IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661 Volume 2, Issue 6 (July-Aug. 2012), PP 05-12
www.iosrjournals.org

Reusing Run-Time Type Identification (RTTI) Generated By Compiler For Construction Of The Class Hierarchy Graph (CHG).

Sajad Ahmad Bhat
Dept. Of Computer Science, CMJ University Shillong

Abstract:*In statically-typed object-oriented languages optimization plays an important role to make program compilation faster. A number of methods have been used, and a variety of algorithms have been suggested and designed for optimization of statically typed object-oriented languages like C++ and Java. One of the important considerations inoptimization of statically typed object-oriented language is function de-virtualization. Function de-virtualization converts virtual function calls to direct calls, i.e. a virtual function call is replaced by a call to the method of some class. The direct calls can be further inlined to enhance the performance. Class hierarchy Analysis (CHA) [3] is the well-known technique used to identify the virtual calls in a program that can be converted into direct calls. The class hierarchy Analysis starts with a building a Class Hierarchy Graph (CHG) that represents the relation between the various classes of program and the visible methods within these classes. In fact the most basic data structure for developing optimization algorithms is the CHG, which abstracts the Base-derived class relationship that use virtual functions.*

A number of algorithms have been designed for constructingClass Hierarchy Graph (CHG) like the one designed by Bacon, D.F[1]that uses the source level information to build CHG. Several alternatives have been presented to this approach as well. In this article we will present the method for construction of CHG by reusing the RTTI (RuntimeType Identification) generated by the complier.

I. Introduction and Objectives:

A Class Hierarchy Graph represents the inheritance relationship between the classes and represents the various member functions that are inherited/overridden or defined in a class. A Class Hierarchy Graph generally consists of three main sets which are; a set of base classes, a set of derived classes and a set of methods. The classes form the nodes of the CHG connected by edges which represent the derivation relationship between the classes and the methods which are visible in a class are represented inside the class nodes.

Formally a Class hierarchy Graph (CHG) is a three tuple (C, D, and M) were:

- C is a set of classes which represents the nodes of the CHG.
- D is a set of derivations, which are ordered pairs of classes forming the edges of the graph
- M is a set of visible methods that can be invoked through a reference to an object of a particular type.

Any edge in set is an ordered pair of type <b,d> where b□C is a base class and d□C is a derived class. Similarly the each visible method $M_r$in set M is a triplet of the form (c, m, d) where

- c □ C is the class in which $M_r$represents a visible method
- m □ M is the method which in either inherited or declared by c.
- d □ C is the class which defines the m. if the method is declared by the class itself then c=d, or c □ is inherited from the one of the base classes.

Consider the code the following code segment in the listing1.

```
Class L{
Virtual void get () {…some_code…}
};
Class Q : public virtual L{
Virtual void get() {…some-code…)
};
Class N : public virtual L {
};
Class O : public M{
Virtual void put (){…some-code…}
};
Class P : public N, public O{
Void put () {…some_code…}
};
```

Listing 1 :A simple set of C++ class declarations.

The corresponding Class Hierarchy Graph of the above code fragment is as follows.

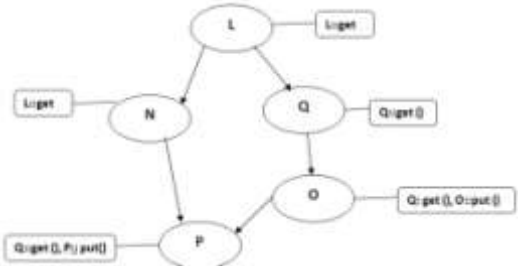


Figure 1: CHG for listing 1.

Class Hierarchy Graph has a vast area of applications in design and development of optimization algorithms. CHG can be used to compute a set of live classes and live methods [1] which have the application in constructing the call graph of a program. The call graph can be used to identify the set of call sites within a program and can help to identify the target virtual calls in a program to be de-virtualized.
The main tasks to be accomplished as the objectives of this work are as follows:

- To show how RTTI generated by compiler can be reused to construct Class Hierarchy Graph.
- To design reliable and efficient algorithms for constructing Class hierarchy Graph over existing ones.
- To know the advantages of Class Hierarchy Graph contraction using this method.
- To calculate the complexity of algorithms for knowing the cost penalty for constructing Class Hierarchy Graph.

II. RTTI (Runtime Type Identification) InStatically Typed Object- Oriented Languages:

RTTI provides some information about the object at the run time such as name of its type. RTTI can be applied to simple data types such integers, charactersetc. as well to the generic objects.RTTI in C++ and Java is the implementation of some more generic concept called **Reflection** or more specifically ***Type-Introspection***. In the original design of C++ BajarneStroustrip does not include the RTTI as a part of it because he thought that this mechanism is frequently misused [14]. In contrast the java has better RTTI system making possible to declare properties for accessing the objects and the language implements persistency. Almost all the C++ compilershave now RTTI support with them. Standard C++ has typeid() operator for getting the type information. The arguments of typeid() operator is an expression which can be a reference or a pointer of an object or a type name. It returns a constant reference to a type_info object containing some information about the type of an object. The pure object-oriented languages like JAVA have a more advanced RTTI system with them. Typically there are two situations when RTTI mechanism is used in JAVA, first is the *downcasting* to a child class and second is the checking the type of an object with the help of an*Instanceof* operator.

III. Class Hierarchy Graph (CHG) and the related works

In the history of analysis and optimization of statically typed object-oriented languages three main algorithms have been published in the literature, Unique Name [12] Class Hierarchy Analysis [13] and Rapid Type analysis [5]. The Unique Name algorithm performs the analysis at the linking time of the program. But both Class Hierarchy Analysis and Rapid Type analysis algorithm uses class CHG as the basic building block for performing analysis. The main motivation of all the algorithms is to resolve the virtual function calls. The Unique Name (UN) was the first published study of virtual function calls by Calder and Gurnwald 1994 [12]. They attempted to optimize the C++ programs at link time, and were confined to the information available in object files. The Class Hierarchy Analysis (CHA) uses both statically declared typed of an object and the class hierarchy of the program to determine the target virtual function calls in a program .Rapid Type Analysis (RTA) starts with a call graph generated by performing Class Hierarchy Analysis it uses information about instantiated class to further reduce the set of executable virtual functions, thereby reducing the size of call graph [5].there are many other algorithms that are related to this work, for example the Variable Type Analysis (VTA) that is a recent work applicable in java. The above mentioned algorithms are the basic and are the corner stone of many other such works.

IV. De-virtualization of Virtual Function Calls

Function De-virtualization is the technique of replacing the virtual function calls by direct calls. The primary focus in optimization of statically typed object-oriented languages is the function De-virtualization.

Virtual function calls are implemented by using the virtual tables and the virtual table pointers. They potentially increase the overhead of calling mechanism and further prevents any form of inlining that could have been possible at the call site. A single virtual function call cost a little complexity but when it is called inside a loop can degrade the performance logarithmically. Consider the following code in listing 2.

```
Class L {
Public: L () { };
~L ()  { };
Virtual void get () { ...some_code….};
};
Class Q: public L {
Public:
B () { };
~ B () { };
}
Void foo (void)
{
 Q q, *bptr;
bptr = &b;
bptr -> get ();
}
```

Listing 2: simple set of class definition in C++

In the listing 2 it is clear the function call to get(), actually calls to the get function of class L. such virtual function calls are usually implemented by Virtual Tables and Virtual Pointers, which increases the significant and reduces the inlining chances at the call site. This virtual call to get() method can possibly be replaced by a direct function call to get() method in class L in the fallowing manner.

bptrt ->L::get ();

At static time automatic De-virtualization in a compiler the knowledge of class hierarchy is required i.e. to know which subclasses have virtual factions that override those declared in their baseclasses/supercalsses. For instance in Listing 1 the call to get() method can have the way L::get() by analyzing that Q does not override the virtual method get() defined in its baseclass L. This is known as "**Class Hierarchy Analysis" (CHA).**The basic data structure in the CHA is the Class Hierarchy Graph.

V. Considerations in building CHG using RTTI

The complexity of building the CHG depends on the visibility of the class definitions. If all the class definitions are visible for example if they are present in a single file, then building CHG is straightforward. However, if the base classes and the subclasses are in different files and CHG is needed to be constructed makes the situation complex. In such cases the compiler depends up the features whereby all the compilation units are readily perceptible. For instance the HP and Intel compilers with the option–ipo or +04 or +OWhole-programe_modefor a host of others. Consider the following code segment in listing 3:

```
#include <file1.h>
Void DoAction(L *ptr)
{
DoOtherAction (ptr->get ()} ;
<file file1.Cp>
Class L {
Public:
L () { };
~L () { };
Virtual void get () {
… some_code….};
};
<file file2.Cp>
# include <file1.h>
Class Q: public L {
Public:
Q() { };
~Q() { };
Virtual voidget() {
…. Some_code….};
};
```

Listing 3: C++ class definitions with overridden virtual function get () and are supposed to be in different files.

In above we have two classes L and Q put in two separate files. We assume that the class L is defined in file1.h and the derived Class Q is in file2.cp. The file1.cp have a function DoAction() as well. Since the class Q overrides the method get () of class L the CHG of above lsiting3 will be as fallows.

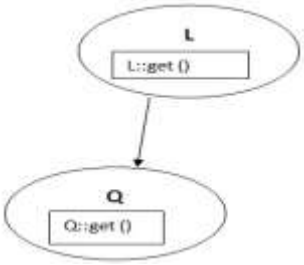


Fig 2: CHG for listing 3.

Since it would be impossible to statistically infer from CHG that the call toptr ->get () in DoAction() can be replaced by ptr -> L::get() as the objects of either class L or Class Q can be passes as actual parameters to the DoAction(). If an object of Class Q is passed as parameter and the call is replaced by ptr -> L::get () then it will result some incorrect code. These types of virtual function callinvocations are also De-virtualized by using a technique called dynamic De-virtualization.

VI. Building Class Hierarchy Graph (CHG)

Building Class Hierarchy Graph is a straight forward process once all the required information to build it is available. Building CHG using RTTI needs to keep track of some static structures like Virtual Tables, Virtual pointers, Class Tables, typeid structures etc. Generally the C++ front end compiler doesn't emit any information to code out class hierarchy. Since most of the compilers support RTTI, and to support RTTI C++ front end produces the type structure of the classes that uses virtual functions. This information generated is encoded in the form of intermediate code generated by a C++ compiler. When optimizer checks this intermediate code, it decodes and constructs the Class Hierarchy Graph from the type structure generated to supportthe RTTI. Since every class that makes use of virtual functions (or a class that is derived from a class which has virtual functions in it) is given a secret data member in the form of a virtual table. Thesetup of this virtual table is done by the compiler at compile time. For each virtual function that may be called by the object of the class, there is an entry as a function pointer in the virtual table. For pure virtual functions in ABC the virtual table stores NULL pointers. Virtual Table is also created for classes that have virtual base classes. In such cases, the virtual Table has a pointer to the shared instance of the base class in addition to the pointers to the class's virtual functions. Another important static structure is the Virtual Table Pointer or _Vptr.The Virtual Table pointer or _Vptr, is a hidden pointer that is added by the Compiler to the baseclass. The main purpose of the Virtual Pointer is that it points to the Virtual Table of the class. The _Vptr is transparently stored by each object of the class with virtual functions. By following this hidden _Vptr the call to a virtual function by an object can be resolved. There may be multiple Vtable (Vitual Table) static structures for multiple or virtual inheritance. For class L in listing 1 there is a static structure named e.g.<Vtble_L> that is created. The <Vtble_L> consists a number of different fields. The**typeid** structure is pointed by one its field pointers. And one of the fields of <Vtble_L> also points to the get () function. The internal class representation of the listing 1 may look like as shown in fig 4.

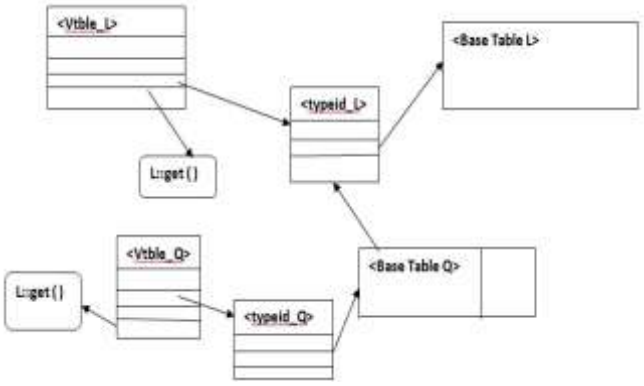


Fig 4: Vtable, typeid and Base-class structure for listing 1.

On analyzing the Virtual Tables, we can find some useful information there. The most important are the static variables of type <Vtble_L>. By checking theseVtable static variables we can deduce the name of classes and of those virtual functions which are visible in these classes. Hence the class nodes of fig 2 and the virtual functions visible within them can be constructed. Since the supercalss/subclass relationship information is not present in Vtable. So the edges of the CHG can't be inserted from information available in Viable. For inserting the derivation edges in the CHG, we have to depend on the typeid structures which are constructed for each class and we should note that Vtable points to **typeid** structure as well. The typeid structure contains the type information of the class and is used for C++ calls likedymaic_cast, typeidetc. This information can be adopted to extract the supercalss-subclass hierarchy. One of the fields in typeid structure points to the Basecalss Tables. The Baseclass Table is nothing but an array of pointers that points directly to the typeid structures of the superclass (typeid structure of the superclass of that class which is under consideration) shown in fig 4.
From fig4 we can infer how the derivation edges can be inserted in the CHG. It is clear that Vtable, typeid and BasecalssTables encode the class hierarchy information. It is apparent in the fig 4 that Base Table of class Q points to the typeid structure of class L, which means that the class L is the basecalss of the class Q. the derivation edge from node L to node Q in fig 2 can now be inserted using this information. Considering the above facts the algorithm for building CHG is as fallows.

1. BuildCHG( ) {
2. For each Vtable (vt) created in a file do {
3. Create a node (nod) for vt if not already created;
4. Add all virtual functions (vrf) that can access though vtr to nod;
5. Say tid = typid of vtr;
6. For each basecalss (bc) of tid do {
7. Find the hierarchy node (hn) corresponding to bc ;
8. Add hn as a baseclass of of nod;
9. Add nod as a subclass of hn;
10. }
}
}

Listing 5: Basic Algorithm to build CHG.

In the case of multiple inheritance multiple Vtalbes are created for the same class. In such a case were multiple Vtalbes are created for the same class the algorithm in listing 5 becomes slightly complex. Therefore algorithm can be modified such that all nodes constructed for the same class can be merged into a single node and to maintain derivation edge information accordingly. Consider the following code:

```
Class L {
L ( )  { }
~L () { }
Virtual void get () {
….. some_code….
}
};
Class Q {
Q ( ) { }
~Q ( ) { }
 Virtual void get ( ) {
Some_code };
};
Class N:public L, public Q {
{
N ( ) { };
~N ( ) { };
Virtual void get ( ) {
….some-code…};
};
Void foo(void)
{
N ob, *bptr;
bptr = &ob;
bptr ->get ( );
```

}

Listing 6: multiple inheritance in C++.

There will be two Vtalbes say <Vtble_N1> and <Vtble_N2> for the class N, and the typeid structure say <typeid_N> of class N will be shared by both the Vtables. Since BuildCHG( ) algorithm creates a node for each for Vtable hence two nodes separate will be created for <Vtble_N1> and <Vtble_N2>. Both the Vtables will contain the same information for the class N therefore created two separate nodes for each <Vtble_N1> and <Vtble_N2> will be the wasteful. So it is suggested and understood to merge this type of nodes to create a single node. If we apply the above algorithm to the listing 6 it will result the following CHG.

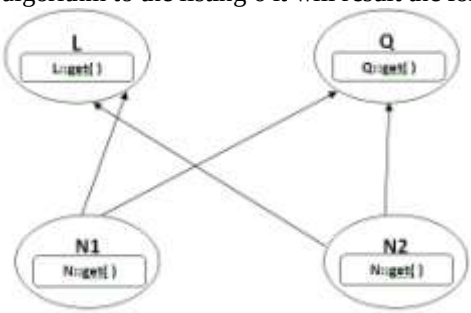


Fig 5: class hierarchy graph of listing 6 after applying CHG algorithm.

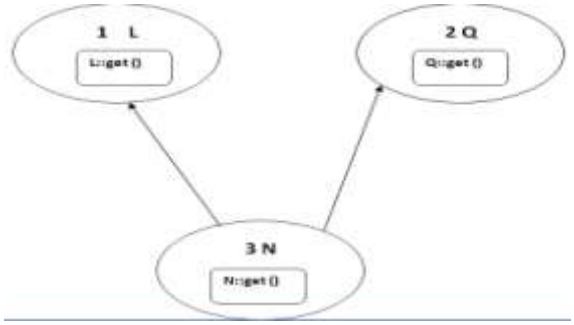
Since an additional node is created for the second Virtual table, this is a wasteful because both the Vtables and their corresponding nodes carry the similar information. Therefore it is better to merge these nodes for class N.

We can make some changes to the algorithm using some basic assumptions to the algorithm in listing 5. The basic draw back to the algorithm is that is creates the redundant nodes which is a wasteful to keep in CHG. To overcome node redundancy problem we can make use of method signatures to identify the nodes with similar information. Once the nodes with similar information are identified it will be easy to merge them into single node. The following listing shows such an algorithm for building CHG without node redundancy.

```
1.  BuildCHG( ) {
2.  Set i=0;
3.  for each Vtable (vt) created in a file do {
4.  Create a node (nod) for vt if not already created;
5.  Assign NodNumber(i) to nod;
6.  Add all virtual functions (vrf) that can be accessedthoughvtr to nod;
7.  Say tid = typeid of vt;
8.  for each basecalss (bc) of tid do {
9.  Find the hierarchy node (hn) corresponding to bc;
10. If not found go to step 3;
11. for each function vrf in nod and vrp in previous node do{
12. If sig(vrfp)= sig(vrfn) // vrfp and vrfn are virtual functions in previous node and new node respectively
13. If bases(nod)=bases(NodP)
14. MergeNodes(nodP,nod);// nodp is previous node with NodeNumber(i-1)
15. Else
16. Add   hn   as a  baseclass of of nod;
17. Add  nod as  a subclass of hn;
18. }
19. }
20. }
21. }
```

Listing 7: Modified algorithm to build CHG.

The algorithm in listing7 assigns a Node Number using NodeNumber (i) function to every node in Class Hierarchy Graph. This Node Number will help to locate the previous Nodes in Class Hierarchy Graph. If the class hierarchy node (hn) is found corresponding to the baseclass (bc) the algorithm starts comparing the function signature in the new node (nod) and previous node (nodP) we assume that each function signature assigned a unique number. In addition to function signatures we compare the baseclasses of current and previous nodes to make sure they are identical and are worth to be merged. Then we begin to merge the nodes using MergeNode(nodP,nod) function otherwise we simply add the node to the CHG. If we apply this algorithm to the code in listing 6 we will get the CHG in fig 6.



VII. What happens when RTTI is disabled:

Some compilers have the facility to disable the RTTI, for example compilers such as g++ have the support for the options like –fno-rtti for instance to deactivate the RTTI information for the code that will not use RTTI functionally. In such cases when the code is known not use RTTI functionally, the compiler still has the ability to generate the RTTI information for optimizer. But, the proceeding passes of the optimizer can destroy the RTTI information after De-virtualization had been put into action. The final executable will not contain thenRTTI information as expected from a –fno-rrti option.

VIII. Complexity of the algorithm

The main purpose of this article is not the study thecomplexityofalgorithms in detail but still we can analyze the worst-case time complexity of the BuidCHG () algorithm. Let's use the notation O(X) instead of O (| X|), this type of notation is used by Bacon D.F [5] for notational simplicity. For worst-case complexity let's assume that sets and mappings data structures are implemented with data structures data allows insert and delete operations to be performed in O(log n) time, were n is the size of set.

The if statements in the line number 12 and 14 in the inner most loop compares the function signatures and base class of two adjacent nodes. This can be done in constant time because the function signatures too are represented by unique numbers as we have assumed in the algorithm. However if the total number of functions are (TF) then time complexity of building unique signatures will be O (TFlogTF). The number of virtual functions in a node is the upper bound to the inner most for loop at line number 11 thus operations within this loop can be performed at the cost of O(log nvrf) were (nvfr)is the total number of virtual functions in a node. The for loop at line number 8 explores the base classes pointed by typeid, hence the maximum number of iterations performed will be the number of base classes that a typeid points to, hence the worst-case complexity will O(log nbc) were (nbc) will be the number of base classes pointed by a typeid. The upper most for loop is to explore the each Vtable. Since the number of Vtalbes provides the upper bound to the number of iterations performed. If denote the the total number of Vtables by N(Vtables) then total number of steps in the algorithm can be expressed as

$O(\sum$ N(Vtables) nbc log nbc))

To observe we use maximum number of Vtables

□=max |N(Vtables)|

And the maximum number of base classes pointed by the typeid of Vtableβ=max|nbc|

Then we can re-write the worst-case running time expression as

O(∑□ βlogβ)

This is equal to

O(C□ βlogβ)were C is the set of Vtables.

IX. Conclusion

In this article we presented methodfor constructing the CHG that uses information emitted by the C++ compiler for RTTI. This method has several advantages over the existing ones.The main advantages are that the front end of C++ compiler needs not to generate any extra information for constructing Class Hierarchy Graph in the optimizer. The informationthat is generated for RTTI is reused for construction of CHG. The RTTI information is maintained by most C++ compilers, as this method has ability to provide a global mechanism for constructing CHG. it is always good to carry de-virtualizations at a late phase, this method can be put into use bypostlink-time optimizersor tools that checksobject code, so as to enable such tools or optimizers to perform

the De-virtualization at a very late phase. This method canefficiently be used to carve out the CHG from the executable/object files, and for this we do not have to depend and any addition information. Since this algorithm is generic enough, so other compilers can easily adopt it as well.

References

[1]. David F. Bacon and Peter F. Sweeney "Fast and Static Analysis of C++ Virtual Function Calls",object oriented programming systems, languages, and applications (OOPSLA) 1996.

[2]. UrsHolzle and GerladAigner "Eliminating Virtual Function Calls in C++ Programs"Conference Proceedings, Springer Verlag LNCS 1098, pp. 142-166 ECOOP 1996.

[3]. Grove, Chambers. And Dean, J., D. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis" Tech Report, Dept. of CSE, University of Washington, 1994.

[4]. David Bernstein, YaroslavFedorov, Sara Porat, Joseph Rodrigue, and EranYahav.Compiler Optimization of C++ Virtual Function Calls. 2nd Conference on Object-Oriented Technologies and Systems, Toronto, Canada, June 1996.

[5]. David F. Bacon, [Fast and Effective Optimization of Statically Typed Object-Oriented Languages] Ph.D. Thesis, University of California Berkeley, 1997.

[6]. Wu, P.-C. And Wan[g], F.-J. 1996. On efficiency and optimization of C++ programs. Softw. Pract. Exper., 26, 4 (Apr.), 453{465.

[7]. Porat, S., Bernstein, D., Fedorov, Y., Rodrigue, J., and Yahav, E. 1996. [Compiler optimizations of C++ virtual function calls]. In Proceedings of the Second Conference on Object-Oriented Technologies and Systems, (Toronto, Canada, June). Usenix Association, pp. 3{14.

[8]. Bacon, D. F., Graham, S. L., and Shar[p], O. J. 1994. Compiler transformations for high-performance computing. ACM Comput. Surv., 26, 4 (Dec.), 345{420.

[9]. Pande, H. D. and Ryder, B. G. 1995. Static type determination and aliasing for C++. Tech. Rep. LCSR-TR-250, Dept. of Computer Science, Rutgers University, (July).

[10]. Pande, H. D. and Ryder, B. G. 1996. Data-flow-based virtual function resolution. In Proceedings of the Third International Static Analysis Symposium, volume 1145 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, pp. 238{254.

[11]. Nackman, L. R. and Barton, J. J. 1994. Base-Class Composition with Multiple Derivation and Virtual Bases. In Proceedings of the 1994 USENIX C++ Conference, (Cambridge, Mass., Apr.). Usenix Association, Berkeley, Calif., pp. 57{71.

[12]. Calder, B. and Grunwald, D. 1994. Reducing indirect function call overhead in C++ pro-grams. In Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages, (Portland, Ore., Jan.). ACM Press, New York, N.Y., pp. 397{408.

[13]. Dean, J., Grove, D., and Chambers, C. 1995. [Optimization of object-oriented programs using static class hierarchy analysis]. In Olthoff, W., Ed., Proceedings of the Ninth European Conference on Object-Oriented Programming { ECOOP'95, volume 952 of Lecture Notes in Computer Science, (Aarhus, Denmark, Aug.). Springer-Verlag, Berlin, Germany, pp. 77{101.

[14]. Ellis, M. and Stroustroup, B. 1990. [The Annotated C++ Reference Manual]. Addison-Wesley, Reading, Mass.