Testing, Reliability, and Interoperability Issues in the CORBA Programming Paradigm

Gang Xing
Computer Science and Engineering Department
The Chinese University of Hong Kong
Shatin, N. T.
Hong Kong
+852 26098412
gxing@cse.cuhk.edu.hk

Michael R. Lyu
Computer Science and Engineering Department
The Chinese University of Hong Kong
Shatin, N. T.
Hong Kong
+852 26098429
lyu@cse.cuhk.edu.hk

ABSTRACT

CORBA (Common Object Request Broker Architecture) is widely perceived as an emerging platform for distributed systems development. In this paper, we discuss CORBA's testing, reliability and interoperability issues among multiple program versions implemented by different languages (Java and C++) based on different vendor platforms (*Iona Orbix* and *Visibroker*). We engage 19 independent programming teams to develop a set of CORBA programs from the same requirement specifications, and measure the reliability of these programs. We design the required test cases and develop the operational profile of the programs. After running the test, we classify the detected faults and evaluate the reliability of the programs according to the operational profile. We also discuss how to test the CORBA programs based on their specification and Interface Design Language (IDL). We measure the interoperability of these programs by evaluating the difficulty in exchanging the clients and servers between these programs. The measurement we obtained indicates that without a good discipline on the development of CORBA objects, interoperability would be very poor, and reusability of either client or server programs is very doubtful. We further discuss particular incidences where these programs are not interoperable, and describe future required engineering steps to make these programs interoperable, testable, and reliable.

Keywords:

CORBA, Testing, Reliability, Interoperability, Defect Classification, Metrics.

1. INTRODUCTION

Common Object Request Broker Architecture (CORBA) [12] is a subject for wide study [2,10,13], and abundant CORBA information is available from various resources [4]. CORBA, the component standard of the Object Management Group (OMG), is generally perceived as an emerging platform for distributed systems development. Its purpose is to provide a platform-independent, language-independent, and vendor-independent component standard for distributed systems.

Although CORBA has gained significant attention recently, the testing, reliability, and interoperability issues for the CORBA programming paradigm remain largely unexplored today. It is the objective of this paper to address these issues using a real-life CORBA project. The organization of this paper is as follows. In Section 2, we introduce an experimental CORBA project and show the metrics of the resulting program versions. In Section 3 and Section 4, we discuss respectively, testing and reliability issues based on our project experience. In Section 5, we assess interoperability of the CORBA programs and present some special cases. Section 6 gives conclusions and future work of this paper.

2 EXPERIMENTAL PROJECT DESCRIPTION

2.1 General Information about the Project

In the fall of 1998 we engaged 19 programming teams to design, implement, test, and demonstrate a Soccer Team Management System using CORBA. The duration of the project was 4 weeks. The programming teams (2-3 students for each team) participating in this project were required to independently design and develop a distributed system, which allows multiple clients to access a Soccer Team Management Server for 10 different operations. The teams were free to choose different CORBA vendors (*Visibroker* or *Iona Orbix*), using different programming languages (*Java* or *C++*) for the client or server programs. These programs have to pass an acceptance test, when the programs were subjected to two test cases for each of the 10 operations: one for normal operation and the other for operation which would raise exceptions.

Among these 19 programs 12 chose to use *Visibroker*, while 7 chose to use Iona *Orbix*. For the 12 *Visibroker* programs, 9 uses Java for both client and server implementation, 2 uses C++ for both client and server implementation, and 1 uses

Team	Client	Server
P1, P2, P3, P7, P8, P10, P11, P12, P17	Visibroker/JAVA	Visibroker/JAVA
P6, P16	Visibroker/C++	Visibroker/C++
P9	Visibroker/JAVA	Visibroker /C++
P4, P5, P13, P14, P15, P18, P19	Iona Orbix/C++	Iona Orbix/C++

Table 1: ORBs and Languages Usag2.2 Program Metrics

Java as its client and C++ as its server. The detailed list is shown in Table 1.The software metrics of these 19 programs are listed in Table 2. The metrics were collected using *etags* and some *perl* scripts. These programs range from 500 to 5000 lines of code (LOC). The large size of program P12 was due to fancy user interface and on-line help commands. The distribution of the client code versus server code is 1.79.

Team	Total LOC*	Client LOC	Server LOC	Client Class	Client Method	Server Class	Server Method
P1	512	182	330	3	5	13	20
P2	1129	613	516	3	15	5	26
P3	1874	1023	851	3	23	5	62
P4	1309	409	900	3	12	1	23
P5	2843	1344	1499	4	26	1	25
P6	1315	420	895	3	3	1	39
P7	2674	1827	847	3	17	5	35
P8	1520	734	786	3	24	4	30
P9	2121	1181	940	4	22	3	43
P10	1352	498	854	3	12	5	41
P11	563	190	373	3	12	3	20
P12	5695	4641	1054	14	166	5	32
P13	2602	1587	1015	3	27	3	32
P14	1994	873	1121	4	12	5	39
P15	714	348	366	4	11	4	33
P16	1676	925	751	3	3	23	44
P17	1288	933	355	6	25	5	35
P18	1731	814	917	3	12	3	20
P19	1900	930	970	3	3	2	20
Avg	1832.2	1024.8	807.4	3.42	4.21	21.74	32.58

Table 2: General Software Metrics

3. TESTING ISSUES FOR CORBA PROGRAMS

3.1 Test Preparation and Procedure

In order to evaluate the reliability of these CORBA programs, we apply test cases to the program versions and assess reliability based on the test results. We describe our testing procedure, interpret the result, and discuss some testing issues.

The test cases are mainly derived according to the requirement specifications. We first define a simple, normal test case for each operation. Then we define the test cases which may generate exceptions when applied to each operation. We define these test cases with the help of IDL. For example, a well-suited IDL definition for the *CreateTeam* operation in the Soccer Team Management System is:

```
Void CreateTeam(Name TeamName,PlayerList Players)
    raise (SameTeamName,BadTeamName,
    BadPlayerList, SamePlayerName,
    SamePlayerNumber, InvalidRole,
    NotEnoughPlayer,TooManyPlayer,
    NotEnoughGoalKeeper,NotEnoughCenterForward,
    NotEnoughLeftWing,NotEnoughLeftWing);
```

We can then define test case for each exception accordingly. These test cases are classified in the following types:

- Invalid parameter (BadTeamName, BadPlayerList)
- Teams conflict with rules (NotEnoughPlayer, NotEnoughGoalKeeper, etc.)
- Invalid player (SamePlayerName, InvalidRole)

We use the same method to define the test cases for other operations. The test case distribution is listed in Table 3. The test procedure is shown in Figure 1. In order to reduce the testing work for these program versions, we define a test sequence for each operation to cover all the test cases.

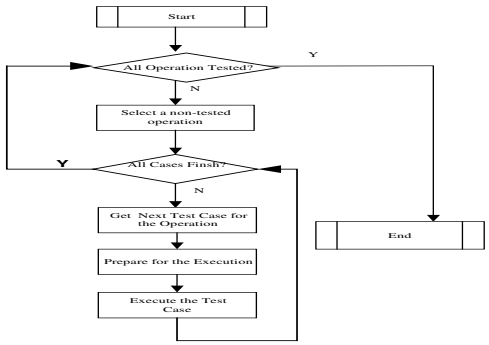

Figure 1: Test Procedure

3.2 Experiment Results and Interpretation

Because the program of Team 1 can not pass the acceptance test, we do not include it in our further evaluation. The test result can be presented in a 57x18 matrix (57 test cases, 18 accepted program versions). For each element in the matrix, three values are possible: P, F and M. These values indicate the three categories of the following test results:

Pass (P): The program passes the test case cleanly.

Fail (F): The program fails to pass the test case.

Operation	Number of Test Cases
Add Player	7
Remove Player	8
Move Player	13
Change Player 's Role	7
Create Team	13
Remove Team	2
Search Role	2
Search Player	2
Print Team	2
Print All	1
Total	57

Table 3: Test Cases Distribution

	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	Total
Pass	44	48	42	54	34	40	50	51	53	48	52	21	47	50	27	51	44	21	777
Fail	7	3	3	2	13	3	1	4	2	6	1	17	4	2	30	3	4	35	140
Maybe	6	6	12	1	10	14	6	2	2	3	4	19	6	5	0	3	9	1	109
PassRate	0.77	0.84	0.74	0.95	0.60	0.70	0.88	0.89	0.93	0.84	0.91	0.37	0.82	0.88	0.47	0.89	0.77	0.37	0.76
PassRate'	0.88	0.95	0.95	0.96	0.77	0.95	0.98	0.93	0.96	0.89	0.98	0.70	0.93	0.96	0.47	0.95	0.93	0.39	0.86

Table 4: Test Results for the Program Versions

Operation	Create Team	Remove Team	Add Player	Remove Player	Move Player	Change Role	Search Player	Search Role	Print Team	Print All	Total
Pass	132	34	94	115	176	107	34	34	34	17	777
Fail	52	2	11	17	38	13	2	2	2	1	140
Maybe	50	0	21	12	20	6	0	0	0	0	109
OpPassRate	0.56	0.94	0.80	0.75	0.75	0.85	0.94	0.94	0.94	0.94	0.76
OpPassRate'	0.78	0.94	0.88	0.91	0.84	0.95	0.94	0.94	0.94	0.94	0.86

Table 5: Test Results for the Operations

Maybe (M): The test case, designed to raise exceptions, can not apply to the program because the client side of the program deliberately forbids it. In this situation, we can not make sure whether the server is designed properly to raise the expected exceptions, so we put down "maybe" as the result. The definitions of the pass rate and the reliability in this paper, therefore, consider two conditions: Condition (a) includes the "Maybe" cases and Condition (b) excludes it.

We define pass rate for the Team *j* according to the two conditions as follows:

Condition (a): $PassRate_j = \frac{P_j}{C}$,

Condition (b): $PassRate'_j = \frac{P_j + M_j}{C}$,

where P_j is the number of "Pass" cases for program *j*, M_j is the number of "Maybe" cases program *j*, and C is the total number of cases applied to the programs (i.e. , 57).

The overall result is listed in Table 4.

From Table 4 we see that Team 16 and Team 19 have low pass rate as they fail to process many exceptions properly. In addition, Team 13 has many "maybe" cases due to their special user interface which avoids many illegal test cases designed to test for exception handling.

We also define pass rate for each operation *i* according to the two conditions as follows:

Condition (a): $OpPassRate_i = \frac{O_i}{T_i}$,

Condition (b): $OpPassRate_i' = \frac{O_i + OM_i}{T_i}$,

where O_i is the number of "Pass" test cases for operation *i* , OM_i is the number of "Maybe" test cases for operation *i*, and T_i is the total cases for operation *i.*

The result is listed in Table 5.

From Table 5 we can see that the operations *Create_Team* and *Move_Player* have the lowest pass rates. The reason is the complexity of these operations, as they need to consider more for the processing of normal cases and exceptions. Furthermore, the *Create_Team* operation has a large number of "Maybe" results, because many program versions forbid the test cases that would raise exceptions.

During the project testing and evaluation process, we discover some problems in the CORBA programs. These problems may affect the programs' reliability as well as their portability. Because some project teams lack experience in the Object-oriented (OO) methodology and CORBA program development, they have difficulties in designing Interface Definition Language (IDL). Here is a list of the typical problems:

Exception definitions

Missing exceptions: The specification required that exceptions be raised for most operations. However, some team's IDL does not consider the exceptions exclusively and comprehensively, and operational failures may occur. For example, the operation to add a player to a team should raise an exception when the number of player already reaches the maximum value. This can cause the implementation of client or server to fail should such situation occurs.

Extra exceptions: More exception definitions on the client side may not cause the program to fail but this usually implies redundant code, which is not reachable. However, extra exceptions on the server side usually represent incorrect results, as these exceptions will be unexpected to clients, causing them to fail.

Encapsulation

Some teams' IDLs include implementation-related operations and attributes. This practice breaks the encapsulation rule of the OO method. For example, the IDL of a team defines "PlayerExist()" to test if a duplicated player exists in the system, while this operation should be a private operation encapsulated in the implementation.

Misunderstanding the specifications

Quite a few failures result from misunderstanding of the specifications, and the corresponding operations defined in IDL carry wrong or inconsistent semantics. Consequently, the implementation of the program cannot respond correctly to the operation.

After the testing process was conducted, we detected 140 program defects among the program versions. These defects can be classify into the following three categories:

Category 1: Exception handling defects

A. Server side exception handling

a. Missing exception:

This happens when the server side program does not throw the necessary exception. Such a situation usually comes from the IDL definition problem as addressed above. Moreover, the implementation may also fail to check if there should be an exception.

b. Extra or Wrong exception:

Extra exception is considered as a wrong one since the client will not be able to recognize the exception, when the server throws an unexpected exception. This defect seldom occurs. However, it may occur due to the exception scope problem. For example, an interface "Team" defines an exception "DuplicateName" while another interface "TeamManager" also has the same name exception. When in the operation of "TeamManager", the program wants to raise an exception of "Team::DuplicateName". Nevertheless, without the scope modifier "Team::", the default exception "TeamManager::DuplicateName" is raised instead.

B. Client side exception handling

a. Missing exception:

The programmers forget to catch the expected exception at the client. This kind of defect occurs frequently since some of programmers are not familiar with the CORBA exception mechanism.

b. Wrong exception:

The programmers catch the exception correctly, but give an incorrect response. This kind of defect occurs when a special process is needed in the exception handling code. The conditions for extra exception on the client side represent unreachable code (no test cases can trigger the code) and they are not accounted for in our testing.

Category 2: Memory management defects:

The CORBA programs also experience the same memory management problem as traditional programs. Here we only address the CORBA-related memory management defects.

a. "_duplicate()/_release()" problem:

ORB adds a memory management mechanism "_duplicate()/_release()," which allocates and reallocates the memory based on an object reference number. Programmers may need to call "_duplicate()/_release()" to change the reference number. If programmers fail to manage it correctly, memory leaks happen or mysterious failures may occur.

b. Language mapping problem:

Some problems come from inadequate memory mapping. For example, the following statement can be defined in IDL:

```
typedef string <MAXLEN>NameStr;
```

However, the implementation code maybe written as

	Exception	Memory	Other	Total
Server	49 (missing: 43,extra or wrong : 6)	17	6	72
Client	51 (missing: 38,wrong: 13)	8	9	68
Total	100	25	15	140

Table 6: Distribution of Defects

follows:

```
NameStr name;
strcpy(name,<some_string>);
```

This looks fine. But the strcpy() may cause the problem of no memory allocation for the variable *name*. This defect occurs due to the misleading definition in IDL. When mapping to C++, it just maps the "string" to "char *" in the Orbix/C++, thus causing implementation problems.

Category 3: Other defects

Other defects similar to traditional programs also arise in the project. Most of them are related to user interface or process parameters.

According to the above definitions, the distribution of the defects is recorded, classified, and shown in Table 6.

From Table 6, we can see that over 70 percent of the total defects come from exception handling. This indicates that exception handling routines are the most difficult part of CORBA programming for distributed systems.

4. RELIABILITY MEASUREMENT OF THE CORBA PROGRAMS

Software reliability [6] is the probability of failure-free software operation for a specified time in a specified environment. We use the similar procedure specified in [9] to evaluate software reliability in our experiment. We note, however, that it is not easy to obtain execution time for CORBA programs, as many factors affect the operation execution time. They include, for example, programming language, platform, ORB implementation, user implementation, etc. Since it is difficult for us to get an accurate execution time measure for each operation in these programs, we evaluate the reliability of each accepted program based on the defects we get from our test and the probability of each operation. We test and evaluate the client and server for each program as a whole, and assume that each test case has the same execution time for the same program.

Operational profile [9] is a list of occurrence probabilities of each operation in the input domain of an application. Because the application in our experiment is a new information management system, the operational profile cannot be obtained from any historical data. Consequently, we have to estimate the occurrence probability of each operation. This is shown in Table 7. Based on the test results and operational profile, we define the reliability for each program j as the following two conditions:

Condition (a): $R_j = \sum_{i=1}^{n} \frac{OP_{i,j}}{T_i} \lambda_i$,

Condition (b): $R'_j = \sum_{i=1}^{n} \frac{OP_{i,j} + M_{i,j}}{T_i} \lambda_i$, where

n - number of operations (i.e. , 10)

R_j - Reliability for program j

R_j' - Reliability for program j (treat "maybe" as pass)

$OP_{i,j}$-"Pass" test cases for operation i , program j

$M_{i,j}$-"Maybe" test cases for operation i , program j

T_i - total cases for operation i

λ_i- Probability for operation i

Operation	Probability
Add Player	15%
Remove Player	15%
Move Player	15%
Change Player 's Role	15%
Create Team	10%
Remove Team	10%
Search Role	5%
Search Player	5%
Print Team	5%
Print All	5%

Table 7: Operational Profile

The results are listed in Table 8.

We also list the average reliability for *Visibroker*/*Orbix* program, and Java/C++ programs, as shown in Table 9.

Team	P2	P2	P3	P4	P5	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	Avg.
R	0.817	0.817	0.851	0.852	0.977	0.79	0.911	0.913	0.955	0.902	0.92	0.508	0.859	0.915	0.601	0.915	0.882	0.848
R'	0.923	0.923	0.969	0.959	0.985	0.969	0.992	0.955	0.985	0.932	0.992	0.732	0.955	0.971	0.601	0.952	0.952	0.927

Table 8: Reliability for Each Team

	R	R'
Orbix	0.832	0.926
Visibroker	0.845	0.927
C++	0.799	0.879
JAVA	0.883	0.964

Table 9: Average Reliability Measures for Different Classes Programs

From Table 9, we can see that the reliability of *Visibroker* programs is higher than that of *Orbix* programs. Moreover, the reliability of the teams using Java is higher than those using C++. The result does not necessary mean that using *Visibroker* and Java is better than using *Orbix* and C++. Instead, this may be due to the CORBA mapping for C++, which is more complicated than that for Java. Moreover, the programmers are generally more familiar with Java than with C++ according to their prior project experience.

5. INTEROPERABILITY ISSUES ABOUT CORBA PROGRAMS

Interoperability is the ability of two or more systems or components to exchange information and to use the information that has been exchanged [5]. Interoperability issues for CORBA specification is discussed in lots of papers, while [8] addresses an evaluation framework for interoperability of CORBA on WWW. The issues relating to interoperability are raised in [1] for embedded systems, in [14] for virtual Intranet, and in [7] for open systems. The evaluation for CORBA interoperability is also addressed in [3,11].

In the Common Object Request Broker Architecture and Specification 2.2 [12], *interoperability* means "interORBability," where the Object Request Broker (ORB) is the middle-ware that handles the communication details between the objects. Many papers discuss interoperability issues at this level. The CORBA 2.0 standard adopted in December of 1994, for example, defines "true" interoperability by specifying how ORBs from different vendors can communicate using a common protocol. However the interoperability among CORBA components, typically in a client-server relationship, is seldom fully assessed in the literature. In particular, there is a lack of experimental evaluation and assessment of interoperability from the software engineering viewpoint.

In this section, we perform a detailed analysis on 18 accepted programs for the assessment of their interoperability. We consider a broader definition of *interoperability* as components cooperating each other in a distributed architecture, no matter whether they are in the same ORB or not. The interoperability is defined by how easy it is to inter-exchange the client code and the object implementation code of any pair of program versions. Since all the programs are based on the same requirement specifications, we try to exchange the client and object implementation of the programs and evaluate the difficulty of this task. For example, for a pair of program (A, B), we can use the client code of program A and the object implementation code of program B (or visa versa) and see if they are interoperable. We give the following five assessments marks for interoperability:

1 very difficult to inter-operate different client and server from the program pair.

2 possible to inter-operate, but with considerable effort for code modification.

3 interoperable with moderate effort.

4 interoperable with some effort.

5 readily interoperable with minimal effort.

The result of this assessment effort is shown as a 18x18 matrix in Table 10. Note that this is a symmetric matrix. The overall interoperability assessment for each program, taken as the average of its interoperability mark with respect to other programs, is shown in the last row (or column). The overall interoperability mark for this project is shown in the intersection of the last row and the last column.

From Table 10 we can see that there is clearly a lack of interoperability among these 18 program versions (indicated by the low average mark of 1.42), even though they are developed based on the same specification requirements. Programs P3, P6, P7, P11, P14, P17, P19, in particular, are extremely difficult to inter-operate with other program versions. Only a few pairs of programs achieve higher interoperability marks, but they are sparse.

One potential reason for the lack of interoperability may be due to the usage of different CORBA vendor platforms. To examine this hypothesis, we separate those programs with *Visibroker* implementations from those with *Orbix* implementations, and show their corresponding interoperability matrices. These matrices are a subset of Table 10. Consequently, we obtain the measure for the interoperability marks: The average for *Visibroker* programs

C\S	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	AVG
P2	-	1	1	1	1	1	3	3	5	1	3	2	1	2	3	1	2	1	1.88
P3	1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1.00
P4	1	1	-	2	1	1	2	2	2	1	2	3	1	2	1	1	2	1	1.53
P5	1	1	2	-	1	1	2	2	2	1	2	2	1	2	2	1	3	1	1.59
P6	1	1	1	1	-	1	1	1	1	1	1	1	1	1	1	1	1	1	1.00
P7	1	1	1	1	1	-	1	1	1	1	1	1	1	2	1	1	1	1	1.06
P8	3	1	2	2	1	1	-	2	3	1	3	2	1	2	2	1	2	1	1.76
P9	3	1	2	2	1	1	2	-	2	1	2	2	1	2	3	1	2	1	1.71
P10	5	1	2	2	1	1	3	2	-	1	4	2	1	2	2	1	2	1	1.94
P11	1	1	1	1	1	1	1	1	1	-	1	1	1	1	1	1	1	1	1.00
P12	3	1	2	2	1	1	3	2	4	1	-	2	1	1	2	1	2	1	1.76
P13	2	1	3	2	1	1	2	2	2	1	2	-	1	1	2	1	2	1	1.59
P14	1	1	1	1	1	1	1	1	1	1	1	1	-	1	1	1	1	1	1.00
P15	2	1	2	2	1	2	2	2	2	1	1	1	1	-	1	1	1	1	1.41
P16	3	1	1	2	1	1	2	3	2	1	2	2	1	1	-	1	3	1	1.65
P17	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-	1	1	1.00
P18	2	1	2	3	1	1	2	2	2	1	2	2	1	1	3	1	-	1	1.65
P19	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-	1.00
AVG	1.88	1.00	1.53	1.59	1.00	1.06	1.76	1.71	1.94	1.00	1.76	1.59	1.00	1.41	1.65	1.00	1.65	1.00	**1.42**

Table 10: Interoperability Matrix

is 1.49, while for the *Orbix* programs is 1.48. In other words, they improve only slightly when we examine programs implemented in the same CORBA system. Moreover, there is virtually no distinction in interoperability between *Visibroker* programs and *Orbix* programs.

Although the overall interoperability for these program versions is very low, there is a subset of the 18 programs whose interoperability among each other is high. These are the programs P2, P10, and P12, whose interoperability matrix is shown in Table 11.

C\S	P2	P10	P12	AVG
P2	-	5	3	4.00
P10	5	-	4	4.50
P12	3	4	-	3.50
AVG	4.00	4.50	3.50	4.00

Table 11: Interoperability among a Subset of Programs

In evaluating the interoperability among these programs, the first problem we can immediate identify is the difference of the IDL interface design among these programs. In this project, we deliberately avoid to specify a common IDL for the programming teams. We consider that in real CORBA system implementation many different applications may run without an identical IDL. Interoperability among objects, then, has to be achieved by modifying the clients or the servers. We notice that the interoperability is low in P3, P4, P14 and P17, because P3, P4, P14 and P17 all have very special interfaces, which are not compatible with others'. Specifically, the lack of interoperability is contributed by the following factors:

Interface level

The lack of a similar IDL affects interoperability among the program versions. The difference in interface name is another problem. Although the application is the same for these programs, a considerable effort needs to be made to allow these programs to inter-operate with one another.

Operations and attributes

If different programs have different operation names (or attribute names), then we have to use the same mechanism, which we use to deal with the interface name problem. The required effort is also non-trivial. Parameters present another problem for interoperability. Although two operations may have the same function, the parameter's order and format may be different. This also makes the two operations incompatible.

Exception handling

Every team uses its own exceptions. They have different names and semantics. In order for the programs to work together, we need to check the exception handling portion of the program and make them compatible.

Other problems in IDL

As indicated in Section 3.2, some teams add special operations and attributes in their IDL. These operations and attributes are implementation-related. They should be encapsulated in the implementation part of the program and should not be placed in the IDL. Their appearances on IDL make it hard for these programs to be interoperable.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we discuss CORBA's testing, reliability and interoperability issues among multiple program versions from a small-size yet real project. We design the required test cases, develop the operational profile of the programs, and measure the reliability of these programs. We also discuss how to test the CORBA programs based on their specification and Interface Design Language. Moreover, we give defect analysis for the CORBA programs. We also discuss interoperability issues for the CORBA programs, provide a rough measurement of interoperability, and indicate the difficulties and mistakes that programmers made to make the CORBA programs less interoperable.

Based on the experience in testing CORBA programs and in evaluating their reliability and interoperability results, we plan to formulate requirement specification techniques and design constraints for the CORBA programming paradigm. These techniques and constraints should be amended to a traditional software engineering process for the development of CORBA programs in order to achieve higher reliability and interoperability for distributed systems. We will study the techniques to test, evaluate and develop software based on the CORBA paradigm. We will apply software fault-tolerance techniques to achieve high reliability and availability goals. The software engineering techniques and the software fault-tolerance techniques can be combined to form a CORBA development framework. We believe such a framework is important to the quest for reliability and interoperability that has been promised by the CORBA approach for distributed systems.

ACKNOLEDGEMENT

The work described in this paper was partially supported by a grant from the Research Grant Council of the Hong Kong SAR (Project No. CUHK4432/99E).

REFERENCES

[1] D. Allen, "CORBA Technology for cross-domain interoperability in embedded military systems, and issues in its use," *Proceedings of WORDS '96 Second Workshop*, pp. 173 -178, 1996.

[2] S. Baker, *CORBA Distributed Objects: Using Orbix*, Addison-Wesley, November 1997.

[3] T. Brando, "Interoperability and the CORBA Specification," *MITRE Document MP 95B-58*, <URL: http://www.mitre.org/research/domis/reports/UNO.html >, February 1995.

[4] Cetus Links, *CORBA Links*, <URL: http://www.cetus-links.org/oo_corba.html>

[5] Institute of Electrical and Electronics Engineers, *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, New York, NY, 1990.

[6] M.R. Lyu, *Handbook of Software Reliability Engineering*, McGraw Hill, 1996

[7] T. Mowbray, T. Brando, "Interoperability and CORBA-Based Open Systems," *Object Magazine, 3(3)*, pp. 50-54, September-October 1993.

[8] S. Mahajan, J. Chen, M. Li, C. Mingins, B. Meyer, "CORBA on WWW: Evaluative Framework for Interoperability Issues," *Proceedings of TOOLS 27*, pp. 351-360, 1998.

[9] J. Musa, "Introduction To Software Reliability Engineering And Testing" *Proceedings of The Eighth International Symposium on Software Reliability Engineering - Case Studies*, pp. 3-12, 1997.

[10] T. Mowbray, R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects,* August, 1995.

[11] Object Management Group, *Interoperable Name Service Enhancements RFP*, <URL: ftp://ftp.omg.org/pub/docs/orbos/97-12-33.ps>, 1997.

[12] Object Management Group, *"CORBA/IIOP 2.2 Specification,"* updated July 1998, <URL: http://www.omg.org/corba/corbaiiop.html>, 1998.

[13] J. Siegel, *Corba Fundamentals and Programming*, John Wiley and Sons, April, 1996.

[14] A. Zarli, V. Amar, H. Adeli, "Integrating STEP and CORBA for Applications Interoperability in the Future Virtual Enterprises Computer-based Infrastructures," *Proceedings of ISS '97,* pp. 309-315, 1997.