98]. While class diagrams are used to represent the static structure of the system, in terms of its classes and of the relationships among classes, interaction diagrams are focused on class instances (objects), working together to carry out some task. Their behavior (instead of their static structure) is represented as a sequence of messages that are exchanged among objects. The evolution over time of the method dispatches characterizes the overall behavior.

The elements represented in the interaction diagrams are the objects (actors) created by the program. The interactions among them can be modeled in two ways: by emphasizing the time ordering of the messages (*sequence diagram*), or by emphasizing the sequencing of the messages in the context of the structural organization of the objects (*collaboration diagram*). In the latter case, the Dewey numbering system (incremented integer numbers separated by dots) is used to indicate that a given message triggers the exchange of a set of other nested messages. Thus, if 1 is the sequence number of the first message, 1.1 and 1.2 are respectively used for the first and second nested messages.

Reverse engineering of the interaction diagrams from the code can be conducted either dynamically or statically. Dynamic extraction of the interactions among actors requires the availability of a full, executable system, which is run with some predefined input data. The statements issuing calls to methods are traced during the execution, with information about the physical addresses of the source and target objects. The objects to include in the diagram are thus identified by their address, while the messages exchanged are those traced during the execution. Their sequencing is given by their ordering in the trace file. The main disadvantages of this approach are that it does not apply to incomplete systems, but only to whole, executable ones, and that the resulting diagrams describe the system for a single execution with given input values. A static, conservative analysis of the code for the reverse engineering of the interaction diagrams addresses both problems.

2.2 Recovery algorithm

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

addGen(graph: Graph, stmt: 'lhs = new A()')
1   lhs' ← expand(lhs)
2   N ← N + 1
3   for each l in lhs'
4       addGenToNode(l, {A-N})
5   end for

propagate(graph: Graph)
1   for each n in graph
2       in[n] ← collectOut(pred[n])
3       out[n] ← gen[n] ∪ in[n]
4   end for

expand(x: Var): Locations
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 ← {A::f::x}
10  end if
11  return S
```

Figure 2.1: *Object analysis algorithm working on the object flow graph.*

The static recovery of the interactions among objects is done in two steps: first, the objects created by the program and accessible through program variables are inferred from the code. Then, each call to a method is resolved in terms of the source object and of the target object involved in the message exchange.

A static approximation of the objects created by a program and of their mutual relationships can be obtained by performing a flow propagation inside the Object Flow Graph (OFG), described in more detail in [TP02] and summarized here for completeness.

Objects can be identified statically by the program point where they are allocated. Correspondingly, each allocation statement (e.g., `p = new A()`) will be associated to an object, indicated as `A-N`, with `A` the class it belongs to and `N` a unique identifier (an integer either computed incrementally or equal to the line number). The set of elements `A-N` extracted from a program approximate the set of objects the program may create at run time. The main source of approximation consists of their multiplicity: since it is im-

possible to determine statically the number of times a statement is executed, the expected multiplicity of each object `A-N` is unknown. Relating method calls to objects that are identified by allocation site provides a better approximation than just using the type of the objects on which the invocations are made. In fact, objects allocated at different program points can be distinguished and objects belonging to a subclass of the declared class are assigned the exact type.

Given the set of statically approximated objects `A-N`, in order to be able to resolve method calls, it is necessary to statically approximate the set of objects each program variable might reference (the set of objects referenced by `p` for the call `p->f()`). For this purpose the OFG is built and information about object references is propagated inside it. OFG nodes are program locations (local variables, parameters, class fields, etc.). OFG edges are created each time a statement introduces a data flow from a location to another one. The typical example is the assignment statement `lhs = rhs`, which creates a data flow from `rhs` to `lhs`. Correspondingly, the OFG contains an edge from the node associated with `rhs` to the `lhs` node. Function and method parameters are handled similarly: an edge from each actual parameter to the corresponding formal parameter is created. Moreover, an edge from the target of the message to the `this` location (from `p` to `f::this`, for `p->f()`) has to be created in case of method calls. Finally, values returned from functions and methods introduce data flows that have to be explicitly represented in the OFG. This is achieved by means of an edge from a fictitious `return` location to the location storing the returned value (from `f::return` to `x`, for `x = f()`).

Fig. 2.1 shows the details for OFG edge construction and object creation (procedures *addEdges* and *addGen*), in the case of an assignment statement. A preliminary operation of field expansion (procedure *expand*) is required whenever the involved locations are class fields. In fact, if a location `x` is a class field, it should be represented as the field of one or more of the identified objects (i.e., as `A-N.x`, not just `x`). In order to get an approximate set of objects having `x` as their fields, the currently available output of flow propagation is used. This can be obtained as *out[A::f::this]*, where *A* is the class containing the method `f` which contains the statement under analysis. If *y* is any of the objects referenced by the pointer `A::f::this`, the location accessed through the field `x` will be `y.x`. If a field is accessed through a pointer different from `this` (as in `p->x`), *out[A::f::p]* should be used at line 5 of *expand* in Fig. 2.1 instead of *out[A::f::this]*. The two procedures *addEdges* and *addGen*, given for the assignment statement, can be easily adapted to statements of other types (e.g., function and method calls, and all statements introducing some data flow), as explained above.

OFG construction (procedures *addEdges* and *addGen*) and flow propagation (procedure *propagate*) are executed repeatedly, until a fixpoint is reached. This is because the objects referenced by program variables may influence the result produced by the procedure *expand*, which is in turn used for OFG construction, thus influencing itself the flow propagation outcome. Propagation inside each OFG node (procedure *propagate*) exploits standard flow analysis equations: the incoming flow information (*in[n]*) is collected from the predecessors (*pred[n]*), and is transformed into outgoing information (*out[n]*) by

adding the objects created at n (*gen[n]*).

```
resolveCall(stmt: 'p->g()'): CallPairs
1   A ← class(scope(stmt))
2   f ← method(scope(stmt))
3   sources ← out[A::f::this]
4   targets ← ∅
5   pp ← expand(p)
6   for each p' in pp
7       targets ← targets ∪ out[p']
8   end for
9   return (sources, targets)
```

Figure 2.2: *Algorithm for the static resolution of method calls.*

Once the objects referenced by program locations are obtained by flow analysis, method calls can be resolved by means of the algorithm shown in Fig. 2.2. Given a call statement of the form `p->g()` inside a method `f` of class `A`, the source objects and the target objects of the call are respectively those referenced by the `this` pointer of the current method and by the location `p`. In case `p` is a class field, a preliminary operation of field expansion is again required to get the proper locations (`A-N.p` if `p` is a field of the object `A-N`). Source and target objects, returned by the procedure *resolveCall*, are connected by a *call* relationship.

Fig. 2.3 shows a portion of the C++ code for a Video Rental application, taken from [Fow99]. Each customer (class `Customer`) is associated with a set of rentals (field `_rentals` of type `vector<Rental*>`). A rental contains a reference to the rented movie (field `_movie` of type `Movie*`), as well as the number of rental days (field `_daysRented`). Among the others, class `Customer` has the method that produces a statement (of type `string`), containing customer's data, the list of rented movies with related amount due, the total charge, and the points associated with a frequent renter initiative. A simple `main` function was included to make the whole system run. It creates a `Customer`, a `Rental` and a `Movie`, and it glues them together properly. Finally, it prints the customer's statement to the standard output.

Fig. 2.4 depicts a portion of the OFG for the code in Fig. 2.3. Three objects are allocated by this piece of code, at the lines numbered 37, 38 and 39. They are identified respectively as `Customer-37`, `Rental-38` and `Movie-39`. In the OFG they are inside the *gen* set of the variables on the left hand side of the allocation statements (`main::c`, `main::r` and `main::m` respectively). This information is propagated along the OFG edges until a fixpoint is reached. OFG edges connect two locations if a data flow may occur between them. This is the case of the edge between `main::c` and `Customer::addRental::this`, due to the invocation of `Customer::addRental` on `main::c` at line 42, and similarly of the edge between `main::c` and `Customer::statement::this` (line 43). The edge between `main::r` and `Customer::addRental::act_rental` and that between `main::m` and `Rental::setMovie::act_movie` are due to parameter passing (lines 42 and 41 respectively). The edge be-

```
1   class Customer {
2     string _name;
3     vector<Rental*> _rentals;
4    public:
5     Customer(string act_name);
6     string getName();
7     void addRental(Rental* act_rental)
8        { _rentals.insert(_rentals.end(), act_rental); }
9     double getTotalCharge();
10    int getTotalFrequentRenterPoints();
11    string statement();
12  };
13  string Customer::statement() {
14    ostringstream result;
15    result << "Rental record for " << getName() << "\n";
16    for (int i = 0 ; i < _rentals.size() ; i++) {
17      Rental* each = _rentals[i];
18      result << "\t" << each->getMovie()->getTitle() << "\t" <<
19        each->getCharge() << "\n";
20    }
21    result << "Amount owed is " << getTotalCharge() << "\n";
22    result << "You earned " << getTotalFrequentRenterPoints() <<
23      " frequent renter points";
24    return result.str();
25  }
26  class Rental {
27    Movie* _movie;
28    int _daysRented;
29   public:
30    Rental(int act_daysRented);
31    Movie* getMovie(); { return _movie; }
32    void setMovie(Movie* act_movie); { _movie = act_movie; }
33    double getCharge();
34    int getFrequentRenterPoints();
35  };
36  void main(int argc, char* argv[]) {
37    Customer* c = new Customer("J. Smith");
38    Rental* r = new Rental(3);
39    Movie* m = new Movie("Underground");
40    m->setPrice(Movie::NEW_RELEASE);
41    r->setMovie(m);
42    c->addRental(r);
43    cout << c->statement() << endl;
44  }
```

Figure 2.3: *C++ code for a Video Rental application.*

tween `Rental::setMovie::act_movie` and `Rental-38._movie` is due to the assignment at line 32 and requires the expansion of `_movie` into `Rental-38._movie`, deriving from *out*[`Rental::setMovie::this`] = {`Rental-38`}. The edge between `Customer::addRental::act_rental` and `Customer-37._rentals` requires a slightly more sophisticated analysis, in that it involves the insertion of an object into a container (line 8), instead of a plain assignment. It can be reduced to a plain assignment, with `act_rental` as right hand side and `_rentals` as left hand side, by adopting an approach to container analysis similar to that described in [TP01]. The edge between `Rental-38._movie` and `Rental::getMovie::return` requires the expansion of `_movie` (line 31) into `Rental-38._movie`, based on the *out* of `Rental::getMovie::this`.

Tab. 2.1 contains the results produced by the algorithm for the resolution of sources

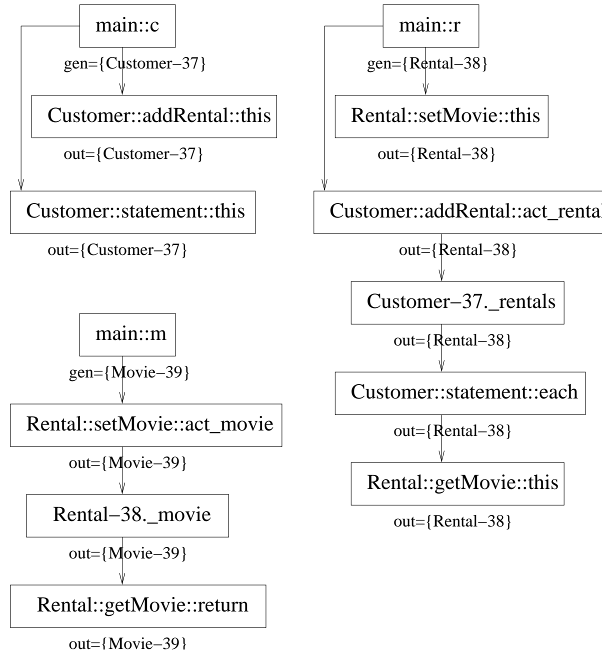

Figure 2.4: *Object Flow Graph for the code in Fig. 2.3.*

and targets of method calls, for all the calls that can be found inside method `statement` of class `Customer`. The source of all the invocations is `Customer-37`, because this is the only object in the *out* of `Customer::statement::this` (see the OFG in Fig. 2.4). When the target of the message is still (implicitly) `this` (calls c_1, c_5, c_6), the resulting target is equal to the source object, `Customer-37`. When the invocation is performed on the variable `each`, the *out* set of `Customer::statement::each` gives `Rental-38` as the target (calls c_2, c_4). Finally, the target object of the call c_3 is the object returned by the previous call (c_2), and can thus be obtained from the *out* set of `Rental::getMovie::return`, which contains `Movie-39`.

Based upon the results of method call resolution in Tab. 2.1, the collaboration and sequence diagrams in Fig. 2.5 can be generated.

2.3 Dealing with incomplete systems

In order to produce complete interaction diagrams, the algorithm described in the previous section assumes that all allocation points are in the code under analysis. This means that the system under analysis comprises all the driver modules necessary to build all of the needed objects. In actor-based systems it is very common to build only an incomplete system, consisting of a cohesive set of interacting classes that perform a given, well defined task, and are expected to be reused in different contexts. In these cases it would be desirable to be able to derive the interaction diagrams even if not all object creations

Call	Line	Called method
c_1	15	`getName`
c_2	18	`getMovie`
c_3	18	`getTitle`
c_4	19	`getCharge`
c_5	21	`getTotalCharge`
c_6	22	`getTotalFreqRenterPts`
	Sources	Targets
c_1	`Customer-37`	`Customer-37`
c_2	`Customer-37`	`Rental-38`
c_3	`Customer-37`	`Movie-39`
c_4	`Customer-37`	`Rental-38`
c_5	`Customer-37`	`Customer-37`
c_6	`Customer-37`	`Customer-37`

Table 2.1: *Resolution of method calls.*

are in the code, to understand the behavior of the incomplete subsystem in isolation, independently of its usages in a given application. To achieve this, all method invocations are taken into consideration and when the source or the target of a call are not associated with any recovered object, although their classes are part of the system under analysis, a generic object is introduced. The result is an interaction diagram in which placeholders for generic objects are present for objects not allocated inside the analyzed code.

Let us consider, for example, the code in Fig. 2.3 *without* `main`. No object is allocated anywhere, and consequently the interaction diagrams are empty. However, the analysis of method `Customer::statement` reveals the presence of 6 invocations (see call graph in Fig. 2.6). Since the type of the sources and targets of these invocations are classes defined in the analyzed code, it would be appropriate to visualize the related interactions among *generic* objects.

Resolution of method calls for incomplete systems is shown in Fig. 2.7. All calls are considered in sequence. Results of flow analysis are used to determine source and target objects (invocation of procedure *resolveCall*). If one or both of the two sets are empty, a generic object associated to the related class is used instead (`A*` indicates a generic object of class `A` or any subclass). In this way call edges are generated even when the object analysis algorithm fails to determine the object issuing or receiving a message. When a non-generic object `A-N` is the target of a call, it cannot be excluded that an externally allocated object be an alternative target of the same call. Thus, if the declared type of the call target is `A`, and `A-N` is the target object, `A*` must be also conservatively assumed as an alternative target, unless further information is available about the excluded code. Moreover, if the excluded code introduces data flows that alter the OFG, it is necessary to take them into account, in order for the result to be still conservative. An example of this situation is the presence of external container classes [TP01]. `A*` indicates that

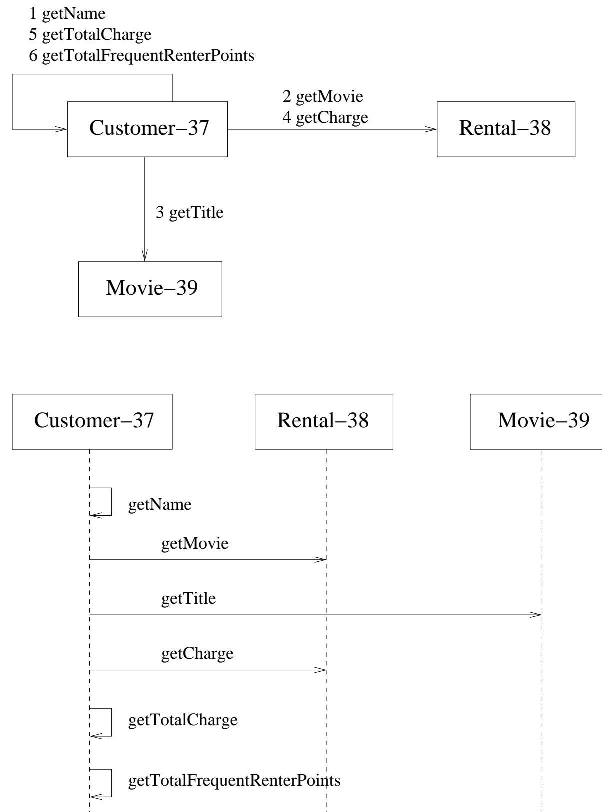

Figure 2.5: *Collaboration and sequence diagrams for the code in Fig. 2.3.*

no allocation point for the given object is in the code. Procedure *expand* in Fig. 2.1 has to be slightly modified to account for generic objects. When the set *out[A::f::this]*, considered at line 5, is empty, `A*` is assigned to `y`, and consequently the location for the field `x` becomes: `A*.x`. In the presence of subclassing, if the allocation point is part of the analyzed code, the allocated object is assigned the exact type (if `A1` inherits from `A` and the allocation expression is `new A1()`, the object will be identified accurately as `A1-N`). On the contrary, when a generic object is introduced because the allocation point is missing, the actual type may be any derived class, and the recovered information is less precise than for objects allocated in the code (`A*` is used for the external allocation of objects of any subclass of `A`, including `A` itself).

With reference to the example in Fig. 2.3, if we assume that the whole `main` function is missing, the *resolveAllCalls* procedure would generate interaction diagrams coincident with those in Fig. 2.5, except for the object names, which would be `Customer*`, `Movie*`, and `Rental*` instead. Since in this example only one allocation point is present in the code for each of the three classes, there is no substantial difference between the diagrams obtained for the complete and for the incomplete system. In cases where more allocation points are present (say, `Movie-40` in addition to `Movie-39`), the approximation introduced by completing the interaction diagrams with generic ob-

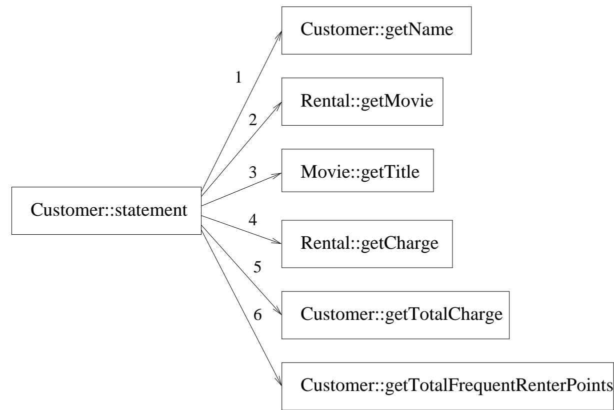

Figure 2.6: *Call graph for the code in Fig. 2.3.*

```
resolveAllCalls(callGraph: CallGraph): CallEdges
1   callEdges ← ∅
2   for each stmt: 'p->g()'
3       (sources, targets) ← resolveCall(stmt)
4       if sources = ∅
5           sources ← {A*}
6       endif
7       if targets = ∅
8           targets ← {B*}
9       endif
10      for each s in sources
11          for each t in targets
12              callEdges ← callEdges ∪ 's →g t'
13          end for
14      end for
15  end for
16  return callEdges

where A = class(scope(p)), B = type(p)
```

Figure 2.7: *Resolution of method calls for incomplete systems.*

jects becomes evident. In fact, all objects of a same class are collapsed into a single object for that class (e.g., `Movie-39` and `Movie-40` would be replaced by a single object `Movie*`). Of course, the proposed algorithm works well in all intermediate cases, in which some allocation points are in the code and some are missing.

2.4 Focusing and numbering

The interaction diagrams in Fig. 2.5 represent the message exchange among objects that is triggered by the execution of the method `statement` inside the class `Customer`. In other words, the view focuses on the interactions occurring when a particular computation (`Customer::statement`) is performed. This corresponds to the natural approach to drawing the interaction diagrams in forward engineering. In fact, it usually makes no

sense to draw just one huge diagram for the whole functioning of the system. It is rather preferable to split it according to the most important subcomputations. This is the key to handle the complexity of large systems.

When interaction diagrams are reverse engineered, the overall plot containing all actors (objects) and all message exchanges may be unusable, because its size exceeds the cognitive abilities of humans even for relatively small systems. However, it is possible to focus the view on subcomputations, thus following the natural approach to the construction of these diagrams. This is achieved by restricting the view to a subset of the calls issued in the program: those belonging to a method of choice. The modification of the recovery algorithm is minimal. After running the procedure *resolveAllCalls*, only the nodes reachable in the call graph from the method of choice are taken into account. The set of call edges returned by procedure *resolveAllCalls* is restricted to the methods in the selected portion of call graph. If this is not enough to produce interaction diagrams of manageable size, the second option available to the user is cutting a part of the system and analyzing an incomplete system, which still encompasses all the key classes involved in the computation of interest. As discussed in the previous section, the introduction of generic objects allows analyzing incomplete systems as well. To summarize, applicability of the proposed approach to large systems can be achieved by filtering the relevant information in two ways: 1) only the calls issued inside a method of interest are resolved; 2) an incomplete system, including only the interesting classes, is analyzed.

Method calls in collaboration diagrams are numbered according to the Dewey notation. Such numbering is exploited also to draw the sequence diagrams, in that the temporal (vertical) ordering is induced by them. It is possible to obtain the proper numbering of method calls by means of the numbering algorithms shown in Fig. 2.8, 2.9.

```
numberCalls(stmtBlock: Statements, num: Integer): Integer
1   for each stmt in stmtBlock
2       if stmt = 'p->g()'
3           callNum[stmt] ← num
4           num ← num + 1
5       endif
6       if stmt = 'if (expr) bk1 else bk2'
7           n1 ← numberCalls(bk1, num)
8           n2 ← numberCalls(bk2, num)
9           num ← max(n1, n2)
10      endif
11      if stmt = 'loop (expr) bk'
12          num ← numberCalls(bk, num)
13      endif
14  endfor
15  return num
```

Figure 2.8: *Numbering of method calls.*

The first step, described in Fig. 2.8, consists of numbering each call edge in the call graph. The first time the procedure *numberCalls* is invoked, it has a method body (block of statements) as first and 1 as the second parameter. An incremental number is associated

with each call statement (line 3) and each nested block of statements is handled similarly to the main block, by recurring inside it (at line 11 only the case of a loop containing a nested block is represented for simplicity). Statements with more than one nested block of statements, such as an `if` statement with both then and else part, require a special treatment, in that the value of the number to use for the first statement following the `if` will be the maximum between the value generated inside the `then` part of the `if` and that generated along the `else` part.

Thus, for a code fragment like this (contained inside a method *f*):

```
if (c) {
  o1->m1();
  o2->m2();
} else {
  o3->m3();
}
o4->m4();
```

```
5   f → m1
6   f → m2
5   f → m3
7   f → m4
```

if *num* is equal to 5 when the `if` is encountered, the absolute numbers attached to the calls to *m1* and *m2* are respectively 5 and 6, the absolute number attached to *m3* is 5, and the next value of *num*, used for *m4*, is 7. The alternative between the two branches of the `if` is indicated by giving them a same initial numbering (5, for both `f` → `m1` and `f` → `m3`).

If the body of `Customer::statement` is passed to *numberCalls*, the numbering represented in Fig. 2.6 is produced.

```
numberFocusedCalls(stmtBlock: Statements,
                   curNum: DeweyNumber)
1   for each stmt in stmtBlock
2       if stmt = 'p->g()'
3           deweyNum ← curNum.callNum[stmt]
4           printNumberedCall(deweyNum, stmt)
5           if g is not on the callStack
6               push(g, callStack)
7               numberFocusedCalls(body[g], deweyNum)
8               pop(g, callStack)
9           endif
10      endif
11      if stmt = 'if (expr) bk1 else bk2'
12          numberFocusedCalls(bk1, curNum)
13          numberFocusedCalls(bk2, curNum)
14      endif
15      if stmt = 'loop (expr) bk'
16          numberFocusedCalls(bk, curNum)
17      endif
18  endfor
```

Figure 2.9: *Numbering of method calls focused on a method.*

The second step in the generation of the Dewey numbers for the collaboration diagram, described in Fig. 2.9, assumes that the view is focused on some method. Corre-

spondingly, *numberFocusedCalls* is invoked with the body of the selected method as the first parameter, and an empty Dewey number as the second. When a call is encountered, the related Dewey number is obtained by concatenating the current Dewey number and the number of the call, separating them with a dot. The new Dewey number generated for the call is passed to a recursive invocation of *numberFocusedCalls*, executed on the body of the called method. Computation of the Dewey numbers inside the called method is not activated in case recursion is detected (check at line 5). For the other statements (lines 11 through 17), the procedure just enters each nested block of statements, where it is reapplied.

```
1   double Customer::getTotalCharge() {
2     double result = 0;
3     for (int i = 0 ; i < _rentals.size() ; i++) {
4       Rental* each = _rentals[i];
5       result += each->getCharge();
6     }
7     return result;
8   }
9   double Rental::getCharge() {
10    return _movie->getCharge(_daysRented);
11  }
```

Figure 2.10: *Bodies of methods* `Customer::getTotalCharge` *and* `Rental::getCharge`.

Let us consider the example in Fig. 2.3 with the addition of the body of `Customer::-getTotalCharge` and `Rental::getCharge`, the code of which is given in Fig. 2.10. The only call inside the first method (to `Rental::getCharge` at line 5), will have an absolute number equal to 1. Similarly, the call at line 10 to `Movie::getCharge` has an absolute number equal to 1. When Dewey numbers are determined for a collaboration diagram focused on `Customer::statement`, the call to `getCharge` at line 19 of Fig. 2.3 propagates a Dewey number equal to 4 inside the recursive invocation of *numberFocusedCalls* on the body of `Rental::getCharge`. Here, the call to `Movie::getCharge` (line 10, Fig. 2.10) will be numbered 4.1. Then, statement 21 of `Customer::statement` is encountered, resulting in the re-application of *numberFocusedCalls* on `Customer::getTotalCharge` with current Dewey number equal to 5. The call at line 5 of Fig. 2.10 in turn triggers another invocation of *numberFocusedCalls* with current Dewey number 5.1. Inside `Rental::getCharge`, the call to `Movie::getCharge` (again, line 10 of Fig. 2.10) will thus be numbered 5.1.1. The final numbering for the resulting collaboration diagram is given in Fig. 2.11.

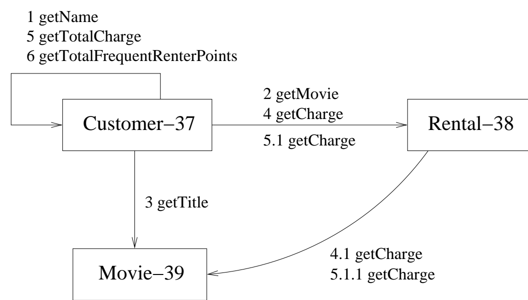

Figure 2.11: *Collaboration diagram for the example in Fig. 2.3, 2.10 with Dewey numbers.*

Chapter 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 interaction diagrams.

The capabilities of **RevEng** to produce usable interaction diagrams for real world systems have been preliminarily tested on the offline C++ code developed at CERN within the Alice experiment, one of the five experiments being installed at the Large Hadron Collider (LHC) in Geneva. The offline software of the Alice experiment is devoted to the simulation, reconstruction, and analysis of the data that will be produced by the experiment.

LOC	subsys	modules
456165	27	974

Table 3.1: *Size of the Alice experiment code.*

In total, 429 351 Lines Of Code (LOC) have been analyzed (see Table 3.1), including comments and headers. The overall system can be naturally split into 27 subsystems, following the physical structure of the subdirectories. The number of implementation files in the analyzed system (headers excluded) is 974. These numbers indicate that the analysis task conducted is a challenging one.

In order to generate manageable diagrams, each of the subsystems was considered in isolation, and within each subsystem, each computation (i.e., the body of each method) was considered separately, by focusing the interaction diagram on it. In total, 3247 focus operations have been (automatically) performed.

Fig. 3.1 shows the number of nodes in a focused interaction diagram on the horizontal axis, and the number of graphs containing such a number of nodes as the vertical displace-

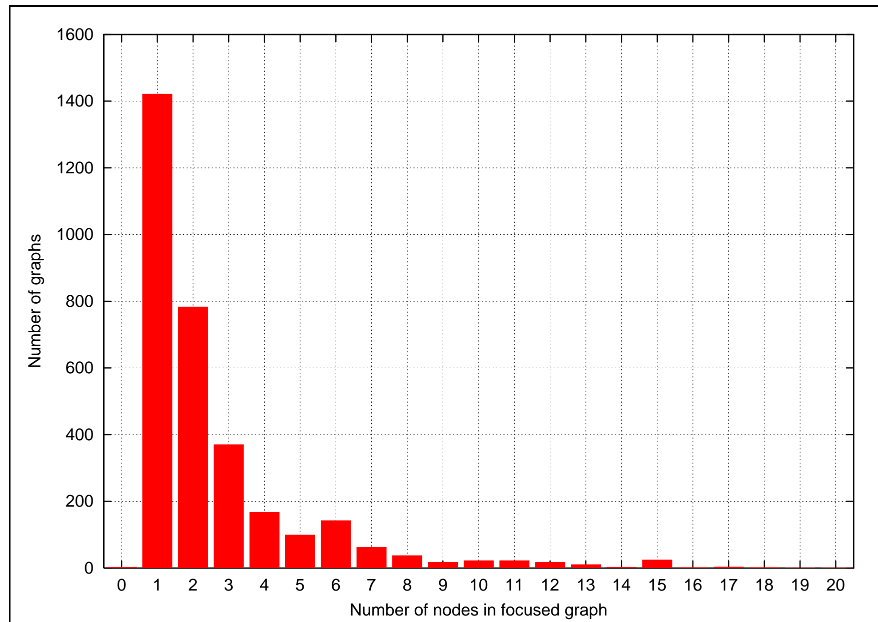

Figure 3.1: *Histogram of the focused graphs with a given number of nodes.*

ment of each histogram bar. It is apparent from Fig. 3.1 that most of the focused graphs have a size (number of nodes) which is below 6, largely within the cognitive capabilities of a human being. This means that partial analysis and focusing have been effective. The interaction diagrams associated with each computation are of manageable size. At the same time, the objects created by the excluded subsystems are represented as generic objects, if the classes they belong to are part of the subsystem under analysis, and the related information is not lost, but just approximated. On the contrary, the size of the interaction diagrams produced for the subsystems, with no focusing applied, is often much larger, as indicated in Table 3.2. For the whole system (no partial analysis applied), the situation is expected to be even worse.

While for some subsystems (e.g., *ALIFAST*) the interaction diagrams could be displayed without focusing, there are several cases (e.g., *ITS*) in which the focusing mechanism is essential to support the practical usage of the information in the extracted diagrams. This means that in some cases partial analysis alone was not sufficient to restrict the scope of the investigation to a manageable size.

Fig. 3.2 contains the collaboration diagram automatically extracted by our tool for the method `MakeTrackCandidatesWithOneSegmentAndOnePoint` of the class `AliMUONEventReconstructor`. All the objects in this diagram are generic, thus indicating that they are not allocated inside the subsystem under analysis (note that generic objects are created only for classes under analysis, not for library/external classes). Without generic objects an empty diagram would have been produced. Guards have been generated for invocations inside conditional statements (the asterisk indicates a loop).

Fig. 3.3 depicts the collaboration diagram focused on method `Update` of class `Ali-`

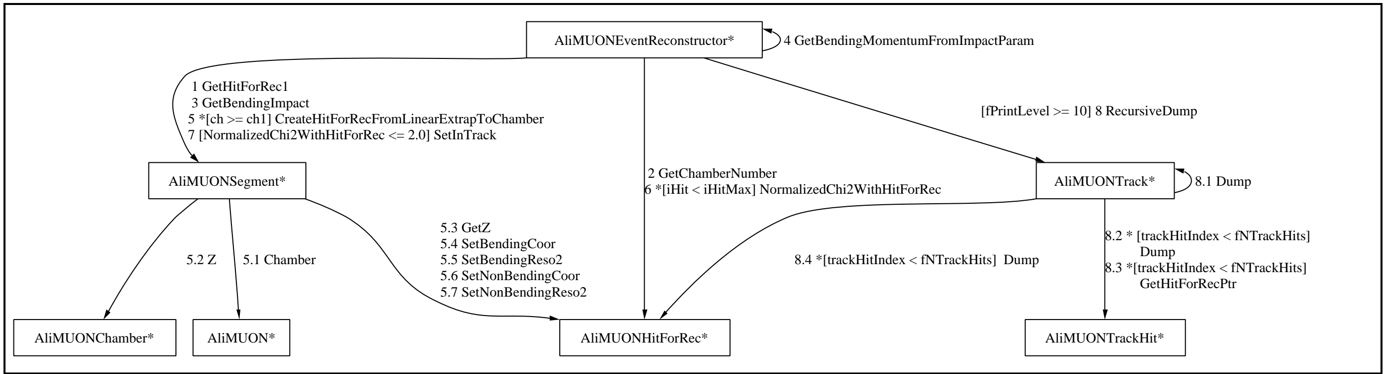

Figure 3.2: *Collaboration diagram focused on method* `MakeTrackCandidatesWithOneSegmentAndOnePoint` *of class* `AliMUONEventReconstructor`*, taken from the subsystem* `MUON`*.*

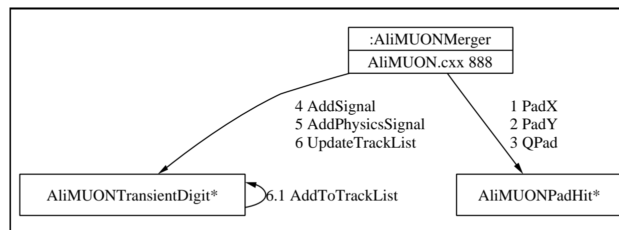

Figure 3.3: *Collaboration diagram focused on method* `Update` *of class* `AliMUONMerger`*, taken from the subsystem* `MUON`*.*

`MUONMerger`. This second example shows the importance of mixing generic objects with objects for which the allocation point is known. In fact, while for one object the allocation statement is available in the analyzed code, this is not the case of the other objects in the diagram, that could be approximated thanks to the use of generic elements.

Table 3.3 gives some metrics indicating how the information represented in an interaction diagram is spread across the code. The diagram in Fig. 3.2 (resp., 3.3) can be obtained manually by inspecting 7 (3) files, and, in these files, 19 (9) methods. All related operations of file opening, code reading and program understanding are eliminated if the interaction diagrams are automatically extracted from the code.

The two diagrams above have been shown to the author of the code, who found them meaningful, in that:

- the views are compact and understandable;
- the information summarized in the diagrams is spread across several modules/classes in the code;
- interactions among objects are mixed with ongoing computation in the code, while they are abstracted in the diagram.

Subsystem	Nodes	Subsystem	Nodes
ALIFAST	9	AliGeant4	51
CASTOR	2	CONTAINERS	36
EMCAL	27	EVGEN	38
FMD	11	HBTAN	122
ITS	211	MUON	105
PHOS	73	PMD	5
RALICE	85	RICH	53
START	8	STEER	47
STRUCT	20	TGeant3	43
TGeant4	121	THbtp	1
THijing	1	TMEVSIM	3
TOF	22	TPC	84
TRD	78	ZDC	10

Table 3.2: *Size (nodes) of the interaction diagrams generated for each subsystem.*

	Files	Methods
Fig. 3.2	7	19
Fig. 3.3	3	9

Table 3.3: *Number of files and methods to be inspected to recover Fig. 3.2/3.3.*

Chapter 4

Conclusions

The proposed technique for the reverse engineering of the interaction diagrams has been applied to about half million lines of C++ code. To generate diagrams of manageable size, two complementary techniques have been exploited: partial analysis and focusing on methods. Combined together, they have been fundamental to produce usable diagrams. The resulting views have been evaluated by the author of the related code, who judged them extremely informative and able to summarize information spread across the code.

In the ongoing work, we are investigating focusing at the class level (selection of a given class together with all the classes it uses). We are also considering method invocations in the context of the state change produced on the target objects. Interactive facilities to hide some and aggregate other parts of the diagram will be developed to enlarge the visualization toolkit that is necessary when real systems are analyzed.

Chapter 5

History of the Deliverable

In this section, the history of the deliverable is described along the four years of the KLASE project.

5.1 1st year

During the first year of the KLASE project, a preliminary activity was conducted to determine how entities in actor-based systems map onto implementation entities that can be analyzed by means of reverse engineering techniques. Various implementation platforms have been considered and various development environments explored. The result of this survey is that:

1. actors can be safely mapped to objects in object-oriented systems;
2. communication among actors can be safely mapped to message exchange (method invocations) among objects in object-oriented programming.

5.2 2nd year

The second year of the project was devoted to the theoretical definition of the analysis framework necessary for reverse engineering and of the related algorithms. Properties of the involved algorithms and alternative approaches have been evaluated thoroughly to choose the best trade-off between analysis costs and accuracy of the results. The framework resulting from this activity comprises:

1. the OFG model of the inter-object data flows occurring in the system and retrieved by means of a static code analysis;

2. a flow propagation algorithm, which determines the objects referenced by each program variable;

3. Dewey numbering and focusing algorithms, for the temporal ordering of the interactions.

5.3 3rd year

The third year of the project was mainly devoted to the implementation of the proposed techniques into a tool that could be used with real software. The target programming language was C++, because a case study with end-users of the technique was available in this language. The parser and the static code analysis required by the reverse engineering tool have been developed completely at IRST.

5.4 4th year

The last year of the project was mainly devoted to the in-field usage of the reverse engineering tool. Some of the results obtained from such case studies are reported in this document and represent the final validation of the effectiveness and usefulness of the proposed approach.

Bibliography

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. Phd Thesis, DIKU, University of Copenhagen, 1994.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language – User Guide*. Addison-Wesley Publishing Company, Reading, MA, 1998.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[GC01] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, November 2001.

[KG88] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proc. of OOPSLA'88, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 191–205, November 1988.

[KM96] 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.

[LMR92] 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.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Constructing precise object relation diagrams. In *Proc. of the International Conference on Software Maintenance (ICSM)*, Montreal, Canada, October 2002. IEEE Computer Society.

[OMG01] OMG. Unified modeling language (UML) specification, version 1.4. Technical report, Object Management Group, September 2001.

[PKV94] 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.

[RD99] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings of the International Conference on Software Maintenance*, pages 13–22, Oxford, England, 1999.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. *Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[TP00] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. of OOPSLA, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 264–280, 2000.

[TP01] P. Tonella and A. Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In *Proceedings of the International Conference on Software Maintenance*, pages 376–385, Firenze, Italy, 2001. IEEE Computer Society.

[TP02] Paolo Tonella and Alessandra Potrich. Static and dynamic C++ code analysis for the recovery of the object diagram. In *Proc. of the International Conference on Software Maintenance (ICSM 2002)*, pages 54–63, Montreal, Canada, October 2002. IEEE Computer Society Press.

[WH92] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.

[WMFB+98] 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.