Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers

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

Abstract

UML diagrams, and in particular the most frequently used one, the class diagram, represent a valuable source of information even after the delivery of the system, when it enters the maintenance phase. Several tools provide a reverse engineering engine to recover it from the code.

In this paper, an algorithm is proposed for the improvement of the accuracy of the UML class diagram extracted from the code. Specifically, important information about inter-class relations may be missed in a reverse engineered class diagram, when weakly typed containers, i.e., containers collecting objects whose type is the top of the inheritance hierarchy, are employed. In fact, the class of the contained objects is not directly known, and therefore no relation with it is apparent from the container declaration.

The proposed approach was applied to several software components developed at CERN. Experimental results highlight that a substantial improvement is achieved when the container type information is refined with the inferred data. The number of relations otherwise missed is relevant and the connectivity of the associated class diagrams is radically different when containers are considered.

1 Introduction

Reverse engineering tools provide useful high level information about the system being maintained. Their output diagrams can support the program understanding activities, can drive refactoring and restructuring interventions and can be employed to assess the traceability of the design into the code. Therefore, it is important that the representations recovered from the code be accurate, i.e., exploit all static information present in the code in order to reverse engineer entities and relations.

Although most of the commonly used programming languages are strongly typed, the libraries that implement containers are often designed to collect objects whose type is not declared. The *Collections* framework that is distributed with the new version of the Java library [8] provides several classes and algorithms to group and manipulate `Objects`, i.e., instances of the top level class from which all user classes descend. The ROOT C++ library [3], which is widely employed in High Energy Physics computing, offers several containers and container operations for instances of subclasses of the top level class `TObject`.

When the UML class diagram [11] is automatically reverse engineered from the code, the relationships between classes are determined according to the type of the instance variables and method parameters. In presence of weakly typed containers, such as those mentioned above, a high risk exists of missing some relations based on containers. In fact, no information on the type of the contained objects can be found in the declaration of the containers.

In this paper, an algorithm is proposed for the inference of the container types. The basic idea is that before insertion into a container each object has to be allocated, and allocation requires the full specification of the object type. Symmetrically, after extraction from a container each object has to be constrained to a specific type, in order to be manipulated with type-dependent operations. Flow propagation of the pre-insertion and post-extraction type information results in a static approximation of the container types. Such information can be used to refine the UML class diagram extracted from the code and to recover some of the otherwise missed relations between classes.

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 [9]. One of the planned deliverables in such streamline is a reverse engineering tool able to extract the UML class diagram from C++ code. The accuracy of such diagram can

be improved by analyzing the type of the objects inserted into (or extracted from) ROOT containers. An evaluation of the class diagram improvement produced by the proposed analysis was conducted using CERN code as a case study.

The difficulties in reverse engineering and maintenance of object oriented applications are discussed in [7] and [14]. Although available commercial tools (Rose[1], Together[2], etc.) allow the extraction of the UML class diagram from the code, only few research works [5, 6, 10] have addressed the problem of filtering relevant information and providing accurate and meaningful high level views. The work most closely related to ours is [4], where the container type is analyzed with the purpose of moving to a hypothetical strongly typed version of the Java language. A set of constraints is derived on the type parameters that are introduced for each potentially generic class (e.g., containers). A templated instance of the original class which respects such constraints can safely replace the weakly typed one, thus making most of the downcasts unnecessary and allowing for a deeper static check of the code. The main differences between [4] and our work are purpose and approach. Our purpose is improving the reverse engineered class diagram, while in [4] the purpose is migrating from weakly to strongly typed containers. Our approach is based on flow analysis for container type inference, while in [4] constraints on type parameters are derived. When such constraints are resolved for a particular instantiation of the generic class, the accuracy of the inferred container types is equivalent to ours.

The core analysis algorithm we propose for the inference of the contained object type is based on the type inference techniques described in [2, 12]. Specifically, the algorithm presented in [2] was reinterpreted in the context of flow analysis [1]. In fact, the original formulation of the analysis attaches sets of types to each program variable, and updates them according to the statements encountered in a flow insensitive visit of the instructions. Type information is not explicitly propagated along the edges of a support graph; rather it is modified each time an influencing statement is encountered. In our approach, variables are still associated to container type information, but such information is propagated inside object extraction/insertion graphs. The program statements are used to build the support graphs for flow propagation, instead of inducing an update of the type information of each variable.

The paper is organized as follows: Section 2 describes the container type inference algorithm and discusses its application to C++. In Section 3 the tool developed for the reverse engineering of the UML class diagram from C++ code is presented. The tool embeds the proposed container analysis. Its usage with the code developed for one of the ongoing experiments at CERN, Alice, is discussed in Section 4, where the impact of container analysis on the accuracy of the UML class diagram is evaluated. Finally, conclusions are drawn in Section 5.

[1] Rose is a UML based visual modeling tool commercialized by Rational Software Corporation.

[2] Together is a UML design tool developed by TogetherSoft.

2 Inference of container type

In the following, the algorithm for the inference of the type of containers will be first described with reference to a simple programming language, which abstracts the main language constructs relevant for this analysis. Then, the adaptations necessary to apply it to C++ are discussed.

2.1 Container type inference algorithm

(1)	S	::=	p = [(A)] q;	$\{(q,p) \in E_{EG}, (q,p) \in E_{IG}\}$
(2)		|	p = [(A)] c.extract();	$\{(c,p) \in E_{EG}\}$
(3)		|	c.insert(q);	$\{(q,c) \in E_{IG}\}$
(4)		|	p = new A();	$\{\}$
(5)		|	c.insert(new A());	$\{\}$

Figure 1. *Abstract syntax of the statements, S, in the simple language used to illustrate the algorithm.*

Figure 1 depicts the abstract productions of a mini-language for object creation and manipulation via containers. Productions are numbered on the left and enriched with some additional information on the right, within curly brackets. The square bracket is a meta-symbol of the grammar indicating that the included expansion is optional. All other symbols in the grammar productions are part of the language.

Statement (1) represents an assignment, where variable q is assigned to p and optionally cast (i.e., coerced) to type A. Statement (2) shows one of the typical operations on containers: the extraction of an object and its assignment to a variable, possibly after cast. The next statement is the insertion of the object referenced by q into the container c. The instantiation of class A and the assignment of the resulting object to p has the syntax of statement (4), while the insertion of a newly created object into container c is represented in the last statement of the language.

Information about the type of the objects in a container can be retrieved starting from object creation and type coercion. Such information is propagated along all intermediate assignments that lead, respectively, to the insertion of the new object into a container, or back to the container from which the cast object was extracted. Two directed

graphs are introduced for the propagation of the type information up to the containers, the *Extraction Graph* (EG) and the *Insertion Graph* (IG). The nodes of both graphs (N_{EG} and N_{IG}) are variable or container names, while the edges (E_{EG} and E_{IG}) are all and only those defined on the right of Figure 1. The assignment statement induces an edge from the source variable (right hand side) to the target variable (left hand side) in both graphs. The extraction statement generates an edge from the container to the reference variable in EG, while the insertion of statement (3) is associated with an edge from the variable to the container in IG. In both graphs the presence of a link between two nodes indicates that some data are transferred from the source to the target.

Type information originates at the statements where objects are created or cast (respectively (4), (5) and (1), (2) in Figure 1). A **GEN** set (such a terminology is standard in flow analysis [1]) is associated to each node n in EG and in IG, containing the type information generated at the statements in which such a node appears. Specifically, the **GEN** set of a node in EG or IG is non empty if the variable associated to the node is involved in any of the following instructions:

p = (A) q;	$\{A \in GEN_q, q \in N_{EG}\}$
p = (A) c.extract();	$\{A \in GEN_c, c \in N_{EG}\}$
p = new A();	$\{A \in GEN_p, p \in N_{IG}\}$
c.insert(new A());	$\{A \in GEN_c, c \in N_{IG}\}$

Formally, the **GEN** set of a node is the minimum set satisfying these 4 constraints (it is empty if the node is associated to no constraint). The node actually generating the type information depends on the statement, being, for example, the node associated with the right hand side (q) of the assignment for statement (1), and with the container (c) for (2). Then, type information is propagated inside graphs EG and IG, until the fixpoint is reached, according to the following equations:

$$\begin{array}{ll} IN_n = \bigcup_{p \in pred(n)} OUT_p & \text{if } n \in IG \\ IN_n = \bigcup_{p \in succ(n)} OUT_p & \text{if } n \in EG \\ OUT_n = GEN_n \cup IN_n & \end{array}$$

Each node n stores the incoming and outgoing flow information respectively inside sets **IN** and **OUT**, which are initially empty. Types collected in the **IN** set are those coming from the predecessors of n, $pred(n)$, if n belongs to IG (forward propagation) or the successors of n, $succ(n)$, if n belongs to EG (backward propagation). Incoming information is transformed into outgoing by adding the types generated at the current node. The worst-case complexity of the fixpoint algorithm is quadratic in the number of nodes in IG and EG, times the cost of the operation of set union, which can be implemented very efficiently by adopting proper data structures.

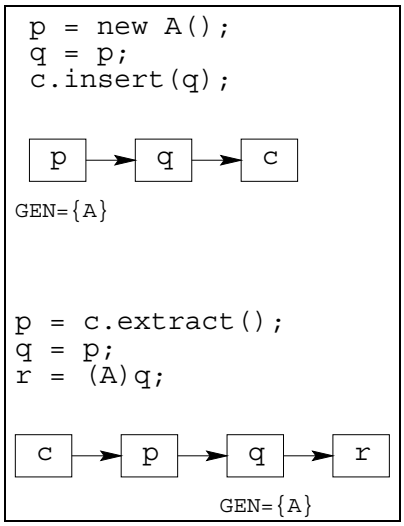

Figure 2. *Example of type information generation inside IG (top) and EG (bottom).*

Figure 2 contains two code fragments in the language defined above. The statements at the top are associated with the insertion graph IG, where only node p has a non empty **GEN** set ($GEN_p = \{A\}$). In fact, p is the target of an assignment whose source is a newly created object of type A. Such object reference is then copied into variables and inserted inside containers according to the edges in IG. Therefore, the type of the objects contained in c becomes apparent after a flow propagation of the information generated at p through q up to c. The fixpoint of the flow equations presented above includes $OUT_c = \{A\}$, thus providing the needed container type.

The statements at the bottom of Figure 2 are associated with an extraction graph in which q has a non empty **GEN** set containing A, since its content is cast to A before being assigned to r. The edges suggest that object references in the container c are copied into p, q and r. A backward propagation of the cast information from q until c produces the needed type information for container c, being its **OUT** set equal to $\{A\}$.

The type information that is computed for a container c may include more than a class. In fact, different type information generated at different nodes may reach the same container. Moreover, the type information resulting from the fixpoint in the IG may be different from that obtained in the EG. To reduce the set of alternative types of objects in a container c to a single type, T_c, the Least Common Ancestor (LCA) in the inheritance hierarchy of the classes is determined:

$$T_c = LCA(OUT_c[EG] \cup OUT_c[IG])$$

Although in a hierarchy more than a common ances-

tor can be found at the same minimum distance, the cases of multiple LCAs in the inheritance hierarchy of program classes are expected to be rare – being associated to multiple inheritance. They can be reduced to a single container type by restarting the LCA computation from the set of least common ancestors found.

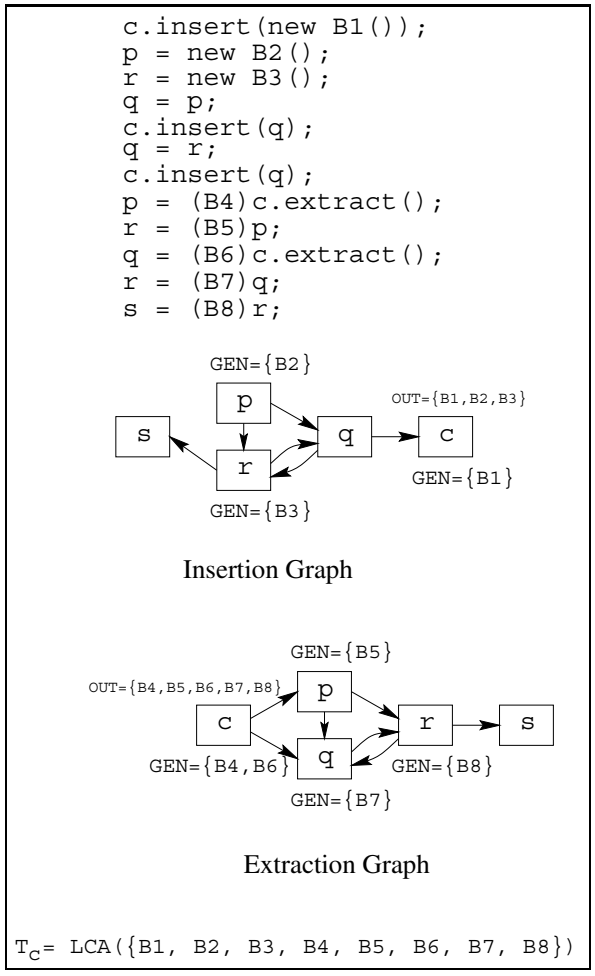

Figure 3. *Example program and associated insertion and extraction graphs. The container type is the LCA of all possibilities collected.*

Figure 3 contains another example highlighting an interesting feature of the proposed algorithm: it is flow insensitive. The statements in the example program can be considered sequentially. In this way, the computation they perform is meaningful (e.g., object insertion is preceded by creation). However, the same insertion and extraction graphs, depicted at the bottom of Figure 3, are obtained when statements are ordered differently, because the procedure for the construction of IG and EG is not sensitive to the order of the statements. Consequently, the results obtained after the fixpoint computation are the same for programs with different control flows and same instructions, and so are the inferred container types.

The choice to adopt an analysis algorithm which is insensitive to the control flow of the program is suggested by efficiency and complexity issues. In fact, the control flow sensitive counterpart has lower performances and is much more complex, especially when calling context sensitivity is required. The only situation in which a control flow sensitive approach is expected to produce more accurate results is when the same variable is reused in different control flow branches to hold objects of different type which are stored in different containers. In such a case the flow insensitive algorithm mixes the different types because of the presence of a shared variable, while a flow sensitive algorithm would propagate the type information only along legal control flows, thus giving different types to the different containers. Therefore, if variables are seldom reused for different aims, the precision of the proposed algorithm is expected to be comparable to that of a control flow sensitive one. The assumption of limited variable reuse seems to be a reasonable one.

Insertion and extraction graphs in Figure 3 contain the indication of the **GEN** sets, associated respectively with object creation and type coercion. By propagating such information along the graph edges (backward in EG) until the fixpoint is reached, two **OUT** sets are determined for the container c in the two graphs: $OUT_c[IG] = \{B1, B2, B3\}$ and $OUT_c[EG] = \{B4, B5, B6, B7, B8\}$. If all the classes in such sets descend, directly or indirectly, from a common ancestor, say B, which happens to be the first common ancestor encountered moving upward in the class hierarchy, then B is the type inferred for container c, i.e., the objects contained in c can be assumed to be of type B (or any derived class), instead of being instances of the top level class.

The knowledge about the type of the objects stored in a container can help improving the accuracy of the UML class diagram extracted from the code. If in the example of Figure 3 the container c is a data member of a class A, an aggregation (or, more generally, an association) relation can be inferred from the container type determined by the proposed algorithm. Since the contained objects are of type B, a new aggregation between class A and class B can be inserted in the UML class diagram of the program, if not already present. Such relation is missed by reverse engineering tools that do not perform container type inference and assume the top level class as the type of the contained objects.

2.2 Application to C++

When moving from the simple language of Figure 1 to a real programming language such as C++, several important

details have to be considered, but no substantial change has to be made to the proposed algorithm.

Being the container type inference insensitive to the control flow, all related statements (conditional instructions, loops, etc.) can be ignored.

Variables have been implicitly assumed to have unique names, while in practice the same name can occur in different scopes indicating different entities. Therefore, the name of the variables has to be augmented with a scope indicator.

An access to objects by *reference* was considered in the examples discussed above, where a reference is a handle leading to the object, as in Java, and as with C++ pointers. Alternative accesses to objects are provided in C++ by the possibility to directly allocate the object on the stack and to define aliases (*references* in the C++ jargon) of already allocated objects. Consequently, some operators for address manipulation are available in C++, as the *address-of* (&) and the *dereference* (*). They should be handled by performing a pointer analysis [2, 12].

In the simple language introduced in Figure 1, variables are not declared before being used. In C++ all variables in a program must be declared, and their type can be exploited as additional source of information about the container types, provided that it is different from the top level class. In such cases it can be inserted in the **GEN** set of the declared variable both in IG and EG.

Class attributes can be manipulated within class methods or directly as the fields of an object (e.g., `x = 1` vs. `o.x = 1`, where `x` is a class field). The two entities have to be unified when performing the container type analysis, and have to be represented by a unique node in IG and EG.

Finally, function and method invocations introduce an association between formal and actual parameters, equivalent to that between left and right hand sides of assignments. In addition, an implicit association is assumed in C++ between the `this` pointer and the object on which a method is called. Moreover, the value returned by a function or method can be assigned to a variable. A special location can be defined to store such a value, say variable `return`, which plays the role of right hand side of the assignment.

```
B* A::f(B1* f1, ..., Bk* fk) {...}
...
x = o->f(a1, ..., ak);
```

$(a1, f1), ..., (ak, fk)$
$(o, this), (return, x)$

Figure 4. *Example of method invocation in C++. The associated edges in IG and EG are shown at the bottom.*

Figure 4 shows an example of method definition followed by its invocation, in C++. For each pairing between actual and formal parameter, an edge has to be added both in IG and EG, representing the information transfer from the method argument to its respective formal parameter. In addition, an edge links the object on which the call is performed to the `this` pointer and an edge connects the `return` location to the left hand side of the assignment. In fact, the type of object `o` has to be propagated to the `this` pointer, which may be inserted into a container, while the return value may, for example, be associated with an object extraction and thus again be useful for the container type analysis. Note that for the sake of simplicity the scope of the involved variables is not explicitly indicated in Figure 4. The full naming of the formal parameters is: `A::f::f1, ..., A::f::fk`, and a similar scope prefix has to be applied to variables `o`, `x`, `this` and `return`.

3 RevEng, a tool for the reverse engineering of C++ code

The reverse engineering tool **RevEng** allows visualizing the class diagram extracted from the code in the UML notation [11]. By analyzing a set of source files, it builds a net of objects associated with syntactic entities and relations found in the code, which can be displayed in a graphical user interface written in Java. It implements the container analysis algorithm described in the previous Section.

The set of objects extracted from the input files is based on a general model of the C++ language, which can be easily extended in an incremental way. A simplified version of this model is represented in Figure 5, using UML.

Each class in the diagram represents an entity of a C++ program: the class *Module* corresponds to the modules composing the program. The set of classes, global variables and functions that can be contained in a module are modeled as aggregations respectively of instances of the classes *Variable*, *Class* and *Function*. In addition, the local variables in a function are modeled through an aggregation with the class *Variable*, while the methods and fields in an instance of *Class* are represented as aggregations of objects of class *Method* and *Variable* respectively. As in the case of functions, the local variables of a method are expressed as an aggregation of objects of class *Variable* and the parameters as an association with class *Parameter*. The statements representing the body of class Method and Function are modeled by means of an aggregation with class Expression. The association with class *SuperClass* leads to the parents in the inheritance hierarchy. Two associations are used to model the entities corresponding to the implementation file (i.e., the file in which the methods are implemented) and the set of required header files. Classes used to model other entities, such as the types defined using `typedef` statements,

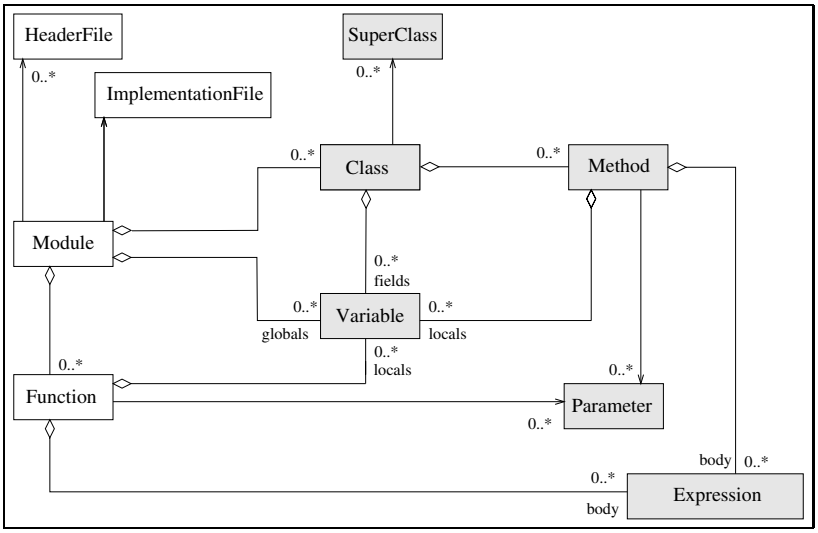

Figure 5. *The C++ language model (simplified version).*

are not reported in the diagram.

When generating the class diagram, only the entities related to the concept of *Class* are considered, among those in the language model. They are depicted in grey.

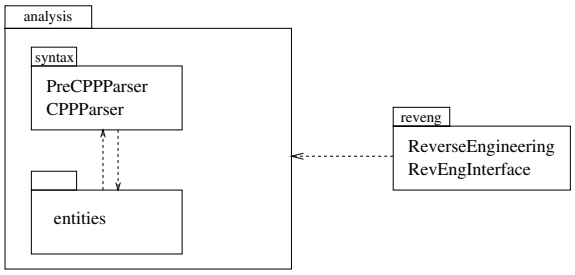

Figure 6. *The architecture of* **RevEng**.

RevEng is composed of two main parts (see Figure 6). The first, whose purpose is analyzing C++ code, is represented by the logical package *analysis*. The second, realized by the package *reveng*, is devoted to creating the class diagram and visualizing it. Package *analysis* is a very general package which gathers information about an input C++ program. It is employed by several different applications, one of which is **RevEng**.

The package *analysis* is subdivided into two packages, *entities* and *syntax*. The first contains the entities in the model of the C++ language (see Figure 5). The syntactic analysis of an input file is performed by the classes *PreCPPParser* and *CPPParser* from the second package, *syntax*. The class *PreCPPParser* performs a preprocessing necessary to the *CPPParser*.

The input to the *PreCPPParser* is a set of preprocessed C++ files (-E option with most UNIX compilers). This operation, necessary to expand the macros, produces a corresponding set of files containing also all the directly and indirectly included files plus additional information for the specific compiler. The class *PreCPPParser* filters the compiler specific directives (as for example `__extension__`, `__const__`, `__attribute__`) and identifies, among all the included files, only those containing the declaration of user classes, i.e., classes whose methods are implemented in the input modules.

In the second parsing round, the model of the language is populated with all the entities found in the identified files. At this point **RevEng** can query the package *syntax* for all the generated entities and can perform its task.

The classes in the *syntax* package are automatically generated by the tool JavaCC[3]. Given an input grammar, JavaCC generates a Java class implementing a top down parser for that grammar. The C++ grammar used for this work was evolved from that freely distributed with JavaCC, on the basis of the grammar specifications in [13].

The package *reveng* contains the classes *ReverseEngineering* and *RevEngInterface*. The class *ReverseEngineering* visits the net of objects created during the parsing phases and selects the entities corresponding to classes, fields, methods and parameters. Note that only user classes are retained during parsing, and are therefore included in the class diagram. In this way, the diagram is not cluttered with library classes (e.g. `string`) and associated relations, and emphasis is set on the architectural decisions and solutions adopted by the designer. By analyzing the instances

[3] JavaCC - the Java Parser Generator is a product of Sun Microsystems, now distributed and supported by Metamata.

of *SuperClass* associated to each *Class* object, **RevEng** creates the generalization relations between classes in the diagram. The type of fields is taken into account to recognize the association/aggregation relations, while the analysis of method parameters and method invocations is used to recover the weaker relation of dependency. When the field type refers to classes defined outside the user code, the associated relations are not shown. Subsequently, the container type inference algorithm described in Section 2 is applied. All class fields whose type is a generic container can be given a contained object type, thus generating relations with the respective classes, which are otherwise missed.

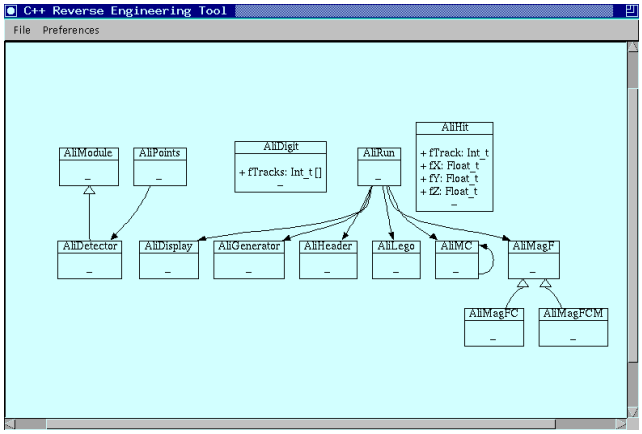

Figure 7. *Class diagram of one of CERN's components. Public fields are shown for every class.*

The class diagram is finally provided to the Dot tool[4], which computes the diagram layout. At this point, the class *RevEngInterface* allows displaying the diagram (see Figure 7). For such a task the Java library Grappa is exploited. The user, operating on the menus of the interface, can select the information to visualize as well as the visualization mode. Given the default view of the diagram, which contains only the class names, the user can decide, for example, to see some or all class fields by expanding one class at a time, or all of them at the same time. Classes of no interest can be removed. Whenever the user performs a change, the layout of the new diagram is automatically re-computed by invoking the Dot program.

When moving from the design to the implementation, aggregations can no longer be distinguished from associations[11]. In fact, acceptable implementations of both relations can be achieved by exploiting class fields of type pointer or reference, or even by directly including an object of the associated class as a field. Of course, containers are another important option, adopted each time the multiplicity of associated objects is greater than one. For such a reason, **RevEng** provides a default interpretation of these relations, which are considered aggregations only in presence of direct inclusion (e.g., `A a;`), but the user is allowed to modify it.

[4] Dot and Grappa are part of the Graphviz package, developed at AT&T.

4 Experimental results

Overall	
Subsystems	23
Implementation files	421
Classes	461
Lines of code	211982

Subsystem	Lines of code	Subsystem	Lines of code
ALIFAST	4203	RICH	11594
AliGeant4	8314	START	1647
ALIROOT	94	STEER	9885
CONTAINERS	4157	STRUCT	9118
CASTOR	1048	TGeant3	10621
EVGEN	10498	TGeant4	12259
FMD	1589	THijing	1967
ITS	30834	TOF	5514
MUON	24630	TPC	21001
PHOS	15590	TRD	14188
PMD	3018	ZDC	1940
RALICE	8273		

Table 1. *Size measures of the code used to test the container analysis algorithm.*

The container analysis algorithm described in Section 2 was applied to the C++ code developed for the Alice experiment at CERN. Table 1 provides some indications on the size of the analyzed modules. The whole system consists of about 212 KLOC (thousands Lines Of Code) and is divided into 23 subsystems. It includes 421 implementation (`.cxx`) files corresponding to 461 classes. The size of the individual subsystems ranges from 94 LOC to 30 KLOC with an average size of about 9 KLOC. Automatically generated code was excluded from the count.

Container classes	21
Insertion methods	88
Extraction methods	128

Table 2. *Features of the containers available from the ROOT library.*

The Alice code exploits the public domain framework

Subsystem	Recall		Precision		Time
AliGeant4	1/2	(50.00%)	1/1	(100.00%)	332.2
ALIFAST	2/2	(100.00%)	2/2	(100.00%)	60.9
CONTAINERS	1/1	(100.00%)	1/1	(100.00%)	59.6
EVGEN	2/2	(100.00%)	2/2	(100.00%)	113.3
ITS	28/29	(96.55%)	28/28	(100.00%)	376.9
MUON	28/28	(100.00%)	28/28	(100.00%)	292.5
PHOS	3/3	(100.00%)	3/3	(100.00%)	207.1
PMD	1/1	(100.00%)	1/1	(100.00%)	43.3
RALICE	12/12	(100.00%)	12/12	(100.00%)	82.0
RICH	19/19	(100.00%)	19/20	(95.00%)	149.9
STEER	6/9	(66.67%)	6/6	(100.00%)	119.8
TGeant3	5/5	(100.00%)	5/5	(100.00%)	104.2
TOF	1/1	(100.00%)	1/1	(100.00%)	72.4
TPC	3/4	(75.00%)	3/3	(100.00%)	207.8
TRD	7/7	(100.00%)	7/7	(100.00%)	164.5
Overall	119/125	(95.20%)	119/120	(99.17%)	2386.4

Table 3. *Precision and recall of the types of the objects inserted into container class fields. Execution times (in seconds) are provided in the last column.*

ROOT [3] for data analysis. This framework provides all the functionalities needed to handle and analyze large amounts of data in a very efficient way. Included are histograming methods in 1, 2 and 3 dimensions, curve fitting, function evaluation, minimisation, graphics and visualization classes to allow the easy setup of an analysis system that can query and process the data interactively or in batch mode. ROOT data collections are based on a set of classes implementing several different containers of objects. All of them are weakly typed and the contained objects are declared to be of type `TObject`, i.e., the top class in the hierarchy underlying the ROOT framework.

As shown in Table 2, ROOT includes 21 container classes with many different methods for object insertion and extraction. An example of one of the ROOT container classes is `TList`, offering the functionalities of doubly linked lists by means of insertion methods such as `Add, AddAt, AddAfter, AddBefore, AddFirst, AddLast`, and extraction methods such as `At, After, Before, First, Last`. Iterators are also available from ROOT to scan the collection of objects in a container and to insert new ones.

Information about C++ entities in the Alice code was obtained by running **RevEng**. Out of the 23 subsystems, 15 contain some class which exploits ROOT containers, in that it declares one or more fields of container type. In total, there are 115 class fields which are containers, for which the computation of the type of the contained objects requires the application of the proposed algorithm.

The container analysis algorithm was run on all Alice code. Specifically, insertion and extraction graphs were built separately for each subsystem, and not for the whole system, assuming that the relative independence between subsystems allows limiting the size of insertion and extraction graphs with a negligible loss of accuracy. Such an assumption was confirmed by the experimental results.

Table 3 shows an evaluation of the accuracy of the proposed algorithm in terms of recall and precision. The *recall* of the algorithm is measured by the number of correct (according to a manual assessment) object types actually retrieved over the total object types to be retrieved. The *precision* measures the number of correct object types retrieved among all types retrieved by the algorithm. Both measures require the availability of the set of correct outcomes, to be compared with the actual results produced by the algorithm. Such reference data were obtained by manually inspecting the source code.

In our experiment, recall and precision levels are high, and 100% of both is reached on several subsystems. In total there are only a few false negatives (6) and false positives (1), leading to an overall recall of 95% and precision of 99%. As a closer analysis has shown, the cases in which the algorithm cannot retrieve the correct type or reports an incorrect type are due to syntactic peculiarities (e.g., dynamic cast) not currently handled, to the need of a preliminary points-to analysis (not performed) to the usage of iterators or to the presence of interactions between subsystems (2 cases).

The order of magnitude of execution time per subsystem is the minute. It can be considered acceptable, since this kind of analysis needs not being conducted with a frequency comparable to that of compilations.

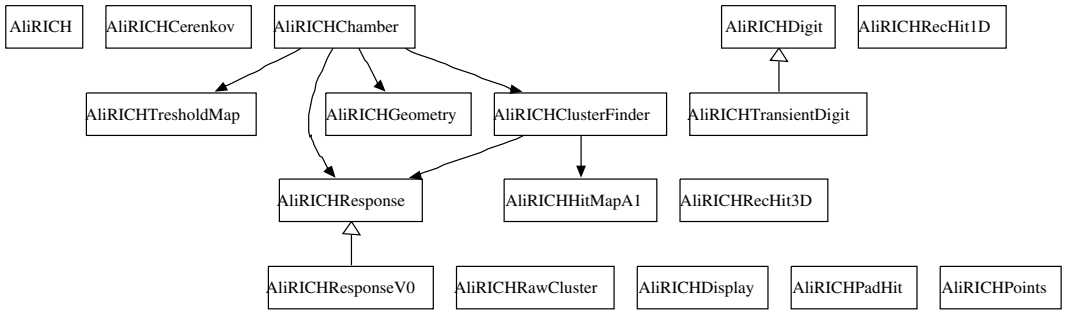

Figure 8. *Portion of class diagram of the subsystem RICH before container analysis.*

Subsystem	Recall		Precision	
AliGeant4	0/0	(100.00%)	0/0	(100.00%)
ALIFAST	2/2	(100.00%)	2/2	(100.00%)
CONTAINERS	1/1	(100.00%)	1/1	(100.00%)
EVGEN	1/1	(100.00%)	1/1	(100.00%)
ITS	24/25	(96.00%)	24/24	(100.00%)
MUON	24/24	(100.00%)	24/24	(100.00%)
PHOS	3/3	(100.00%)	3/3	(100.00%)
PMD	1/1	(100.00%)	1/1	(100.00%)
RALICE	12/12	(100.00%)	12/12	(100.00%)
RICH	14/14	(100.00%)	14/14	(100.00%)
STEER	4/7	(57.14%)	4/4	(100.00%)
TGeant3	5/5	(100.00%)	5/5	(100.00%)
TOF	1/1	(100.00%)	1/1	(100.00%)
TPC	3/4	(75.00%)	3/3	(100.00%)
TRD	7/7	(100.00%)	7/7	(100.00%)
Overall	102/107	(95.33%)	102/102	(100.00%)

Table 4. *Precision and recall of the aggregation relations reverse engineered from the code.*

Since the final purpose of the determination of the contained object types is the extraction of an improved UML class diagram from the code, precision and recall have been re-evaluated with reference to the impact of the analysis outputs on the inter-class relations. For such an evaluation, the set of types associated to contained objects was restricted to those that are user defined classes. In fact, an aggregation relation, stereotyped as `<<contains>>`, can be recovered for each class `A` with a field `f` of container type, whose contained objects are of type `B`, where `B` is another user defined class. The aggregation connects class `A` to class `B`. Table 4 gives the number of correctly retrieved relations over those to be retrieved (recall) and all those retrieved (precision). Results for relations are slightly better than for contained object types, and the overall scores reach 95% for recall and 100% for precision.

Figures 8 and 9 provide an example of improvement in the class diagram, obtained by means of container analysis. The subsystem RICH was reverse engineered and the class diagram extracted from the code was considered before and after container analysis. The different levels of connectivity in the two diagrams are evident from the figures. While in Figure 8 there are many disconnected classes which apparently have no relations with other classes in the diagram, Figure 9 shows that relations between classes do exists and are obtained by means of containers. An aggregation is exploited to represent them, stereotyped as `<<contains>>`. Object insertion into containers appears to be a very relevant means for the construction of inter-class relations, and its analysis has a very remarkable impact on the quality of the recovered class diagram. A dramatic difference was observed in the accuracy of the UML class diagrams before and after container analysis.

5 Conclusion

An algorithm for the inference of the container types was proposed and its application to the improvement of the relations in the UML class diagram recovered from the code was presented. As part of a collaboration between our group at ITC-irst and CERN, a reverse engineering tool working on C++ code was developed; the tool computation of the relations between classes is based on the output of the type inference algorithm. The resulting class diagram is more accurate than the one computed by other available tools, which do not determine the type of the contained objects, thus missing important inter-class relations.

Experimental results suggest that there is a relevant difference between the class diagrams which exploit the inferred container type information with respect to those that do not. A large number of modules, developed at CERN under the Alice experiment, was analyzed with the proposed technique. Since the programming framework adopted by the Alice experiment includes the ROOT library, which offers a variety of weakly typed containers, there was room for the application of our approach. Results indicate that a large fraction of relations is missed if container types are not

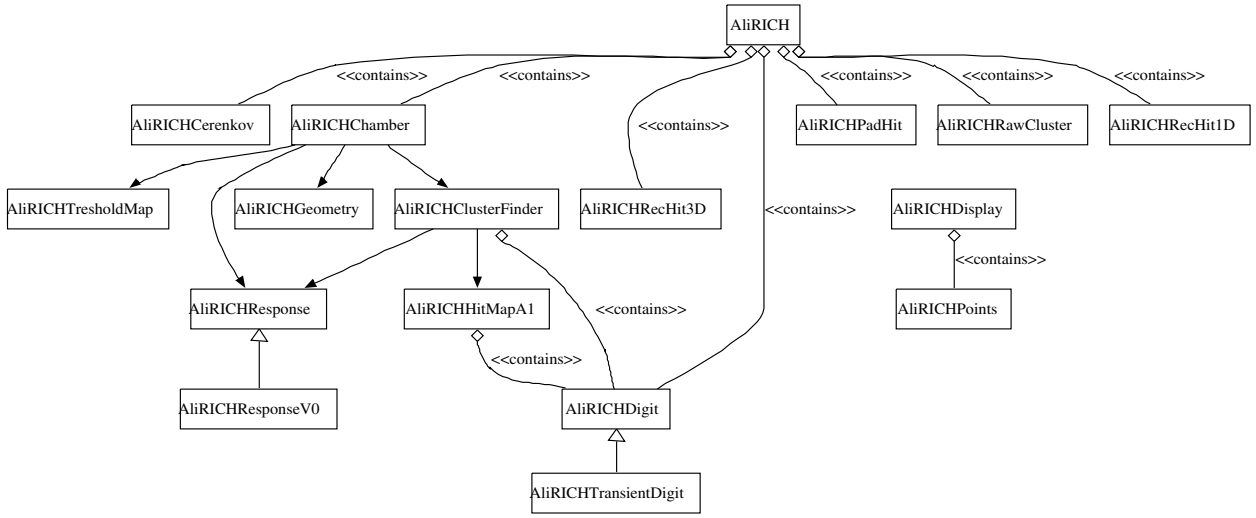

Figure 9. *Portion of class diagram of the subsystem RICH after container analysis.*

determined. Indirect benefits on the program understanding activities are hard to predict. However, the reliability of the reverse engineered diagrams is increased and consequently programmers are likely to trust them much more. In fact, when presenting the diagrams recovered without type inference to the tool users, several known relations between classes were noted to be missing, leaving a feeling of low satisfaction with them. On the contrary, the diagrams of improved quality are much closer to the mental model of the application, being more accurate, and could therefore be used for the high level comprehension of the system and for its evolution.

Acknowledgements

The authors would like to thank Federico Carminati for the stimulating discussions and Bruno Caprile for the useful comments on the manuscript.

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] R. Brun and F. Rademakers. Root – an object oriented data analysis framework. In *Proc. of AIHENP'96, 5th International Workshop on New Computing Techniques in Physics Research*, pages 81–86, Lausanne, Switzerland, 1996.

[4] D. Duggan. Modular type-based reverse engineering of parameterized types in java code. In *Proc. of OOPSLA'99, Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 97–113, Denver, Colorado, USA, November 1999.

[5] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object oriented software. In *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, Monterey, California, USA, 1996.

[6] L. Larsen and M. Harrold. Slicing object-oriented software. *Proc. of the Int. Conf. on Software Engineering*, pages 495–505, 1996.

[7] 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.

[8] P. Naughton, H. Schildt, and H. Schildt. *Java 2: the Complete Reference*. Osborne McGraw-Hill, 1999.

[9] A. Potrich and P. Tonella. C++ code analysis: an open architecture for the verification of coding rules. In *Proc. of CHEP'2000, International Conference on Computing in High Energy and Nuclear Physics*, pages 758–761, Padova, Italy, 2000.

[10] 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.

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

[12] 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.

[13] B. Stroustrup. *The C++ Programming Language (2nd edition)*. Addison-Wesley, 1992.

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