We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Chapter

How to Keep the Binary Compatibility of C++ Based Objects

Donguk Yu and Hong Seong Park

IntechOpen

Abstract

This chapter proposes the binary compatibility object model for C++ (BiCOMC) to provide the binary compatibility of software components in order to share objects among C++ based executable files such as .exe, .dll, and .so. In addition, the proposed model provides the method overriding and overloading, multiple inheritance, and exception handling. This chapter illustrates how to use the proposed model via a simple example in the Windows and Linux environment. The proposed method is validated by application examples and comparisons with known object models such as C++, COM, and CCC in terms of the call time of a method during execution and the binary compatibility such as reusability due to interface version and the types of compilers. Also this chapter shows that BiCOMC-based components compiled with Microsoft Visual C++ and GCC can call each other and the interface version problems are resolved.

Keywords: binary compatibility, component, C++, interface, object model, Windows

1. Introduction

Nowadays, lots of software have been developed based on components because of reusability and composability which can make development and maintenance easier. The component-based approach has some advantages that the cost and time required for maintenance and development can be reduced through the combination of components and the property of encapsulation. In particular, the robot software platforms such as OPRoS [1–3], openRTM [4, 5], and OROCOS [6], which are examples of component-based systems, have been using components of dynamic libraries, such as .dll and .so, in order for components to be able to be developed and maintained with ease. Despite these advantages, there are some hurdles in reusing of the components. The biggest hurdle is the binary compatibility issue of C++ based components, which is whether or not the components in the binary code compiled by a type of compiler are executing together on the same operating system with other components compiled by its old version compiler or other types of compilers. In practice, the number of components implemented with programming languages such as Java and Python has been increasing because those languages do not cause the binary compatibility problems. However, C++ is an important programming language needed for the control of automation machines/devices such as robots and SW-based PLCs because it provides fast performance [7, 8]. In addition, there have been lots of C++ based components or modules

developed and stably used till now for industrial/office/home automation. Because the components were compiled by different types and/or versions of compilers, it is necessary to reuse them effectively. Therefore, the binary compatibility of C++ components (or objects) should be resolved. Note that examples of the components (or objects) are classes, variables, and methods, where a class can include variables, methods, and zero or more classes.

There are following two types of methods in dynamical sharing of C++ classes: the C-based dynamic library method and the sharing method supported by a compiler. Because the C-based dynamic library method cannot directly share classes, the executable files such as .exe, .dll, and .so refer to the abstract class of the same header files and deliver the address of the instance of the class. The sharing method by compiler can directly share classes. But the method cannot share instances of classes compiled by different types of compilers [9]. In other words, there is a serious problem that the sharing of classes is applicable only to the same compiler, which makes spreading of C++ based components difficult.

For instance, let us consider two types of components in the Windows environment, which are made using Visual C++ from Microsoft (MSVC) and GCC from GNU, respectively. A MSVC-based component and a GCC-based component are not mutually compatible in most cases even though they have been made in the same Windows environment. This situation occurs due to the binary compatibility problem. The binary compatibility problem generally causes the situations where the methods of the object cannot be suitably called or its operation is not properly executed. And then the system can enter into a down (or dead) state. To solve the problem, it is necessary to design an object structure compatible in all types of compilers [10].

There have been some researches to solve the binary compatibility problem of C++ objects, examples of which are COM [11], CCC [12], and ZL [13]. COM solves the binary compatibility problem of C++ objects but has some limitations that it should operate in a Windows environment and be supported only by the MSVC compiler [11]. CCC is a library that the classes are composed of only header files to maintain the binary compatibility and designed to use the C++ 11 features [14] to make binary compatible objects with ease. But CCC has a limitation that it can be used only by compilers that support the C++ 11 standard. In other words, CCC does not share objects compiled by compilers not supporting the C++ 11 standard. In the C++ compatible language ZL [13], the binary compatible classes are supported by customized preprocessors and macros. It does not however support multiple inheritances and provides the binary compatibility only for GCC because the compiler for ZL is made from modified version of GCC. In addition COM and CCC do not support the method overloading, but ZL partially supports it. CCC supports exception handling, COM does it in the restricted manner, but ZL does not support it.

This chapter proposes the binary compatibility object model for C++ (BiCOMC) for reusability of software components which provide binary compatibility for sharing objects between C++ executable files in the Windows or the Linux environment. In addition, the proposed model provides the method overloading and overriding, multiple inheritance, and exception handling. And BiCOMC makes each other share the objects generated by different types of compilers such as MSVC, GCC, and ICC. This chapter provides macros of C++ preprocessor for sharing the binary compatible objects easily and independently of the types and versions of compilers. This chapter illustrates how to use the proposed model via a simple example in the Windows and Linux environment. To validate the proposed method, BiCOMC is compared with COM and CCC in terms of the call time during execution and the binary compatibility among interface versions and the types of compilers. Moreover, it is shown that BiCOMC-based components made using both MSVC and GCC can call the methods of each other and the interface version problems are resolved.

In the next section, a binary compatibility object model for C++ (BiCOMC) is proposed, which has the structures of virtual function tables including multiple inheritance and the casting algorithm for conversion of interfaces. It is shown that method overloading and multiple inheritances are supported. In Section 3, the component based on BiCOMC is defined, and its implementation is shown using examples. Section 4 suggests two examples. One is a simple example to illustrate how to use the proposed method in Windows and Linux environment. The other is an example of a robot application to validate the proposed method, where the application consists of three components compiled by different types of compilers. In Section 5, the binary compatibility and the performance measure of the call time of methods are evaluated. Finally, the conclusions are given in Section 6.

2. Binary compatibility object model for C++ (BiCOMC)

2.1 Object model

A BiCOMC is shown in **Figure 1** and has the interface Object as the root of the hierarchy of the class diagrams. Since interfaces are public and can be used by many objects, they do not have any member variables so that binary compatibility is maintained. Note that the interface Object has one pointer variable as shown in **Figure 1** for sharing of a virtual function table. The point variable is the vftptr pointer for accessing of the table and exists at the topmost of the interface. The structure of the virtual function table is explained later.

Let us consider the case where an executable file A can create an object but a different executable file B can delete the object. In this case, the memory allocated by the file A cannot normally be deleted using the C++ delete operator in the file B because the delete operator of the file B cannot invoke the destructor of the class in

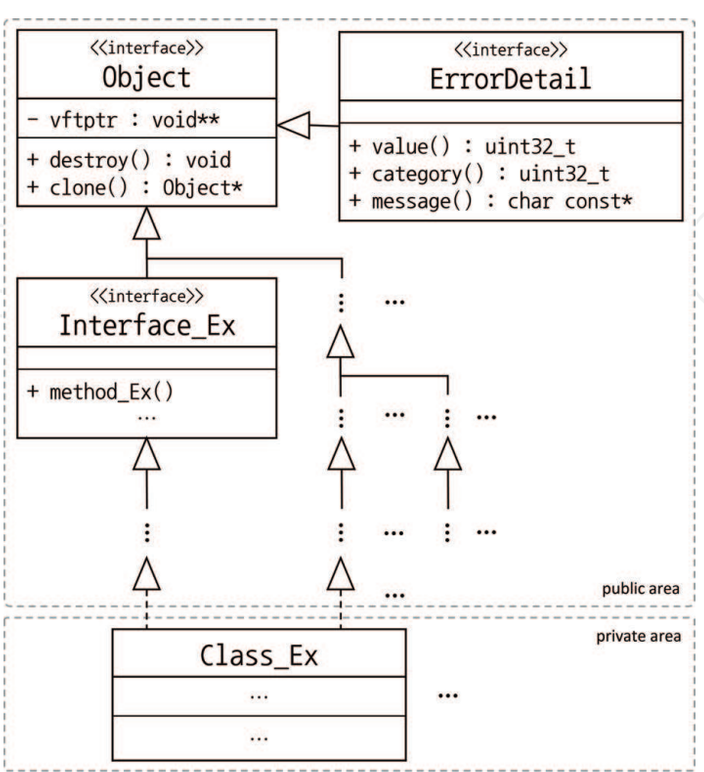

Figure 1.
Class diagram of Object and ErrorDetail interface for BiCOMC.

the file A. To solve this problem, the interface Object has a destroy() method so that objects can be deleted by the executable file which created the objects.

Because objects that are shared externally are exposed in the form of an interface, they cannot know the prototype of the object and then cannot be replicated through the C++ copy constructor. The interface Object has a clone() method that is allowed to clone the object.

When interface methods are implemented, the order of the methods in the virtual function table should be the same as the declaration order of the methods in the interface. So the order of the methods in the interface should not be changed once the interface is open in public. Adding a new method to the interface is allowed if its order is not changed. In other words the insertion of a new method is allowed only as the last method of the interface.

All methods can cause exceptions which are related to the interface ErrorDetail in **Figure 1**. Therefore, only objects implemented in the interface ErrorDetail should be thrown at the occurrence of the exceptions. The interface ErrorDetail has a value() method, a category() method, and a message() method that return an error value, an error category, and a description of the error, respectively.

An interface can inherit only one parent interface, but a class can inherit multiple interfaces. When a class has inherited multiple interfaces, the class has multiple vftptr pointers, which are the addresses of virtual function tables for individually inherited interfaces. The class, which is inherited multiple interfaces, refers to as many virtual function tables as the number of the inherited interfaces, which is shown in **Figure 3**.

2.2 Structure of a virtual function table

The vftptr pointer of the interface Object points to a virtual function table. A virtual function table contains the address of the overridden method and the interface information for the interface. **Figure 2** shows the structure of the virtual

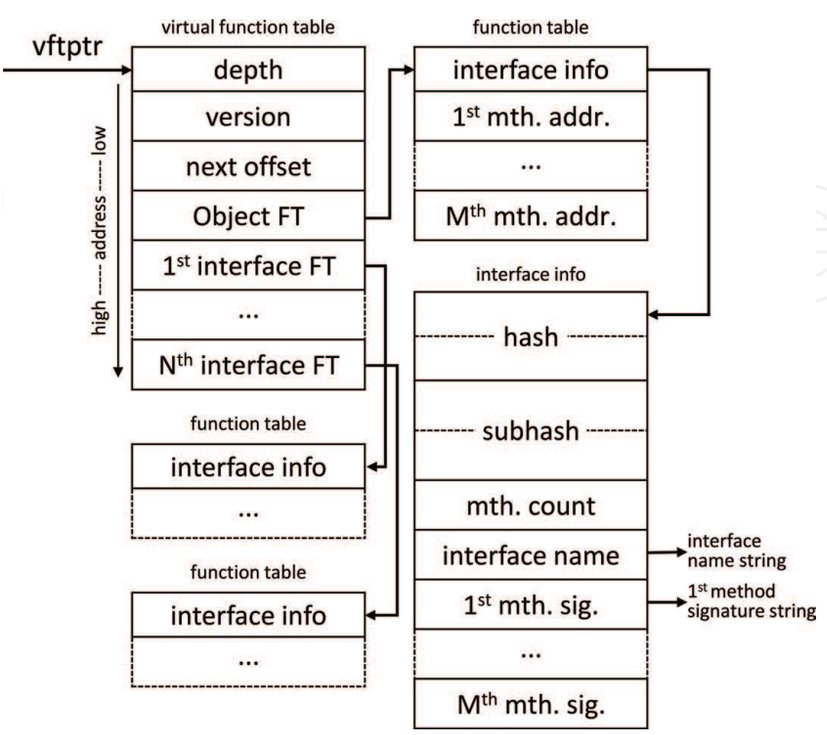

Figure 2.
Structure of a virtual function table.

function table. The sizes of depth, version, and next offset are equal to the size of a void pointer (or void*), respectively.

A virtual function table contains all the parent interfaces inherited by the interface. The first element, depth, of a virtual function table in **Figure 2** stores the number of inheritances from the interface Object to the last interface. If the interface directly inherited by the implementation class is the interface Object, the value of depth is 0. The second element, version, means the version number of BiCOMC for future extension of BiCOMC. The third element, next offset, is used for the objects that inherit multiple interfaces and represents the offset between the current vftptr and the next vftptr, which is explained later. The fourth element Object FT refers to the address of the function table of the interface Object where the actual addresses of overridden methods and the interface information are stored. The remaining elements i-th interface FT points to the function table of the i-th interface (i = 1... N).

The entry interface info in the function table means the interface information. The entry j-th mth. addr. (j = 1... M) is the address of the j-th method. Note that the addresses of the methods are stored in the same order as the order of declaration of the methods.

The interface info includes some elements such as hash, subhash, mth. count, interface name, and j-th mth. sig. (j = 1... M). The hash is a 64bit value, and it is calculated with the name of the interface and the names of inherited interfaces. The same hash means that the name of the interface and the names of the inherited interfaces are the same. The subhash is also a 64bit value, and it is calculated with M method signatures which contain the name of the method, a return type, and parameters' types. The same subhash means that the method definitions of two interfaces are the same. The sizes of the hash and the subhash of the interface may not be the same as the size of void*. Their sizes are calculated in (1) in bytes. If the hash values of two objects are different, the objects are incompatible. The subhash values are used for the method overloading and backward compatibility. So the method signatures should be checked if the subhash values are not matched. Note that the interface name and the j-th mth. sig. (j = 1...M) are null-terminated character strings encoded by UTF-8 and j-th mth. sig. consists of the method name, the return type, and types of parameters:

$$\text{size} = \lceil 16/\text{sizeof}(\text{void}^*) \rceil \times \text{sizeof}(\text{void}^*) \tag{1}$$

As mentioned above, the method signature can distinguish other methods with the same name because the method's signature is based on the method name as well as the types of parameters and the return type. So it can be said that BiCOMC supports the method overloading.

2.3 Structure of virtual function tables in multiple inheritance

In BiCOMC, an object that inherits multiple interfaces has as many vftptr pointers as the number of inherited interfaces. Note that the basic structure of the virtual function table is described in Section 2.2. **Figure 3** shows the structure of the virtual function tables of an object inheriting three interfaces.

In the case where three interfaces have been inherited, three vftptr pointers exist as shown in **Figure 3**. Assume that a 32bit system is used and the next offset of the virtual function **Table 1** is 4, which is the difference between the address of vftptr 2 and the address of vftptr 1 in the instance of class. And the next offset of the virtual function **Table 3** is −8, which is the difference between the address of vftptr 1 and the address of vftptr 3.

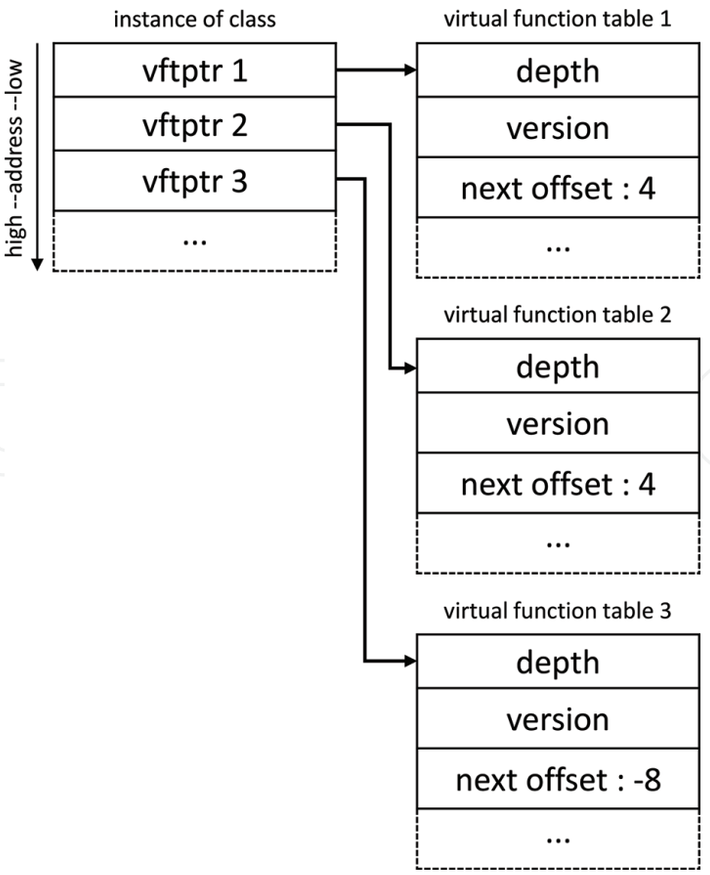

Figure 3.
Structure of virtual function table in the case of multiple inheritance.

BiCOMC			Dynamic library (DLL)					
			MSVC		GCC		ICC	
			9	14	4.5	5.2	14	16
Executable (EXE)	MS VC	9	O	O	O	O	O	O
		14	O	O	O	O	O	O
	GCC	4.5	O	O	O	O	O	O
		5.2	O	O	O	O	O	O
	ICC	14	O	O	O	O	O	O
		16	O	O	O	O	O	O

Table 1.
Tests of the binary compatibility in BiCOMC.

2.4 Interface casting

General casting methods of C++ such as static_cast and dynamic_cast cannot be used among objects created by different types of compilers. Therefore, methods that can cast BiCOMC objects are necessary regardless of a type of compilers used. **Figure 4** shows an algorithm for casting BiCOMC objects to other interfaces.

The casting algorithm gets virtual function tables using two parameters of obj and tgtTable. The algorithm compares the tables of obj with the tables of the target interface in order to check whether or not two interfaces are compatible. NULL is returned if two interfaces are not compatible each other. The seventh, tenth, and

```
1:  function CAST(obj, tgtTable)          ▷ obj is a instance of interface Object.
2:      cntIdx ← ⌈16/sizeof(void*)⌉
3:      p ← obj
4:      repeat
5:          table ← p.vftptr
6:          if tgtTable[1] ≠ table[1] then                    ▷ version mismatch
7:              p ← p + table[2]                          ▷ move to the next interface
8:              continue
9:          else if tgtTable[0] > table[0] then  ▷ incompatible count of methods
10:             p ← p + table[2]                          ▷ move to the next interface
11:             continue
12:         end if
13:         isCompatible ← True
14:         for i ← 0, tgtTable[0] do
15:             tInfo ← tgtTable[i + 3][0]
16:             info ← table[i + 3][0]
17:             if tInfo[cntIdx] > info[cntIdx] then
18:                 isCompatible ← False
19:                 break
20:             end if
21:             if tInfo.hash ≠ info.hash then                ▷ hash mismatch
22:                 isCompatible ← False
23:                 break
24:             end if
25:             if tInfo.subhash ≠ info.subhash then      ▷ subhash mismatch
26:                 for j ← 0, tInfo[cntIdx] do
27:                     tSig ← tInfo[cntIdx + j + 1]
28:                     sig ← info[cntIdx + j + 1]
29:                     if tSig ≠ sig then
30:                         isCompatible ← False
31:                         break
32:                     end if
33:                 end for
34:                 if isCompatible = False then
35:                     break
36:                 end if
37:             end if
38:         end for
39:         if isCompatible = True then
40:             return p
41:         end if
42:         p ← p + table[2]                          ▷ move to the next interface
43:     until p ≠ obj
44:     return NULL
45: end function
```

Figure 4.
Algorithm for interface casting.

forty-second lines in **Figure 4** are for processing of multiple inheritance. Lines 21–24 check hash values to test the names of interfaces and inheritance relationships. That is, lines 21–24 compare the hash values and detect whether they are compatible by detecting whether a new interface is added or a different interface name exists. Lines 25–37 compare subhash values, and two interfaces are considered as compatible interfaces if the values are the same. If they are different, it examines method signatures of interface information to check compatibility. An example of interface casting is shown in **Figure 6**.

3. Definition and implementation of component based on BiCOMC model

In this section the component based on BiCOMC is defined, and its implementation is shown using examples. **Figure 5** shows an example of the class Comp_1 based on BiCOMC, where the interface Interface_1 and the interface Interface_2 inherit the interface Object and Interface_3 inherits Interface_2. The definition of the interfaces in **Figure 5** is shown in **Figure 7**. The class Comp_1 is one of the components that provide interfaces Interface_1 and Interface_3. Interface Interface_1 consists of void mth_1(int) and int. mth_2(), and interface Interface_3 inherits interface Interface_2 and consists of void mth_1() and int. mth_2(int).

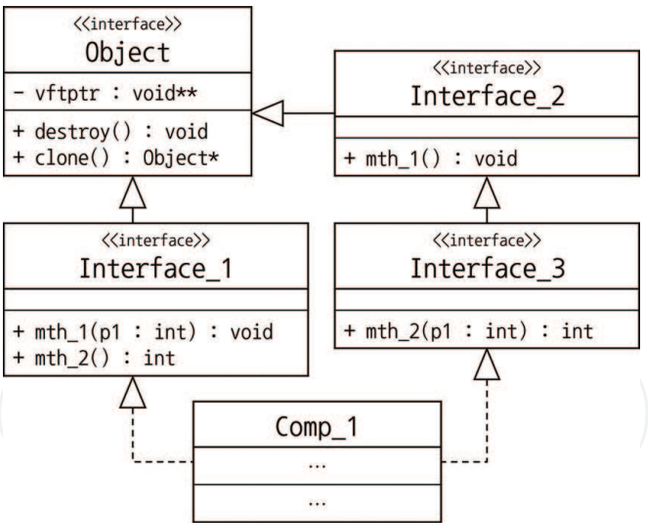

Figure 5.
Class diagram example of relationship between component and interface.

Figure 6 shows an example of the interface casting algorithm in **Figure 4**, which shows that Interface_1 is converted to Interface_2 using bicomc_cast based on the algorithm in **Figure 4**.

The macro BICOMC_INTERFACE defines interfaces as shown in **Figure 7**, using one or two parameters. The first parameter is the name of the interface, and the second parameter is the name of the parent interface and optional. In the case where the second parameter is empty, the interface Object is inherited. The macro BICOMC_DECL_METHOD in **Figure 7** declares the method of the interface. The macro has three parameters as follows: the first parameter is the name of the method, the second parameter is the function type of the method, and the third parameter is the number of parameters of the method. For example, let us consider BICOMC_DECL_METHOD(mth_2, int.(), 0). This macro represents the method int. mth_2() in **Figure 8**.

The interface definition in **Figure 7** is converted into the C++ code of **Figure 8** by C++ preprocessor. Note that C++ code for Interface_2 is not represented. The macro BICOMC_INTERFACE in **Figure 7** is converted to the C++ code related to the inputted interface name such as Interface_1. As seen in the 1st and 17th lines of **Figure 8**, the macro BICOMC_INTERFACE (Interface_1) and BICOMC_INTERFACE (Interface_3, Interface_2) create the codes of class Interface_1 public Object and class Interface_3 public Interface_2, respectively. The macro BICOMC_DECL_METHOD is converted to a method which consists of a name, a return type, and parameters. The second, third, and eleventh lines in **Figure 7** are converted into the third–seventh lines, ninth–fourteenth lines, and nineteenth to twenty-fourth lines in **Figure 8**, respectively. From the virtual function table pointed at by the vftptr pointer of the interface Object, the addresses of functions(or methods) are acquired using the inheritance depth of the interface and the order of declarations of the interface methods as shown in the 5th, 11th, and 21st lines in **Figure 8**, and then the actual overridden methods are called. These methods are converted into the function type as shown in **Figure 9** and then stored. Note that ErrorCode is a wrapper class of ErrorDetail in **Figure 1**.

```
1: Interface_1* p1 = new Comp1();
2: Interface_2* p2 = bicomc_cast<Interface_2*>(p1);
```

Figure 6.
An example of interface casting.

```
1: BICOMC_INTERFACE(Interface_1)
2:   BICOMC_DECL_METHOD(mth_1, void(int), 1)
3:   BICOMC_DECL_METHOD(mth_2, int(), 0)
4: BICOMC_INTERFACE_END(Interface_1)
5:
6: BICOMC_INTERFACE(Interface_2)
7:   BICOMC_DECL_METHOD(mth_1, void(), 0)
8: BICOMC_INTERFACE_END(Interface_2)
9:
10: BICOMC_INTERFACE(Interface_3, Interface_2)
11:   BICOMC_DECL_METHOD(mth_2, int(int), 1)
12: BICOMC_INTERFACE_END(Interface_3)
```

Figure 7.
Definition of interface using macro in **Figure 5**.

```
1: class Interface_1 : public Object {
2:   …
3:   void mth_1(int p1) {
4:     void* r;
5:     if (ErrorDetail* e = vftptr[4][1](this, &r, p1))
6:       throw e;
7:   }
8:   …
9:   int mth_2() {
10:    int r;
11:    if (ErrorDetail* e = vftptr[4][2](this, &r))
12:      throw e;
13:    return r;
14:  }
15: };
16: …
17: class Interface_3 : public Interface_2 {
18:   …
19:   int mth_2(int p1) {
20:    int r;
21:    if (ErrorDetail* e = vftptr[5][1](this, &r, p1))
22:      throw e;
23:    return r;
24:  }
25: };
```

Figure 8.
C++ code of **Figure 7** *after preprocessing.*

In **Figure 9**, parameter I* is the address of the interface instance, the second parameter R* is the address of the variable receiving the return value of the method, and other parameters (P_1…P_N) are parameters of the method. The method stored in the function type is called with the same function-calling convention. Note that exception information is returned using the interface ErrorDetail. When the exception information has been returned, the C++ code of the method throws the exception on behalf of the method as shown in the 6th, 12th, and 22nd lines in **Figure 8**.

Figure 10 shows the implementation of the interfaces in **Figure 7**. The interfaces in **Figure 7** are inherited, and the methods are overridden in order to define the class Comp_1. The interfaces and methods for overriding are set using the macros BICOMC_OVERRIDE and BICOMC_OVER_METHOD. Parameters of the macro

```
ErrorDetail* (*) (I*, R*, P_1, P_2, …, P_N)
```

Figure 9.
Function type stored in the virtual function table.

```
1: class Comp_1 : public Interface_1, public Interface_3{
2:   BICOMC_OVERRIDE(Interface_1, Interface_3)
3:     BICOMC_OVER_METHOD(destroy, void())
4:     BICOMC_OVER_METHOD(mth_1, void())
5:     BICOMC_OVER_METHOD(mth_1, void(int))
6:     BICOMC_OVER_METHOD(mth_2, int())
7:     BICOMC_OVER_METHOD(mth_2, int(int))
8:   BICOMC_OVERRIDE_END()
9: public:
10:   Comp_1() : BICOMC_OVERRIDE_INIT() { ··· }
11:   void destroy() { delete this; }
12:   void mth_1() { ··· }
13:   void mth_1(int p1) { ··· }
14:   int mth_2() { ··· }
15:   int mth_2(int p1) { ··· }
16: };
```

Figure 10.
Implementation of interfaces and overriding.

```
1: class Comp_1 : public Interface_1, public Interface_3{
2:   ···
3:   bool overrideMethod() {
4:     ···
5:     typedef Comp_1 C;
6:     Interface_1::vftptr[3][1]
                = &Method<void(C::*)(), &destroy>::call;
7:     Interface_3::vftptr[3][1]
                = &Method<void(C::*)(), &destroy>::call;
8:     Interface_1::vftptr[4][1]
                = &Method<void(C::*)(int), &mth_1>::call;
9:     Interface_3::vftptr[4][1]
                = &Method<void(C::*)(), &mth_1>::call;
10:    Interface_1::vftptr[4][2]
                = &Method<int(C::*)(), &mth_2>::call;
11:    Interface_3::vftptr[5][1]
                = &Method<void(C::*)(int), &mth_2>::call;
12:    ···
13:  }
14:  bool const holder;
15: public:
16:  Comp_1() : holder(overrideMethod()) { ··· }
17:  ···
18: };
```

Figure 11.
C++ code of **Figure 10** *after preprocessing.*

BICOMC_OVERRIDE are names of all interfaces that the component inherits. Parameters of the macro BICOMC_OVER_METHOD are the names and signatures of the overridden methods. **Figure 11** shows C++ code converted from **Figure 10** by C++ preprocessor. The macro BICOMC_OVERRIDE is converted to bool overrideMethod(), and parameters of the macro such as Interface_1 and Interface_3 are used for clear conversion of overridden methods related to the type of the interface. The BICOMC_OVER_METHOD macros on lines 3–7 in **Figure 10** are converted to codes on lines 6–11 in **Figure 11** and the macro BiCOMC_OVERRIDE_INIT() to holder(overrideMethod()) on the 16th line in **Figure 11**.

4. Examples of BiCOMC-based application

4.1 A simple example

This section describes in sequence how to make a file copy application, which is a simple example based on BiCOMC. The file copy application is designed to run on Windows and Linux and is built using MSVC and GCC, respectively. The dynamic library providing the file copy function, which named utility, is created, and the executable file using this dynamic library, which named app, is generated as the application. The interface is first defined to share an object that provides a method for

copying files. The interface to provide this functionality is defined in the utility.h file as the ICopy interface. **Figure 12** shows that the ICopy interface is defined based on the macro of BiCOMC and is the source code of the dynamic library named *utility.cpp*.

The BiCOMC macro is defined in *object.h*, which is added as shown on line 5 in **Figure 12** to use it. *ICopy* has a *copy()* method that receives two file names of *src* and *dst* as the input parameters and copies *src* to *dst*, which is shown on the eighth line in **Figure 12**.

In order to implement the *ICopy* interface in **Figure 12**, the *Copy* class is written made as shown in the *utility.cpp* of **Figure 13**. In addition, *copy()* method of *ICopy* is overridden in **Figure 13**. Note that the *utility.cpp* file is the source code for the *utility* dynamic library.

The *utility.h* which defines the *ICopy* interface is added as one of the header files on the second line in **Figure 13**, and some header files for the *Copy* class implementation on lines 4–13 are defined for Windows and Linux. The *Copy* class inherits the *ICopy* interface on the 15th line, and the overriding methods are explicitly specified using the BiCOMC macro on lines 16–19, where the specified methods are written on lines 23–37. The *create()* function on lines 40–46 creates an instance of the *Copy* class in order to pass its instance outside of the dynamic library called *utility*. The *main()* function of *app* using *utility* dynamic library is written in the *app.cpp* file, which is illustrated in **Figure 14**.

The *create()* function is called on the 21th line in **Figure 14** to get an instance of the *Copy* class in **Figure 13** but casted to *ICopy** using *bicomc_cast()* since *create()* returns *Object**. The *copy()* method is called to perform file copy on the 24th line. This method is used in try-catch because it generates an exception when *copy()* fails. The commands shown in **Figure 15** are executed to build *utility.cpp* into *utility* dynamic library and *app.cpp* into *app* executable file, respectively.

4.2 BiCOMC-based component for robot application

This section explains how to implement BiCOMC-based components using a robot applications. The robot application consists of three components in a Windows environment as follows: the *ManipulatorComp*, the *MobileComp*, and the *AppComp* in **Figure 16**.

The component *ManipulatorComp* is compiled with GCC 5.2 and controls the manipulator. The component *MobileComp* is compiled with MSVC 14 and controls the mobile platform. The component *AppComp* is compiled with MSVC 10 and coordinates the components *MobileComp* and *ManipulatorComp*. The component *AppComp* accesses the components *MobileComp* and *ManipulatorComp* through the interfaces *IMobile* and *IManipulator*, respectively. Definitions of these interfaces are illustrated in **Figure 17**. **Figure 18** shows the definition of the component *MobileComp* that inherits the interface *IMobile* and overrides the methods of the

```
1: // utility.h
2: #ifndef UTILITY_H__
3: #define UTILITY_H__
4:
5: #include <bicomc/object.h>
6:
7: BICOMC_INTERFACE(ICopy)
8:   BICOMC_DECL_METHOD(copy, void(char const* src, char const* dst), 2)
9: BICOMC_INTERFACE_END(ICopy)
10:
11: #endif // !def UTILITY_H__
```

Figure 12.
C++ code of utility.h.

```cpp
// utility.cpp
#include "utility.h"

#ifdef _WIN32
# include <Windows.h>
# define DL_EXPORT __declspec(dllexport)
#else
# include <sys/sendfile.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
# define DL_EXPORT __attribute__((visibility("default")))
#endif

class Copy : public ICopy {
  BICOMC_OVERRIDE(ICopy)
    BICOMC_OVER_METHOD(destroy, void())
    BICOMC_OVER_METHOD(copy, void(char const*, char const*))
  BICOMC_OVERRIDE_END()

public:
  Copy() : BICOMC_OVERRIDE_INIT() {}
  void destroy() { delete this; }
  void copy(char const* src, char const* dst) {
#ifdef _WIN32
    if (CopyFile(src, dst, TRUE) != 0)
      throw std::runtime_error("copy fails");
#else
    int srcFd = open(src, O_RDONLY);
    int dstFd = open(dst, O_WRONLY | O_CREAT);
    struct stat st; fstat(srcFd, &st);
    int ret = sendfile(dstFd, srcFd, NULL, st.st_size);
    close(srcFd); close(dstFd);
    if (ret == -1)
      throw std::runtime_error("copy fails");
#endif
  }
};

extern "C" DL_EXPORT Object* create() {
  try {
    return new Copy();
  } catch (...) {
    return NULL;
  }
}
```

Figure 13.
C++ code of utility.h.

```cpp
// app.cpp
#ifdef _WIN32
# include <Windows.h>
#else
# include <dlfcn.h>
#endif
#include <iostream>
#include "utility.h"

int main() {
#ifdef _WIN32
  HMODULE handle = LoadLibraryA("utility.dll");
  void* raw = GetProcAddress(handle, "create");
#else
  void* handle = dlopen("./libutility.so", RTLD_LAZY);
  void* raw = dlsym(handle, "create");
#endif

  typedef Object*(*Func)();
  Func create = reinterpret_cast<Func>(raw);
  ICopy* pCopy = bicomc_cast<ICopy*>(create());
  if (pCopy) {
    try {
      pCopy->copy("src.txt", "dst.txt");
    } catch (ErrorCode const& e) {
      std::cerr << e.message() << std::endl;
    }
    pCopy->destroy();
  }

#ifdef _WIN32
  FreeLibrary(handle);
#else
  dlclose(handle);
#endif
  return 0;
}
```

Figure 14.
C++ code of app.cpp.

MSVC & Windows	`cl /Fe utility utility.cpp /LD` `cl /Fe app app.cpp`
GCC & Linux	`g++ -o libutility.so utility.cpp -shared -fPIC` `g++ -o app app.cpp`

Figure 15.
Compilation commands of app.cpp and utility.cpp.

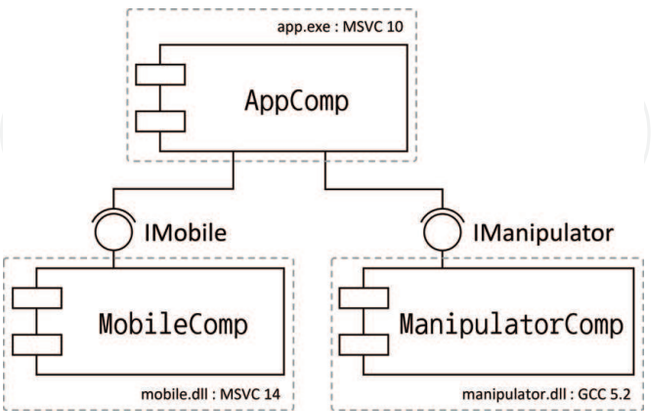

Figure 16.
Configuration example of components for a robot application.

```
1: BICOMC_INTERFACE(IMobile)
2:   BICOMC_DECL_METHOD(stop, void(int), 1)
3:   BICOMC_DECL_METHOD(move, void(double,double), 2)
4:   …
5: BICOMC_INTERFACE_END(IMobile)
6:
7: BICOMC_INTERFACE(IManipulator)
8:   BICOMC_DECL_METHOD(stop, void(int), 1)
9:   BICOMC_DECL_METHOD(move, void(double*,double*,int),3)
10:  …
11: BICOMC_INTERFACE_END(IManipulator)
```

Figure 17.
Definition of interfaces IMobile and IManipulator.

```
1: class MobileComp : public IMobile {
2:   BICOMC_OVERRIDE(IMobile)
3:     BICOMC_OVER_METHOD(stop, void(int))
4:     BICOMC_OVER_METHOD(move, void(double,double))
5:     …
6:   BICOMC_OVERRIDE_END()
7: public:
8:   MobileComp() : BICOMC_OVERRIDE_INIT() { … }
9:   void stop(int mode) { … }
10:  void move(double xPos, double yPos) { … }
11:  …
12: };
```

Figure 18.
Definition of component MobileComp.

interface. The component *ManipulatorComp* and the component *AppComp* can be defined in the similar manner, which is shown in **Figures 19** and **20**, respectively.

Figure 21 shows the operation results of the robot after three components in **Figure 16** are successfully implemented, which are parts captured from a video clip [15]. It can be observed from **Figure 21** that the BiCOMC-based components function properly regardless of the types of compilers.

```
1: class AppComp {
2:   IMobile* pMobile;
3:   IManipulator* pManipulator;
4:   …
5:   void main() {
6:     …
7:     pMobile->move(0.5, 0.0);
8:     …
9:     pManipulator->stop(0);
10:  }
11: };
```

Figure 19.
Example of component AppComp.

```
1: class ManipulatorComp : public IManipulator {
2:   BICOMC_OVERRIDE(IManipulator)
3:     BICOMC_OVER_METHOD(stop, void(int))
4:     BICOMC_OVER_METHOD(move, void(double*,double*,int))
5:     …
6:   BICOMC_OVERRIDE_END()
7: public:
8:   ManipulatorComp() : BICOMC_OVERRIDE_INIT() { … }
9:   void stop(int mode) { … }
10:  void move(double* angles, double* times, int n) {…}
11:  …
12: };
```

Figure 20.
Definition of component ManipulatorComp.

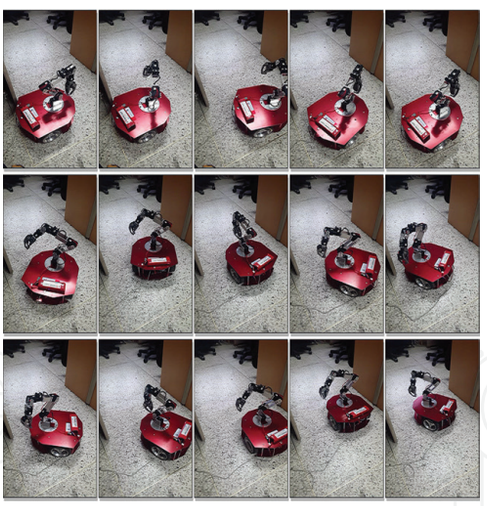

Figure 21.
Video capture of robot application running.

5. Evaluation

This section evaluates the binary compatibility occurred among different types of compilers and verifies whether backward compatibility can be maintained when the interface version has been changed. There are some binary compatibility checking tools like ABI Compliance Checker [16], shlib-compat [17], libabigail [18], and ABICheck [19], but these tools are used to check API/ABI in C language level or to compare virtual function tables generated by the supported compiler. So they cannot check the binary compatibility of C++ objects created by different types of compilers. In addition these tools cannot detect the compatibility

maintained by BiCOMC. For checking of the binary compatibility, this chapter uses the proposed methods explained in Sections 5.1 and 5.2. Finally the call times as performance measures are compared among BiCOMC, COM, and CCC using different types of compilers.

5.1 Evaluation of binary compatibility between compilers

Compilers generally reorder a virtual function table according to each compiler's ABI, which makes the binary compatibility difficult. The interface *CompatibilityChecker* is suggested to check whether the methods of objects shared by different binaries (or executable files) created by different types of compilers normally call each other. The interface *CompatibilityChecker* has the methods *check1()*, *check2()*, and *check3()* arranged in an unordered fashion and is illustrated in **Figure 22**. The methods simply return the values 1, 2, and 3, respectively. In addition, all methods' calling convention is controlled as the same.

These methods are called to check whether the normal return value is delivered. The tests are considered successful if the three methods are normally called and are judged to have failed if any of the methods are not normally called or the program has shut down. The experiments using the interface *CompatibilityChecker* are performed using MSVC 9 and 14, GCC 4.5 and 5.2, and the Intel C++ Compiler 14 and 16 in Windows 10 in order to verify binary compatibility among different types of compilers. Note that the GCC of MinGW-w64 is used. In **Tables 1–3**, "O" means pass, "X" means failure, and "-" means not testable. ICC is the abbreviation of the Intel C++ Compiler. In other words, "O" means that the three methods are normally called.

It can be seen in **Tables 1** and **2** that BiCOMC and CCC guarantee the binary compatibility among MSVC, GCC, and ICC. However, CCC is only available in compilers supporting C++ 11. **Table 3** shows that the binary compatibility between MSVC and ICC is guaranteed as stated in [20], but these two compilers are not compatible with GCC. As stated earlier, the different types of compilers reorder the virtual function table, and then the reordered methods are not called normally. So C++ methods are called abnormally by different types of compilers as shown in **Table 3**, whereas BiCOMC and CCC worked normally because they prevent to reorder a virtual function table.

5.2 Evaluation of backward compatibility of different interface versions

In developing the program, it is necessary to add a new method to the existing interface. That is, the interface version is changed in this case, which is shown in **Figure 23**.

Figure 23 shows two interfaces of *IfaceA* and *IfaceB*, where *IfaceB* inherits *IfaceA*. In the version *v1*, *IfaceA* has a method, *mth_1()*. But the interface *IfaceA* should be upgraded to the new version *v2* because a new method of *mth_3()* is added. Let us consider the following: the v1-related interfaces and the v2-related interfaces are used in the caller and the callee programs, respectively.

```
1: struct CompatibilityChecker {
2:   virtual int check2(int n1, int n2) = 0;
3:   virtual int check1() = 0;
4:   virtual int check3(int n1) = 0;
5: };
```

Figure 22.
Interface for binary compatibility test.

CCC			Dynamic library (DLL)					
			MSVC		GCC		ICC	
			9	14	4.5	5.2	14	16
Executable (EXE)	MS VC	9	—	—	—	—	—	—
		14	—	O	—	O	—	O
	GCC	4.5	—	—	—	—	—	—
		5.2	—	O	—	O	—	O
	ICC	14	—	—	—	—	—	—
		16	—	O	—	O	—	O

Table 2.
Tests of the binary compatibility in CCC.

CCC			Dynamic library (DLL)					
			MSVC		GCC		ICC	
			9	14	4.5	5.2	14	16
Executable (EXE)	MS VC	9	O	O	X	X	O	O
		14	O	O	X	X	O	O
	GCC	4.5	X	X	O	O	X	X
		5.2	X	X	O	O	X	X
	ICC	14	O	O	X	X	O	O
		16	O	O	X	X	O	O

Table 3.
Tests of the binary compatibility in C++.

This test also uses simple methods that return values of 1, 2, and 3, respectively, which are similar to methods used in Section 5.1. The experiments are done using MSVC 14, GCC 5.2, and ICC 16 in Windows 10.

Table 4 shows that BiCOMC can enable the binary compatibility between interfaces of versions *v1* and *v2*, but CCC, COM, and C++ are not compatible between codes with different interface versions. Therefore, BiCOMC supports the backward compatibility of interface versions. MSVC, GCC, and ICC generate a virtual function table in contiguous memory space regardless of inheritance. In this structure of the table, if a new method is added in the parent interface, the offset of child's methods will be changed. For this reason, C++ does not provide the backward compatibility. COM and CCC also have the same reason. BiCOMC has the structure

```
1: struct IfaceA {
2:   virtual int mth_1() = 0;
3:
4: };
5: struct IfaceB : IfaceA {
6:   virtual int mth_2() = 0;
7: };
```

v1

```
1: struct IfaceA {
2:   virtual int mth_1() = 0;
3:   virtual int mth_3() = 0;
4: };
5: struct IfaceB : IfaceA {
6:   virtual int mth_2() = 0;
7: };
```

v2

Figure 23.
Interface for backward compatibility test per version.

Method compiler	BiCOMC	CCC	COM	C++
MSVC 14	O	X	X	X
GCC 5.2	O	X	—	X
ICC 16	O	X	X	X

Table 4.
Tests of the backward compatibility.

of **Figure 2** so that the parent's methods and the child's methods can be stored at independent memory space of each other. Thus BiCOMC supports the backward compatibility of interface versions.

5.3 Call time evaluation

Call times as performance measures are measured in Windows 10 using MSVC 14, GCC 5.2, and ICC 16. The methods of objects are called 10 million times, and the call times are obtained as the average values of all call times. These are shown in **Figures 24–26**, in which MSVC, GCC, and ICC compilers are used for evaluation, respectively. In these figures, two notations such as *XXX* and *XXX-R* are used, where *XXX* is one of BiCOMC, CCC, CPP (or C++), and COM and *-R* means that the method used in the test has a return value.

Tables 1–4 show that BiCOMC provides the best binary compatibility among different types of compilers. **Figures 24–26** show that C++ has the best call time, but the call time of BiCOMC is similar to those of C++ and COM. CPP is generally faster than others because the method calling accesses an optimized virtual function table which accesses directly addresses of methods. COM is also similar to CPP. Note that GCC does not support COM. CCC is slower as std::function, one of the C++ 11 features [14], is basically used [12]. BiCOMC is slower slightly than CPP and COM because it accesses addresses of methods in the *function table* pointed by the *virtual function table*. But the test results show no or little significant difference among BiCOMC and CPP/COM.

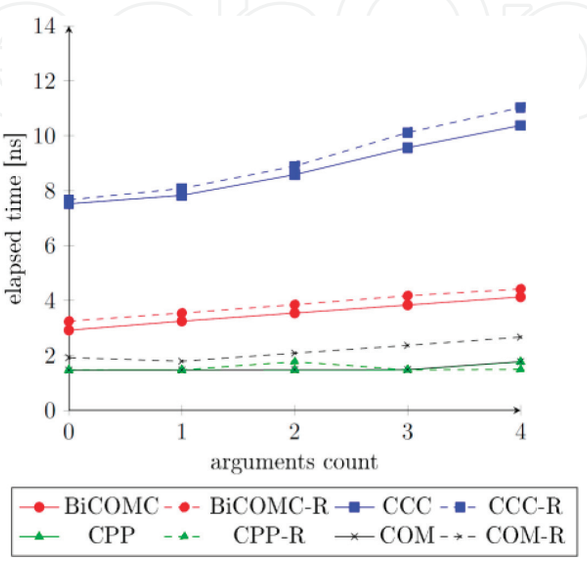

Figure 24.
Call time in MSVC.

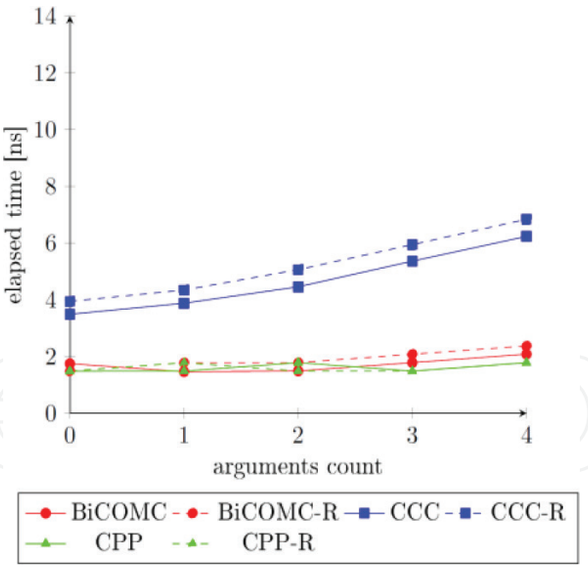

Figure 25.
Call time in GCC.

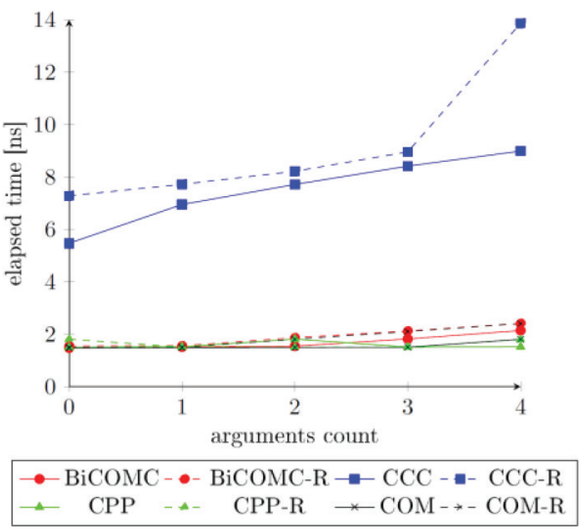

Figure 26.
Call time in ICC.

6. Conclusion

This chapter proposed the binary compatibility object model for C++ (BiCOMC) to provide the binary compatibility of objects necessary for reusability of software components in the Windows and Linux environment in order to share objects among C++ based executable files such as .exe, .dll, and .so. The interfaces for the component, method overloading and overriding, multiple inheritance, and the exception handling were suggested based on BiCOMC model.

The proposed model was validated by application examples and comparisons with commonly known object models such as C++, COM, and CCC in terms of the call time of a method during execution and the binary compatibility such as reusability. The application examples showed that components compiled by GCC and MSVC call each other without any restrictions. From **Tables 1–3**, it can be seen that the BiCOMC provides better binary compatibility in a Windows environment than object models in C++, COM, and CCC, which are compiled in GCC, MSVC, and ICC. The BiCOMC was compared with C++, COM, and CCC in terms of the call times of methods during run time. The results showed that the call time of

the BiCOMC was similar to C++/COM. In other words, the application examples and the evaluation results verified that the proposed method was provided for the binary compatibility among different types of compilers.

In future we will develop and distribute BiCOMC-based components for various applications such as industrial/medical robot applications and factory/home automation application, which can be used regardless of the types of compilers.

IntechOpen

IntechOpen

Author details

Donguk Yu and Hong Seong Park*
Department of Electrical and Electronics Engineering, Kangwon National University, Chuncheon, Republic of Korea

*Address all correspondence to: hspark@kangwon.ac.kr

IntechOpen

© 2018 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. CC BY

References

[1] Han S, Kim M, Park HS. Open software platform for robotic services. IEEE Transactions on Automation Science and Engineering;**9**(3):467-481

[2] Jang C et al. OPRoS: A new component-based robot software platform. ETRI Journal;**32**(5):646-656

[3] OPRoS Site [Online]. Available from: http://ropros.org [Accessed: 03 January 2018]

[4] Ando N et al. Software deployment infrastructure for component based RT-systems. Journal of Robotics and Mechatronics;**23**(3):350-359

[5] OpenRTM Site [Online]. Available from: http://www.openrtm.org [Accessed: 03 January 2018]

[6] OROCOS Site [Online]. Available from: http://www.orocos.org [Accessed: 04 January 2018]

[7] Gherardi L, Brugali D, Comotti D. A Java Vs. c++ performance evaluation: A 3d modeling benchmark. In: Proceedings of International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2012); 5-8 November 2012; Tsukuba. Berlin, Heidelberg: Springer; 2012. p. 161-172. DOI: 10.1007/978-3-642-34327-8

[8] Hildstrom G. Programming Language Performance Comparison [Online]. Available from: http://hildstrom.com/projects/langcomp/index.html [Accessed: 04 January 2018]

[9] Sayfan G. Building Your Own Plugin Framework [Online]. Available from: http://www.drdobbs.com/cpp/building-your-own-plugin-framework-part/204202899 [Accessed: 04 January 2018]

[10] Goldstein TC and Sloane AD. The object binary interface—C objects for evolvable shared class libraries. In: Proceedings of the 6th Conference on USENIX Sixth C++ Technical Conference (CTEC'94). Vol. 6; 11-14 April 1994; Cambridge, MA

[11] COM: Component Object Model Technologies [Online]. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680573(v=vs.85).aspx [Accessed: 04 January 2018]

[12] Bandela JR. Cross Compiler Call [Online]. Available from: https://github.com/jbandela/cross_compiler_call [Accessed: 04 January 2018]

[13] Atkinson K. ABI compatibility through a customizable language. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10); 10-13 October 2010; Eindhoven. The Netherlands. DOI: 10.1145/1868294.1868316

[14] ISO/IEC 14882:2011 Information Technology—Programming Languages—C++, ISO/IEC JTC 1/SC 22

[15] Yu D. Video of Robot Application Running [Online]. Available from: https://www.youtube.com/watch?v=gMMsNINp14g [Accessed: 04 January 2018]

[16] ABI Compliance Checker [Online]. Available from: https://lvc.github.io/abi-compliance-checker [Accessed: 05 January 2018]

[17] Kurtsou G. Shlib-Compat: ABI Compatibility Checker for Shared Libraries with Symbol Versioning [Online]. Available from: https://github.com/glk/shlib-compat [Accessed: 04 January 2018]

[18] The ABI Generic Analysis and Instrumentation Library [Online].

Available from: https://sourceware.org/libabigail [Accessed: 05 January 2018]

[19] Abicheck [Online]. Available from: http://abicheck.sourceforge.net [Accessed: 06 January 2018]

[20] Intel C++ Compiler Compatibility with Microsoft Visual C++ [Online]. Available from: https://software.intel.com/en-us/articles/intel-c-compiler-compatibility-with-microsoft-visual-c [Accessed: 06 January 2018]

IntechOpen