CORBA 2.2 Implementation

Kay Römer
Universität Frankfurt
FB Informatik
Postfach 11 19 32
D–60054 Frankfurt am Main

Arno Puder
International Computer Science Institute
1947 Center St.
Berkeley, CA 94704–1198
USA

Version 2.2.3

Contents

List of Figures

Chapter 1

What is MICO?

The acronym MICO expands to **MI**CO **I**s **CO**RBA. The intention of this project is to provide a *freely available* and *complete* CORBA 2.2 compliant implementation (see [5]). The difference to other free imlementations is, that MICO is developed for educational purposes and that the complete sources are available under the GNU–copyright notice (see chapter 7). We were inspired by Tanenbaum's MINIX project which is something similar in the UNIX community. The following design principles guided the implementation of MICO:

1. start from scratch: only use what standard UNIX API has to offer; don't rely on proprietary or specialized libraries.
2. use C++ for the implementation.
3. only make use of widely available, non–proprietary tools.
4. omit "bells and whistles": only implement what is required for CORBA compliant implementation.
5. clear design even for implementation internals to ensure extensibility.

Although the implementation of MICO is finished, you should visit its homepage frequently for updates. We will continue to develop MICO, providing bug fixes as well as new features. Information about the MICO project is available at:

Europe:	`http://www.vsb.cs.uni-frankfurt.de/~mico/`
USA:	`http://www.icsi.berkeley.edu/~mico/`

Further informations about MICO can be found in the book *MICO is CORBA* published by dpunkt.verlag (`http://www.dpunkt.de/mico`) in Europe and Morgan Kaufmann Publishers, Inc. (`http://www.mkp.com/books_catalog/3-93258-811-8.asp`) in North America. The book includes a CD with the complete source code of MICO as well as binaries for various platforms as ready to run executables. It explains how to install and use MICO. A little tutorial gets you going with a sample CORBA application. All features of MICO are well documented both in the manual and in online man–pages. MICO is fully interoperable with other CORBA implementations, such as Orbix from Iona or VisiBroker from Visigenic. The manual contains a step–by–step procedure showing how to connect MICO with other CORBA implementations. It even includes sample programs from various CORBA textbooks to show you all aspects of CORBA.

How to support MICO

The authors have worked very hard to make MICO a usable and free CORBA 2.2 compliant implementation. If you find MICO useful and would like to support it, there are two possible ways. First of all you can contribute to the development of MICO by implementing those parts of the CORBA standard, which are still missing in MICO. Although MICO is fully CORBA 2.2 compliant, there are some parts of the standard (like the CORBAservices) which are not mandatory and which we did not implement. We hope that our decision to place the complete sources of MICO under the GNU public license will encourage other people to contribute their code (see section 7 for details).

Another way to support MICO is by sending us a small donation which will help us to maintain MICO and to further develop it. If you wish to make a donation, please send it to:

> MICO Development
> Universität Frankfurt
> FB Informatik (20)
> Betriebssysteme und Verteilte Systeme
> Postfach 11 19 32
> 60054 Frankfurt am Main
> GERMANY

Chapter 2

Installation

This chapter explains from where MICO can be obtained, the prerequisites for compiling MICO, how to compile and install MICO, and on which platforms MICO has been tested.

2.1 Getting MICO

The latest MICO release is always available at

```
http://www.vsb.cs.uni-frankfurt.de/~mico/
http://www.icsi.berkeley.edu/~mico/
ftp://diamant.vsb.cs.uni-frankfurt.de/pub/projects/mico/mico-2.2.3.tar.gz
```

New releases are announced over the MICO mailing list. If you want to subscribe send a message containing

```
subscribe mico-devel
```

to `majordomo@vsb.cs.uni-frankfurt.de`.

2.2 Prerequisits

2.2.1 Unix

Before trying to compile MICO make sure you have installed the following software packages:

- gnu make version 3.7 or newer (required)
- C++ compiler and library (required):
  - g++ 2.7.2.x and libg++ 2.7.2, or
  - g++ 2.8.x and libg++ 2.8.x, or
  - egcs 1.x

- flex 2.5.2 or newer (optional)
- bison 1.22 or newer (optional)
- JDK 1.1.5 (SUN's Java developers kit) (optional)
- JavaCUP 0.10g (parser generator for Java) (optional)

`flex` and `bison` are only necessary if you change their input files (files having the suffix `.l` and `.y`) or if you want to compile the graphical user unterface. The last two items (JDK and JavaCUP) are only needed for the graphical interface repository browser, not for MICO itself. So you can get along without installing the Java stuff.

It is important that you use one of the above listed C++ compilers and a C++ library that matches the version of the compiler. Your best bet is using either egcs or g++ 2.8. In contrast to gcc 2.7.2 both of them have proper support for exceptions. egcs is a bit easier to install than g++, because it includes a matching C++ library.

2.2.2 Windows 95/NT

In order to run MICO on Windows 95 or NT you have to use the *Cygnus CDK beta 19*, a port of the GNU tools to Win32 or Microsoft's Visual–C++ compiler. For instructions on how to compile MICO using the Visual–C++ compiler, refer to the file `README-WIN32` in the main directory of the MICO sources.
Install the CDK by running its setup program. Note that you have to install it in the directory the setup program suggests (`c:\Cygnus\CDK\B19`); otherwise `bison` won't be able to find its skeleton files. Then create `c:\bin` and put an `sh.exe` into it. Likewise create `c:\lib` and put a `cpp.exe` into it:

```
mkdir c:\bin
copy c:\Cygnus\CDK\B19\H-i386-cygwin32\bin\bash.exe c:\bin\sh.exe
mkdir c:\lib
copy c:\Cygnus\CDK\B19\H-i386-cygwin32\lib\gcc-lib\2.7-B19\cpp.exe c:\lib
```

Now you are ready to unpack and compile MICO as described in section 2.3.

There are some problems with the current release of the CDK:

- On standalone machines which are not connected to a name server resolving IP addresses other than 127.0.0.1 into host names will hang forever. This is either a problem with the CDK or with Windows in general. On standalone machines you therefore have to make all servers bind to 127.0.0.1 by specifying `-ORBIIOPAddr inet:127.0.0.1:<port>` on the command line.
- The gcc 2.7 that comes with the CDK has broken exception handling. Furthermore it seems to be unable to use virtual memory, at least I get `out of virtual memory` errors although there is a lot of free swap space. I know there are ports of egcs and gcc 2.8 (which might do better), but didn't give them a try.

- There seems to be a problem with automatic TCP port number selection. Usually one binds to port number 0 and the system automatically picks an unused port for you. This basically works with CDK, but sometimes causes hanging connections. The solution is to always explicitely specify port numbers, i.e. give *all* servers—even ones that are started by `micod`—the option `-ORBIIOPAddr inet:<host>:<port>`, where `<port>` is nonzero.

2.3 Installing MICO

The MICO source release is shipped as a tar'ed and gzip'ed archive called

```
mico-2.2.3.tar.gz
```

Unpack the archive using the following command:

```
gzip -dc mico-2.2.3.tar.gz | tar xf -
```

You are left with a new directory `mico` containing the MICO sources. To save you the hassle of manually editing `Makefile`'s and such, MICO comes with a configuration script that checks your system for required programs and other configuration issues. The script, called `configure`, supports several important command line options:

`--help`
: Gives an overview of all supported command line options.

`--prefix=<install-directory>`
: With this options you tell configure where the MICO programs and libraries should be installed after compilation. This defaults to `/usr/local`.

`--disable-optimize`
: Do not use the `-O` option when compiling C/C++ files. It is now safe to use this option because only files that do not use exceptions are compiled using `-O`, which is why optimization is now turned on by default.

`--enable-debug`
: Use the `-g` option when compiling C/C++ files.

`--enable-repo`
: Use the `-frepo` flag when compiling C++ files. This works only with a patched g++ 2.7.2 and will greatly reduce the size of the binaries, at the cost of much slower compilation (this option instructs g++ to do some sort of template repository). You *must* use this option on HP–UX, otherwise you will get lots of error during linking.

`--disable-shared`
: Build the MICO library as a static library instead as a shared one. Shared libraries currently only work on ELF based systems (e.g., Linux, Solaris, Digital Unix, AIX, and HP–UX). If you do not use the `--disable-shared` option you have to make

sure the directory where the MICO library resides is either by default searched for shared libraries by the dynamic linker (`/usr/lib` and `/lib` on most systems) or you have to include the directory in the environment variable that tells the dynamic linker where to search for additional shared libraries. This variable is called `LIBPATH` on AIX, `SHLIB_PATH` on HP–UX and `LD_LIBRARY_PATH` on all the other systems. To run the generated binaries before doing a `make install` you have to set this environment variable like this:

```
    # AIX
    export LIBPATH=<mico-path>/mico/orb:$LIBPATH
    # HP-UX
    export SHLIB_PATH=<mico-path>/mico/orb:$SHLIB_PATH
    # others
    export LD_LIBRARY_PATH=<mico-path>/mico/orb:$LD_LIBRARY_PATH
%$
```

where `<mico-path>` is the absolute path of the directory the MICO sources were unpacked in.

`--disable-dynamic`
: This option disables dynamic loading of CORBA objects into a running executable. For dynamic loading to work your system must either support `dlopen()` and friends or `shl_load()` and friends. See section 4.3.4 for details.

`--enable-final`
: Build a size optimized version of the MICO library. This will need lots of memory during compilation but will reduce the size of the resulting library a lot. Works with and without `--enable-shared`. Does not work on HP–UX.

`--disable-mini-stl`
: As mentioned before, MICO makes use of the Standard Template Library (STL). For environments that do not provide an STL implementation, MICO comes with its own slim STL (called MiniSTL), which is simply a subset of the standard STL sufficient to compile MICO. By default MICO will use MiniSTL. If you want to use the system supplied STL for some reason you have to use the option `--disable-mini-stl`. MiniSTL works well with g++ and greatly reduces compilation time and size of the binaries. Using MiniSTL one could try to compile MICO using a C++ compiler other than g++. But this still has not been tested and may therefore lead to problems.

`--disable-except`
: Disable exception handling. On some platforms (e.g., DEC alpha) g++ has very buggy exception handling support that inhibit the compilation of MICO with exception handling enabled. If this happens try turning off exception handling using this option.

`--with-qt=<qt-path>`
: Enable support for QT. `<qt-path>` is the directory where QT has been installed in.

`--with-gtk=<gtk-path>`
: Enable support for GTK. `<gtk-path>` is the directory where GTK has been installed in.

`--with-tcl=<tcl-path>`
: Enable support for TCL. `<tcl-path>` is the directory where TCL has been installed in.

`--with-ssl=<SSLeay-path>`
: Enable support for SSL. `<SSLeay-path>` is the directory where SSLeay has been installed in.

Now you should run `configure` with the proper command line options you need, e.g.:

```
cd mico
./configure --with-qt=/usr/local/qt
```

Use `gmake` to start compilation and install the programs and libraries, possibly becoming root before installation:

```
gmake
gmake install
```

On some systems you have to take special actions after installing a shared library in order to tell the dynamic linker about the new library. For instance on Linux you have to run `ldconfig` as root:

```
/sbin/ldconfig -v
```

2.4 Supported Platforms

Mico has been tested on the following operating systems:

- Solaris 2.5.1 on Sun SPARC
- AIX 4.2 on IBM RS/6000
- Linux 2.0.x on Intel x86
- Digital Unix 4.x on DEC Alpha
- HP–UX 10.20 on PA–RISC
- Ultrix 4.2 on DEC Mips (no shared libs, no dynamic loading)
- Windows 95/NT (no shared libs)

Addionally some users reported Mico runs on the following platforms:

- FreeBSD 3.x on Intel x86
- SGI–Irix on DEC Mips
- OS/2 on Intel x86 using emx 0.9

Please let us know if you fail/succeed in running Mico on any unsupported platform.

Chapter 3

Guided tour through MICO

3.1 Objects in distributed systems

Modern programming languages employ the *object paradigm* to structure computation within a single operating system process. The next logical step is to distribute a computation over multiple processes on one single or even on different machines. Because object orientation has proven to be an adequate means for developing and maintaining large scale applications, it seems reasonable to apply the object paradigm to distributed computation as well: objects are distributed over the machines within a networked environment and communicate with each other.

As a fact of life the computers within a networked environment differ in hardware architecture, operating system software, and the programming languages used to implement the objects. That is what we call a *heterogenous distributed environment*. To allow communication between objects in such an environment one needs a rather complex piece of software called a *middleware platform*. Figure 3.1 illustrates the role of a middleware platform within a heterogenous distributed environment.

The *Common Object Request Broker Architecture (CORBA)* is a specification of such a middleware platform by the *Object Management Group (OMG)* (see [5]). MICO provides

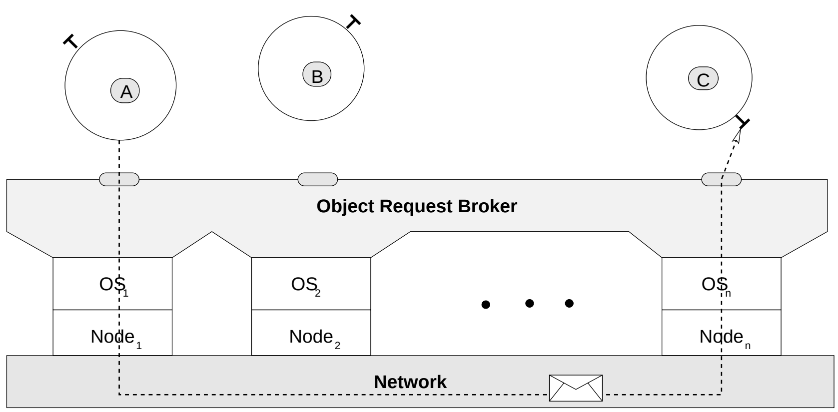

Figure 3.1: Middleware support for objects in distributed systems.

a full CORBA 2.2 compliant implementation. CORBA addresses the following issues:

object orientation
: objects are the basic building blocks of CORBA applications.

distribution transparency
: a caller uses the same mechanisms to invoke an object whether it is located in the same address space, the same machine or on a remote machine.

hardware–, operating system–, and language independence
: CORBA components can be implemented using different programming languages on different hardware architectures running different operating systems.

vendor independence
: CORBA compliant implementations from different vendors interoperate.

CORBA is an open standard in the sense that anybody can obtain the specification and implement it like we did. Besides its technical features this is considered one of CORBA's main advantages over other proprietary solutions.

3.2 State of development

Mico is a fully compliant CORBA 2.2 implementation. Everything that is implemented is CORBA 2.2 compliant, including but not limited to the following features:

- Dynamic Invocation Interface (DII)
- Dynamic Skeleton Interface (DSI)
- IDL to C++ mapping
- Interface Repository (IR)
- graphical Interface Repository browser that allows you to invoke arbitrary methods on arbitrary interfaces
- IIOP as native protocol
- IIOP over SSL
- modular ORB design: new transport protocols and object adapters can easily be attached to the ORB — even at runtime using loadable modules
- support for nested method invocations
- interceptors
- `Any` offers an interface for inserting and extracting contructed types that were not known at compile time

- `Any` and `TypeCode` support recursive subtyping as defined by the RM–ODP
- support of recursive data types
- full BOA implementation, including all activation modes, support for object migration, object persistence and the implementation repository
- BOA can load object implementations into clients at runtime using loadable modules
- Portable Object Adapter (POA)
- support for using MICO from within X11 applications (Xt and Qt)
- naming service
- event service
- relationship service
- property service
- DynAny support

Our goal is to keep the core of MICO fully compliant to the latest version of the CORBA specification, while integrating new CORBA services. Be sure to check the MICO homepage frequently for updates.

3.3 Sample Program

To get you started with MICO, this section presents an example of how to turn a single–process object oriented program into a MICO application.

3.3.1 Standalone program

Imagine a bank which maintains accounts of its customers. An object which implements such a bank account offers three operations[1]: *deposit* a certain amount of money, *withdraw* a certain amount of money, and an operation called *balance* that returns the current account balance. The state of an account object consists of the current balance. The following C++ code fragment shows the class declaration for such an account object:

```
class Account {
    long _current_balance;
public:
    Account ();
    void deposit (unsigned long amount);
    void withdraw (unsigned long amount);
    long balance ();
};
```

[1] This is a somewhat idealistic assumption but sufficient for the scope of this example.

The above class declaration describes the *interface* and the *state* of an account object, the actual *implementation* which reflects the behavior of an account, is shown below:

```
Account::Account ()
{
    _current_balance = 0;
}
void Account::deposit (unsigned long amount)
{
    _current_balance += amount;
}
void Account::withdraw (unsigned long amount)
{
    _current_balance -= amount;
}
long Account::balance ()
{
    return _current_balance;
}
```

Here is a piece of code that makes use of a bank account:

```
#include <iostream.h>

int main (int argc, char *argv[])
{
    Account acc;

    acc.deposit (700);
    acc.withdraw (250);
    cout << "balance is " << acc.balance() << endl;

    return 0;
}
```

Since a new account has the initial balance of 0, the above code will print out *"balance is 450"*.

3.3.2 MICO application

Now we want to turn the standalone implementation from the previous section into a Mico application. Because CORBA objects can be implemented in different programming languages[2] the specification of an object's *interface* and *implementation* have to be separated. The implementation is done using the selected programming language, the interface is specified using the so called *Interface Definition Language (IDL)*. Basically the CORBA IDL looks like C++ reduced to class and type declarations (i.e., you *cannot* write down the implementation of a class method using IDL). Here is the interface declaration for our account object in CORBA IDL:

[2]The CORBA specification currently defines language mappings for a variety of high level languages like C, C++, Smalltalk, Cobol and Java.

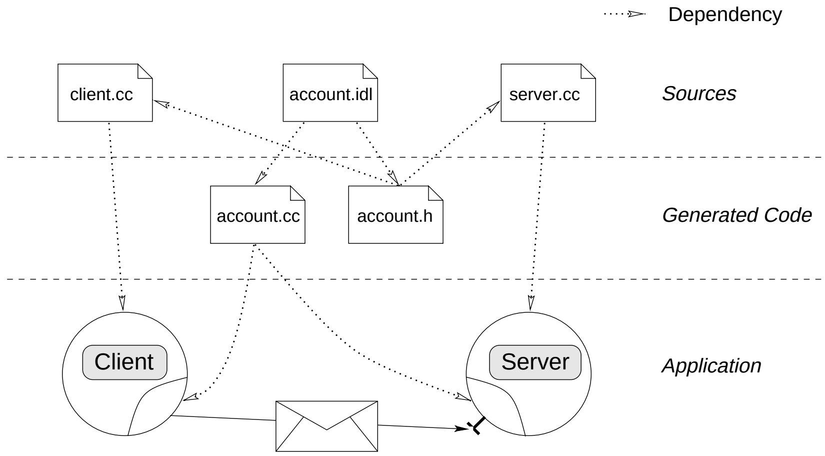

Figure 3.2: Creation process of a MICO application.

```
interface Account {
    void deposit (in unsigned long amount);
    void withdraw (in unsigned long amount);
    long balance ();
};
```

As you can see it looks quite similar to the class declaration in section 3.3.1. The `in` declarator declares `amount` as an input parameter to the `deposit()` and `withdraw()` methods. Usually one would save the above declaration to a file called `account.idl`.

The next step is to run this interface declaration through the *IDL compiler* that will generate code in the selected implementation programming language (C++ in our example). The MICO IDL compiler is called `idl` and is used like this:

```
idl account.idl
```

The IDL compiler will generate two files: `account.h` and `account.cc` (see figure 3.2). The former contains class declarations for the base class of the account object implementation and the stub class a client will use to invoke methods on remote account objects. The latter contains implementations of those classes and some supporting code. For each interface declared in an IDL–file, the MICOIDL compiler will produce three C++ classes[3].

The three classes are depicted in figure 3.3 between the two dashed lines. The class `Account` serves as a base class. It contains all definitions which belong to the interface `Account`, like local declarations of user defined data structures. This class also defines a pure virtual function for each operation contained in the interface. The following shows a bit of the code contained in class `Account`:

```
// Code excerpt from account.h
class Account : virtual public CORBA::Object {
    ...
public:
```

[3]Note that C++ is currently the only language which is supported by MICO.

```
    ...
    virtual void deposit (CORBA::ULong amount) = 0;
    virtual void withdraw (CORBA::ULong amount) = 0;
    virtual CORBA::Long balance () = 0;
}
```

The class `Account_skel` is derived from `Account`. It adds a dispatcher for the operations defined in class `Account`. But it does not define the pure virtual functions of class `Account`. The classes `Account` and `Account_skel` are therefore abstract base classes in C++ terminology. To implement the account object you have to subclass `Account_skel` providing implementations for the pure virtual methods `deposit()`, `withdraw()` and `balance()`.

The class `Account_stub` is derived from class `Account` as well. In contrast to class `Account_skel` it defines the pure virtual functions. The implementation of these functions which is automatically generated by the IDL–compiler is responsible for the parameter marshalling. The code for `Account_stub` looks like this:

```
// Code excerpt from account.h and account.cc
class Account;
typedef Account *Account_ptr;

class Account_stub : virtual public Account {
    ...
public:
    ...
    void deposit (CORBA::ULong amount)
    {
       // Marshalling code for deposit
    }
    void withdraw (CORBA::ULong amount)
    {
       // Marshalling code for withdraw
    }
    CORBA::Long balance ()
    {
       // Marshalling code for balance
    }
}
```

This makes `Account_stub` a concrete C++ class which can be instantiated. The programmer never uses the class `Account_stub` directly. Access is only provided through class `Account` as will be explained later.

It is worthwile to see where the classes `Account` and `Account_skel` are derived from. `Account` inherits from `Object`, the base class for all CORBA objects. This class is located in the MICO library. The more interesting inheritance path is for `Account_skel`. `Account_skel` inherits from `MethodDispatcher`, a class located again in the MICO library. This class is responsible for dispatching a method invocation. It maintains a list of method dispatchers[4]. The class `MethodDispatcher` inherits from `DynamicImplementation`. This class represents the interface to the *dynamic skeleton interface* (DSI) which is described

[4] In this example the list contains only one dispatcher, namely for the Account–object. Later when we discuss interface inheritance this list will contain a dispatcher for each class in the inheritance hierarchy.

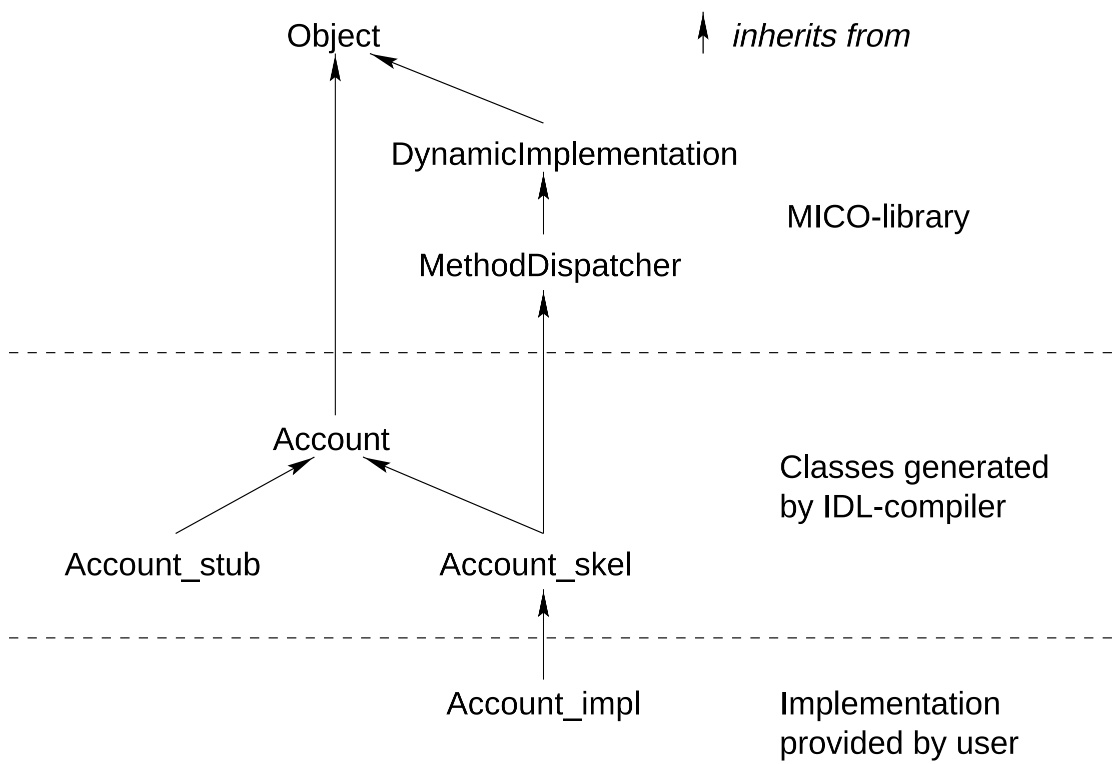

Figure 3.3: Inheritance relationship between stub– and skeleton classes.

by the CORBA standard in chapter [18.4.5]. This is a notably feature of Mico: the code generated by the MicoIDL compiler makes use of CORBAs DII and DSI.

Up until now we have written the interface of an account object using CORBA IDL, saved it as `account.idl`, ran it through the IDL compiler which left us with two files called `account.cc` and `account.h` that contain the class declarations for the account implementation base class (`Account_skel`) and the client stub (`Account_stub`). Figure 3.2 illustrates this. What is left to do is to subclass `Account_skel` (implementing the pure virtual methods) and write a program that uses the bank account. Here we go:

```
#include "account.h"

class Account_impl : virtual public Account_skel
{
private:
  CORBA::Long _current_balance;

public:
  Account_impl()
  {
    _current_balance = 0;
  };
  void deposit( CORBA::ULong amount )
  {
    _current_balance += amount;
  };
  void withdraw( CORBA::ULong amount )
  {
    _current_balance -= amount;
  };
  CORBA::Long balance()
  {
    return _current_balance;
  };
};


int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  // server side
  Account_impl* server = new Account_impl;
  CORBA::String_var ref = orb->object_to_string( server );
  cout << "Server reference: " << ref << endl;

  //------------------------------------------------------------------

  // client side
  CORBA::Object_var obj = orb->string_to_object( ref );
  Account_var client = Account::_narrow( obj );

```

```
45:   client->deposit( 700 );
46:   client->withdraw( 250 );
47:   cout << "Balance is " << client->balance() << endl;
48:
49:   // We don't need the server object any more. This code belongs
50:   // to the server implementation
51:   CORBA::release( server );
52:   return 0;
53: }
```

Lines 3–25 contain the implementation of the account object, which is quite similar to the implementation in section 3.3.1. Note that the class `Account_impl` inherits the from class `Account_skel`, which contains the dispatcher for this interface, via a virtual public derivation. Although the keyword `virtual` is not required in this case, it is a good practise to write it anyway. This will become important when interface inheritance is discussed in section 5.5.

The `main()` function falls into two parts which are seperated by the horizontal line (line 39): Above the separator is the server part that provides an account object, below the line is the client code which invokes methods on the account object provided by the server part. Theoretically the two parts could be moved to two seperate programs and run on two distinct machines and almost nothing had to be changed in the code. This will be shown in the next section.

In line 32 the Mico initialization function is used to obtain a pointer to the *Object Request Broker (ORB)* object—a central part of each CORBA implementation. Among others the ORB provides methods to convert object references into a string representation and vice versa. In line 35 an account object called `server` is instantiated. Note that it is not permitted to allocate CORBA objects on the run–time stack. This is because the CORBA standard prescribes that every object has to be deleted with a special function called `CORBA::release()`. Automatic allocation of an object would invoke its destructor when the program moves out of scope which is not permissible. In our little sample program the server object is deleted explicitly in line 51.

In line 36 the ORB is used to convert the object reference into a string that somehow has to be transmitted to the client (e.g., using Email, a name service or a trader). In our example client and server run in the same address space (i.e. the same process) so we can turn the string into an object reference back again in line 42. Line 43 uses the `Account::_narrow()` method to downcast the object reference to an `Account_var`. The rest of `main()` just uses the account object which was instantiated in line 35.

`Account_var` is a smart pointer to `Account` instances. That is an `Account_var` behaves like an `Account_ptr` except that the storage of the referenced object is automatically freed via the aforementioned `release()` function when the `Account_var` is destroyed. If you use `Account_ptr` instead you would have to use `CORBA::release()` explicitly to free the object when you are done with it (*never* use `delete` instead of `CORBA::release()`).

Assuming the above code is saved to a file called `account_impl.cc` you can compile the code like this[5]:

[5] `mico-c++` and `mico-ld` are wrapper scripts for the C++ compiler and the linker, see section 4.6 for details

```
mico-c++ -I. -c account_impl.cc -o account_impl.o
mico-c++ -I. -c account.cc -o account.o
mico-ld -I. -o account account_impl.o account.o -lmico2.2.3
```

This will generate an executable called `account`. Running it produces the following output:

```
Server reference: IOR:1200000069643a6d69636f2d6c6f63616c2d626f61310\
1000000b77a00001600000001200000069643a6d69636f2d6c6f63616c2d626f6131
Balance is 450
```

You can find the source code for this example in the `demo/account` directory within the MICO source tree.

3.3.3 Separating client and server

CORBA would be pretty useless if you always had to run the object implementation (*server*) and the *client* that uses the server in the same process. Here is how to separate the client and server parts of the example in the previous section into two processes running on the same or on different machines[6].

One problem you have to cope with when moving object implementation and client into separate address spaces is how the client gets to know the server. The solution to this problem is called a *naming service*.

Stringified Object References

The example in section 3.3.2 already used the ORB methods `object_to_string()` and `string_to_object()` to make a stringified representation of an object reference and to turn back this string into an object, respectively.

When separating client and server you have to find a way to transmit the stringified object reference from the server to the client. If client and server run on machines that share a single file system you can make the server write the string into a file which is read by the client. Here is how to do it:

```
// file account_server.cc

#include <iostream.h>
#include <fstream.h>
#include "account.h"

class Account_impl : virtual public Account_skel
{
  // unchanged, see section "MICO Application"
  // ...
};

```

[6]Of course you can have some of the object implementations in the same process and some in other processes. The ORB hides the actual locations of the object implementations from the user

```
13:
14: int main( int argc, char *argv[] )
15: {
16:   // ORB initialization
17:   CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
18:   CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );
19:
20:   Account_impl* server = new Account_impl;
21:   CORBA::String_var ref = orb->object_to_string( server );
22:   ofstream out ("/tmp/account.objid");
23:   out << ref << endl;
24:   out.close ();
25:
26:   boa->impl_is_ready( CORBA::ImplementationDef::_nil() );
27:   orb->run ();
28:   CORBA::release( server );
29:   return 0;
30: }
```

`Account_impl`, the implementation of the account object in lines 7–11 is the same as in section 3.3.2. The `main()` function performs ORB and BOA[7] initialization in lines 16–18, which will evaluate and remove CORBA specific command line options from `argv`, see section 4.1.1 for details. In line 20 an account object is created, lines 21–24 obtain a stringified object reference for this object and write it to a file called `account.objid`.

In line 26 the `impl_is_ready()` method of the BOA is called to activate the objects implemented by the server. The ORB method `run()`, which is invoked in line 27 will enter a loop to process incoming invocations[8]. Just before returning from `main()`, `CORBA::release()` is used in line 28 to destroy the account server object.

```
// file account_client.cc

#include <iostream.h>
#include <fstream.h>
#include "account.h"

int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  ifstream in ("/tmp/account.objid");
  char ref[1000];
  in >> ref;
  in.close ();

  CORBA::Object_var obj = orb->string_to_object (ref);
  Account_var client = Account::_narrow( obj );

  client->deposit( 700 );
```

[7] The Basic Object Adapter

[8] You can make `run()` exit by calling the ORB method `shutdown()`, see section 4.3.4 for details.

```
22:   client->withdraw( 250 );
23:   cout << "Balance is " << client->balance() << endl;
24:
25:   return 0;
26: }
```

After ORB and BOA initialization the client's `main()` function reads the stringified object reference in lines 13–16 and turns it back into an account object stub in lines 18–19. After making some method invocations in lines 21-23 `client` will be destroyed automatically because we used and `Account_var` smart pointer.
Compile the client and server programs like this:

```
mico-c++ -I. -c account_server.cc -o account_server.o
mico-c++ -I. -c account_client.cc -o account_client.o
mico-c++ -I. -c account.cc -o account.o
mico-ld -o server account_server.o account.o -lmico2.2.3
mico-ld -o client account_client.o account.o -lmico2.2.3
```

First run `server` and then `client` in a different shell. The output from `client` will look like this:

```
Balance is 450
```

Note that running the client several times without restarting the server inbetween will increase the balance the client prints out by 450 each time! You should also note that client and server do not necessarily have to run on the same machine. The stringified object reference, which is written to a file called `/tmp/account.objid`, contains the IP address and port number of the server's address. This way the client can locate the server over the network. The same example would also work in a heterogeneous environment. In that case you would have to compile two versions of `account.o`, one for each hardware architecture. But the conversion of the parameters due to different data representations is taken care of by Mico.

Naming Service

What we have actually done in the last section is to implement some very simple kind of *naming service* on top of the file system. A naming service is a mapping between names and addresses which allows you to look up the address for a given name. For example a phone directory is a naming service: it maps people's names to phone numbers.

In the CORBA context a naming service maps names to object references. The simple naming service we implemented in the previous section maps file names to stringified object references. The OMG has defined a more elaborate naming service as a set of CORBA objects, an implementation of which is now shipped with Mico. To use the name service you have to

- run the name service daemon `nsd`
- tell server and client the address of `nsd` using the `-ORBNamingAddr` option (see section 4.1.1 for details)

- make the server register its offered objects with the name service
- make the client query the name server for the server

There is a program called `nsadmin` that can be used to browse and change the contents of the naming service. The `demo/naming` directory contains an example how to use the name service.

The MICO Binder (CORBA Extension)

There is still one problem left: How do you get an object reference for the naming service itself? Especially if the naming service and the client reside on machines that do not share a file system that could be used to pass around stringified object references as in the previous section[9]. Because the CORBA standard does not offer a solution to this problem MICO has to invent its own. Because it might be useful for other purposes as well we decided to make the solution available to you, dear user. Note that using this feature makes your programs incompatible with other CORBA implementations.

The MICO Binder is a very simple naming service that maps (*Address*, *RepositoryId*) pairs to object references. A *RepositoryId* is a string that identifies a CORBA IDL–object and consists of the absolute name of the IDL–object and a version number. RepositoryId's are generated by the IDL compiler. The RepositoryId for the `Account` interface looks like this:

```
IDL:Account:1.0
```

See section [6.6] of [5] for details on RepositoryId's. An *Address* identifies one process on one computer. MICO currently defines three kinds of addresses: *internet addresses*, *unix addresses*, and *local addresses*. An *internet address* is a string with the format

```
inet:<host name>:<port number>
```

which refers to the process on machine `<host name>` that owns the TCP port `<port number>`. *Unix addresses* look like

```
unix:<socket file name>
```

and refer to the process on the current machine that owns the unix–domain socket[10] bound to `<socket file name>`. *Local addresses* look like

```
local:
```

and refer to the process they are used in (i.e., *this* process). Here is an adaption of the account example which uses the MICO binder:

[9] The CORBA standard offers the ORB method `resolve_initial_references()` to obtain an object reference for the naming service. But that only moves the problem to the ORB instead of solving it.

[10] Unix–domain sockets are named, bidirectional pipes.

```
// file account_server2.cc

#include "account.h"

class Account_impl : virtual public Account_skel
{
  // unchanged, see section "MICO Application"
  // ...
};


int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  Account_impl* server = new Account_impl;

  boa->impl_is_ready( CORBA::ImplementationDef::_nil() );
  orb->run ();
  CORBA::release( server );
  return 0;
}
```

The server is essentially the same as in 3.3.3 except that it does not write a stringified object reference to a file. Here is the client:

```
// file account_client2.cc

#include "account.h"


int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  CORBA::Object_var obj
    = orb->bind ("IDL:Account:1.0", "inet:localhost:8888");
  if (CORBA::is_nil (obj)) {
     // no such object found ...
  }
  Account_var client = Account::_narrow( obj );

  client->deposit( 700 );
  client->withdraw( 250 );
  cout << "Balance is " << client->balance() << endl;

  return 0;
}
```

After completing ORB and BOA initialization the client uses `bind()` to bind to an object with repository id `IDL:Account:1.0` that is running in the process that owns port 8888 on the same machine. Lines 14–16 check if the bind failed. Everything else is the same as in section 3.3.3. Compile:

```
mico-c++ -I. -c account.cc -o account.o
mico-c++ -I. -c account_server2.cc -o account_server2.o
mico-c++ -I. -c account_client2.cc -o account_client2.o
mico-ld -o server2 account.o account_server2.o -lmico2.2.3
mico-ld -o client2 account.o account_client2.o -lmico2.2.3
```

Start the server like this, telling it to run on port number 8888:

```
./server2 -ORBIIOPAddr inet:localhost:8888
```

Run the client in a different shell without any arguments. It should behave the same way as the client from section 3.3.3.

If a server offers several objects (lets say `A` and `B`) of the same type (i.e., with the same repository id) and a client wants to bind to `A` it needs a means to distinguish objects of the same type. This is accomplished by assigning objects an identifier during creation in the server and specifying this identifier as an extra argument to `bind()` in the client. The identifier is of type `BOA::ReferenceData`, which is a sequence of octets. You can use `ORB::string_to_tag()` and `ORB::tag_to_string()` to convert a string into such an identifier and vice versa. Here are the changes to the server code:

```
#include "account.h"

class Account_impl : virtual public Account_skel {
public:
 Account_impl (const CORBA::BOA::ReferenceData &refdata)
    : Account_skel (refdata)
  {
    _current_balance = 0;
  }
  // remaining parts unchanged
};

int main( int argc, char *argv[] )
{
  ...
  CORBA::BOA::ReferenceData_var id
    = CORBA::ORB::string_to_tag ("foo");
  Account_impl* server = new Account_impl (id);
  ...
}
```

Changes to the client:

```
#include "account.h"

int main( int argc, char *argv[] )
{
  ...
```

```
  CORBA::BOA::ReferenceData_var id
    = CORBA::ORB::string_to_tag ("foo");
  CORBA::Object_var obj
    = orb->bind ("IDL:Account:1.0", id, "inet:localhost:8888");
  ...
}
```

To avoid hardcoding the address of the server into the client you can leave out the second argument to `bind()` and specify a list of addresses to try using the `-ORBBindAddr` command line option. For example

```
./client -ORBBindAddr local: -ORBBindAddr inet:localhost:8888
```

will make `bind()` try to bind to an account object in the same process and if that fails it will try to bind to an account object running in the server than owns port 8888 on the same machine. Note that addresses specified using `-ORBBindAddr` are only taken into account if you to not specify an explicit address.

The `demo/account2` directory contains an example that uses the Mico binder.

Chapter 4

Implementation Overview

This chapter gives you an overview of how Mico implements the CORBA 2 specification, the implementation components it consists of and how those components are being used.

A CORBA 2 implementation consists of the following logical components:

- the *Object Request Broker (ORB)* provides for object location and method invocation.
- the *interface repository* stores runtime type information.
- one or more *object adapters* which form the interface between object implementations and the ORB; at least the *Basic Object Adapter (BOA)* has to be provided, part of which is the *implementation repository* that stores information about how to activate object implementations.
- the *IDL compiler* generates client stubs, server skeletons and marshalling code from a CORBA IDL according to the supported language mappings.

Each of these logical components has to be mapped to one or more implementation components, which are described in the next sections.

4.1 ORB

The ORB is implemented as a library (`libmico2.2.3.a`) that is linked into each Mico application.

4.1.1 ORB Initialization

Every Mico application has to call the ORB initialization function `ORB_init()` before using Mico functionality.

```
int main (int argc, char *argv[])
{
   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
   ...
}
```

That way the ORB has access to the applications command line arguments. After evaluating them the ORB removes the command line options it understands so the application doesn't have to care about them. You can also put ORB command line arguments into a file called `.micorc` in your home directory. Arguments given on the command line override settings from `.micorc`. Here is a description of all ORB specific command line arguments:

`-ORBNoIIOPServer`
: Do not activate the IIOP server. The IIOP server enables other processes to invoke methods on objects in this process using the *Internet Inter ORB Protocol (IIOP)*. If for some reason you do not want other processes to be able to invoke objects in this process you can use this option. Default is to activate the IIOP server.

`-ORBNoIIOPProxy`
: Do not activate the IIOP proxy. The IIOP proxy enables this process to invoke methods on objects in other processes using IIOP. If you do not want or need this you can use this option. Default is to activate the IIOP proxy.

`-ORBIIOPAddr <address>`
: Set the address the IIOP server should run on. See section 3.3.3 for details on addresses. If you do not specify this option the IIOP server will choose an unused address. This option can be used more than once to make the server listen on several addresses (e.g., a `unix:` and an `inet:` address).

`-ORBId <ORB identifier>`
: Specify the ORB identifier, `mico-local-orb` is currently the only supported ORB identifier. This option is intended for programs that needed access to different CORBA implementations in the same process. In this case the option `-ORBId` is used to select one of the CORBA implementations.

`-ORBImplRepoIOR <impl repository IOR>`
: Specify a stringified object reference[1] for the implementation repository the ORB should use.

`-ORBImplRepoAddr <impl repository address>`
: Specify the address of a process that runs an implementation repository. The ORB will then try to bind to an implementation repository object using the given address. See 3.3.3 for details on addresses and the binder. If the bind fails or if you did neither specify `-ORBImplRepoAddr` nor `-ORBImpRepoIOR` the ORB will run a local implementation repository.

`-ORBIfaceRepoIOR <interface repository IOR>`
: The same as `-ORBImplRepoIOR` but for the interface repository.

`-ORBIfaceRepoAddr <interface repository address>`
: The same as `-ORBImplRepoAddr` but for the interface repository.

[1] IOR means *Interoperable Object Reference*

`-ORBNamingIOR <naming service IOR>`
: The same as `-ORBImplRepoIOR` but for the naming service.

`-ORBNamingAddr <naming address>`
: The same as `-ORBImplRepoAddr` but for the naming service.

`-ORBDebugLevel <level>`
: Specify the debug level. `<level>` is a non–negative integer with greater values giving more debug output on `cerr`.

`-ORBBindAddr <address>`
: Specify an address which `bind(const char *repoid)` should try to bind to. This option can be used more than once to specify multiple addresses.

`-ORBConfFile <rcfile>`
: Specifies the file from which to read additional command line options (defaults to `~/.micorc`).

`-ORBNoCodeSets`
: Do not add code set information to object references. Since code set conversion is a CORBA 2.1 feature this option may be needed to talk to ORBs which are not CORBA 2.1 compliant. Furthermore it may gain some extra speed.

`-ORBNativeCS <pattern>`
: Specifies the code set the application uses for characters and strings. `<pattern>` is a shell–like pattern that must match the `description` field of a code set in the OSF code set registry[2]. For example the pattern `*8859-1*` will make the ORB use the code set ISO–8859–1 (Latin 1) as the native char code set, which is the default if you do not specify this option. The ORB uses this information to automatically convert characters and strings when talking to an application that uses a different code set.

`-ORBNativeWCS <pattern>`
: Similar to `-ORBNativeWCS`, but specifies the code set the application uses to wide characters and wide strings. Defaults to UTF-16, a 16 bit encoding of Unicode.

4.1.2 Obtaining Initial References

The ORB offers two functions for obtaining object references for the interface repository, the implementation repository, and the naming service. Here is an example that shows how to obtain a reference for the interface repository using `resolve_initial_references()`:

```
int main (int argc, char *argv[])
{
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
```

[2]See files `admin/code_set_registry.txt` and `admin/mico_code_set_registry.txt` in the Mico source tree.

```
  ...
  CORBA::Object_var obj =
    orb->resolve_initial_references ("InterfaceRepository");
  CORBA::Respository_var repo = CORBA::Repository::_narrow (obj);
  ...
}
```

If you specify the interface repository by using the ORB command line option `-ORBIfaceRepoAddr` or `-ORBIfaceRepoIOR`, the reference returned from `resolve_initial_references()` will be the one you specified. Otherwise the ORB will run a local interface repository and you will get a reference to this one.

Obtaining a reference to the implementation repository (`"ImplementationRepository"`) and the naming service (`"NameService"`) works the same way as for the interface repository.

There is another method called `list_initial_reference()` that returns a list of names which can be used as arguments for `resolve_initial_references()`. Here is how to use it:

```
int main (int argc, char *argv[])
{
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
  ...
  CORBA::ORB::ObjectIdList_var ids = orb->list_initial_references ();
  for (int i = 0; i < ids->length(); ++i)
    cout << ids[i] << endl;
  ...
}
```

4.2 Interface Repository

The interface repository is implemented by a separate program (`ird`). The idea is to run one instance of the program and make all MICO applications use the same interface repository. As has been mentioned in section 4.1.2 the command line option `-ORBIfaceRepoAddr` can be used to tell a MICO application which interface repository to use. But where to get the address of the `ird` program from? The solution is to tell `ird` an address it should bind to by using the `-ORBIIOPAddr`. Here is an example of how to run `ird`:

```
ird -ORBIIOPAddr inet:<ird-host-name>:8888
```

where `<ird-host-name>` should be replaced by the name of the host `ird` is executed. Afterwards you can run MICO applications this way:

```
some_mico_application -ORBIfaceRepoAddr inet:<ird-host-name>:8888
```

To avoid typing in such long command lines you can put the option into the file `.micorc` in your home directory:

```
echo -ORBIfaceRepoAddr inet:<ird-host-name>:8888 > ~/.micorc
```

Now you can just type:

```
some_mico_application
```

and `some_mico_application` will still use the `ird`'s interface repository.
`ird` can be controlled by the following command line arguments:

`--help`
: Show a list of all supported command line arguments and exit.

`--db <database file>`
: Specifies the file name where `ird` should save the contents of the interface repository when exiting[3]. When `ird` is restarted afterwards it will read the file given by the `--db` option to restore the contents of the interface repository. Notice that the contents of this database file is just plain ASCII representing a CORBA IDL specification.

4.3 BOA

The *Basic Object Adapter (BOA)* is the only object adapter specified by CORBA 2. One of its main features is the ability to *activate* object implementations[4] when their service is requested by a client. Using the *implementation repository* the BOA decides how an object implementation has to be activated[5].

To fulfill these requirements of the CORBA 2 specification the BOA is implemented partially by a library (`libmico2.2.3.a`) and partially by a separate program (`micod`) called the *BOA daemon*.

4.3.1 BOA Initialization

Similar to the ORB initialization described in section 4.1.1 the BOA has to be initialized like this:

```
int main (int argc, char *argv[])
{
   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
   CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");
   ...
}
```

[3] `ird` is terminated by pressing `ctrl-c` or by sending it the `SIGTERM` signal

[4] which basically means running a program that implements an object

[5] i.e. which program has to be run with which options and what activation policy has to be used for the implementation

That way it has access to the applications command line arguments. After evaluating them the BOA will remove the command line options it knows about from `argv`. As for the ORB you can put BOA specific command line options into a file called `.micorc` in your home directory. Arguments given on the command line override settings from `.micorc`. Here is a list of command line options the BOA understands:

`-OAId <BOA identifier>`
: Specify the BOA identifier, `mico-local-boa` is the only currently supported BOA identifier.

`-OAImplName <name of the object implementation>`
: Tell a server its implementation name. This option must be used when launching a persistent server that should register with the BOA daemon.

`-OARestoreIOR <IOR to restore>`
: This options is part of the interface between the BOA daemon and an object implementation. Do not use this option!

`-OARemoteIOR <remote BOA IOR>`
: This options is part of the interface between the BOA daemon and an object implementation. Do not use this option!

`-OARemoteAddr <remote BOA address>`
: This option tells an object implementation the address of the BOA daemon. You should use this option only when starting persistent servers that should register with the BOA daemon. See section 4.3.4 for details.

4.3.2 BOA Daemon

The BOA daemon (`micod`) is the part of the basic object adapter that activates object implementations when their service is requested. Moreover `micod` contains the implementation repository. To make all MICO applications use a single implementation repository you have to take similar actions as for the interface repository as described in section 4.2. That is you have to tell `micod` an address to bind to using the `-ORBIIOPAddr` option and tell all MICO applications this address by using the `-ORBImplRepoAddr` option. For example:

```
micod -ORBIIOPAddr inet:<micod-host-name>:9999
```

Now you can run all MICO applications like this:

```
some_mico_application -ORBImplRepoAddr inet:<micod-host-name>:9999
```

or you can put the option into `.micorc` and run `some_mico_application` without arguments.
`micod` understands the following command line arguments:

`--help`

Show a list of all supported command line arguments and exit.

--db <database file>

Specifies the file name where `micod` should save the contents of the implementation repository when exiting[6]. When `micod` is restarted afterwards it will read the file given by the `--db` option to restore the contents of the implementation repository.

4.3.3 Implementation Repository

The implementation repository is the place where information about an object implementation (also known as *server*) is stored. The CORBA 2 specification gives you only an idea what the implementation repository is for, but does not specify the interface to it. So the design of the implementation repository is MICO specific. Here is the IDL for MICO's implementation repository:

```
module CORBA {
  /*
   * Implementation Repository Entry
   */
  interface ImplementationDef {

    enum ActivationMode {
      ActivateShared, ActivateUnshared,
      ActivatePerMethod,
      ActivatePersistent,
      ActivateLibrary
    };

    typedef sequence<string> RepoIdList;

    attribute ActivationMode mode;
    attribute RepoIdList repoids;
    readonly attribute string name;
    attribute string command;
  };

  /*
   * Implementation Repository
   */
  interface ImplRepository {
    typedef sequence<ImplementationDef> ImplDefSeq;

    ImplementationDef create (...);
    void destroy (in ImplementationDef impl_def);
    ImplDefSeq find_by_name (in string name);
    ImplDefSeq find_by_repoid (in string repoid);
    ImplDefSeq find_all ();
  };
};
```

[6] `micod` is terminated by pressing `ctrl-c` or by sending it the `SIGTERM` signal

Interface `ImplRepository` defined in lines 25–33 is the implementation repository itself. It contains methods for creating, destroying and finding entries. An implementation repository entry is defined by interface `ImplementationDef` in lines 5–20. There is exactly one entry for each server which contains

- name
- activation mode
- shell command or loadable module path
- list of repository ids

for the sever. The name uniquely identifies the server. The activation mode tells the BOA whether the server should be activated once (*shared server*), once for each object instance (*unshared server*), once for each method invocation (*per method server*), or not at all (*persistent server*). See section 4.3.4 for details on activation modes. The shell command is executed by the BOA whenever the server has to be (re)started. Activation mode *library* is used for loading servers into the same process as the client during runtime. Instead of a shell command you have to specify the path of the loadable server module for library activation mode. Finally there is a repository id for each IDL interface implemented by the server. See section 3.3.3 for details on repository ids.

If you have written a server that should be activated by the BOA daemon when its service is requested you have to create an entry for that server. This can be accomplished by using the program `imr`. `imr` can be used to list all entries in the implementation repository, to show detailed information for one entry, to create a new entry, and to delete an entry.

The implementation repository is selected by the `-ORBImplRepoAddr` or `-ORBImplRepoIOR` options, which you usually put into your `.micorc` file.

Listing All Entries

Just issue the following command:

```
imr list
```

and you will get a listing of the names of all entries in the implementation repository.

Details For One Entry

```
imr info <name>
```

will show you detailed information for the entry named `<name>`.

Creating New Entries

```
imr create <name> <mode> <command> <repoid1> <repoid2> ...
```

will create a new entry with name `<name>`. `<mode>` is one of

- `persistent`
- `shared`
- `unshared`
- `permethod`
- `library`
- `poa`

`<command>` is the shell command that should be used to start the server. Note that all paths have to be absolute since `micod`'s current directory is probably different from your current directory. Furthermore you have to make sure that the server is located on the same machine as `micod`, otherwise you have to use `rsh`; see below for examples. `<repoid1>`, `<repoid2>` and so on are the repository ids for the IDL interfaces implemented by the server.

Deleting Entries

```
imr delete <name>
```

will delete the entry named `<name>`.

Forcing Activation of an Implementation

Registering an implementation in the implementation repository does not automatically activate the implementation. Usually a non–persistent implementation is only activated by the BOA daemon when its service is requested by a client. But sometimes you have to force activation of an implementation, for instance to make the implementation register itself with a naming service.

```
imr activate <name> [<micod-address>]
```

will activate the implementation named `<name>`. To do this `imr` needs to know the address of the BOA daemon. Usually this is the same address as for the implementation repository and you do not need to specify `<micod-address>`. Only if the BOA daemon is bound to an address different from the implementation repository address and different from the addresses specified using the `-ORBBindAddr` option you have to specify `<micod-address>` as a command line option to `imr`.

Examples

Assume we want to register the account server `account_server2` from section 3.3.3 as a shared server. Furthermore assume that neither `micod` nor `ird` have been started yet, so we have to get them running first. Assuming the hostname is `zirkon`, you have to do the following:

```
# create .micorc (only do that once)
echo -ORBIfaceRepoAddr inet:zirkon:9000 > ~/.micorc
echo -ORBImplRepoAddr inet:zirkon:9001 >> ~/.micorc

# run ird
ird -ORBIIOPAddr inet:zirkon:9000

# run micod in a different shell
micod -ORBIIOPAddr inet:zirkon:9001
```

Now we are prepared to create the implementation repository entry for `account_server2`. Recall that this server implemented the interface `Account` whose repository id is `IDL:Account:1.0`. Assuming `account_server2` has been copied to `/usr/bin` you can create the implementation repository entry using the following command:

```
imr create Account shared /usr/bin/account_server2 IDL:Account:1.0
```

If `account_server2` is located on host `diamant` (i.e., *not* on `zirkon`) you have to use the `rsh` command. This requires of course that you have entries in your `.rhosts` file that allow `micod` to execute programs on `diamant`. Here is the command to create the implementation repository entry:

```
imr create Account shared "rsh diamant /usr/bin/account_server2" \
  IDL:Account:1.0
```

Now you should change `account_client2.cc` to bind to the address of `micod`. Note that you no longer need to know the address of the account server `account_server2`, you only need to know the address of `micod`. Here is the part of `account_client2.cc` that has to be changed:

```
// account_client2.cc
...
  CORBA::Object_var obj =
    orb->bind ("IDL:Account:1.0", "inet:zirkon:9001");
...
```

Running the recompiled client will automatically activate `account_server2`.

Creating an entry for a loadable module (library activation mode) looks like this if `/usr/local/lib/module.so` is the path to the module:

```
imr create Account library /usr/local/lib/module.so IDL:Account:1.0
```

Note that you have to make sure that a loadable module and a client that wants to make use of the module reside on the same machine.

4.3.4 Activation Modes

As mentioned in the previous section the BOA supports several activation modes. Using them is not simply a matter of creating an implementation repository entry, instead an object implementation has to use special BOA functionality according to the selected activation mode. This section gives you some details on this topic.

Activation Mode *Shared*

Shared servers can serve any number of object instances, which is probably the most widely used approach. The account server from section 3.3.3 is an example for a shared server. Lets look at the code again:

```
 1: // file account_server2.cc
 2:
 3: #include "account.h"
 4:
 5: class Account_impl : virtual public Account_skel
 6: {
 7:   // unchanged, see section "MICO Application"
 8:   // ...
 9: };
10:
11:
12: int main( int argc, char *argv[] )
13: {
14:   // ORB initialization
15:   CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
16:   CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );
17:
18:   Account_impl* server = new Account_impl;
19:
20:   boa->impl_is_ready( CORBA::ImplementationDef::_nil() );
21:   orb->run ();
22:   CORBA::release( server );
23:   return 0;
24: }
```

After creating the implementation repository entry for the account server using the `imr` utility the account server stays inactive until the account client wants to bind to an object with repository id `IDL:Account:1.0`. The BOA daemon recognizes that there are no active account objects and consults the implementation repository for servers that implement objects with repository id `IDL:Account:1.0`. It will find the account server and run it. The account server in turn creates an account object in line 18, which will be announced to the BOA daemon. The server uses `impl_is_ready()` to tell the BOA daemon that it has completed initialization and is prepared to receive method invocations. The BOA daemon in turn finds the newly created account object and answers the bind request from the client with it. Finally `run()` is called on the ORB to start processing events.

`run()` will wait for requests and serve them as they arrive until the `deactivate_impl()` method is called, which deactivates the server. Calling the ORB method `shutdown()` will make `run()` return and the account server will exit. If method invocations arrive after the server has exited the BOA daemon will restart the server. See section 4.3.5 for details on restaring servers.

There are many reasons for calling `deactivate_impl()`. For example we could augment the account objects interface by a management interface that offers a method `exit()` that will shut down the account server[7]:

```
// account.idl
interface Account {
  ...
  void exit ();
};
```

The implementation of the `exit()` method would look like this:

```
// account.idl
class Account_impl : virtual public Account_skel {
  ...
public:
  ...
  virtual void exit ()
  {
    CORBA::BOA_var boa = _boa();
    CORBA::ORB_var orb = _orb();
    boa->deactivate_impl (CORBA::ImplementationDef::_nil());
    orb->shutdown (TRUE);
  }
};
```

Note that we passed a NIL `ImplementationDef` to `deactivate_impl()` as well as to `impl_is_ready()`. Usually the implementation repository has to be searched to find the entry for the server and pass this one. When passing NIL the entry will be searched by the BOA. `shutdown()` has a `boolean wait` parameter which controls whether the ORB should immediately stop processing events (`wait=FALSE`) or wait until all pending requests have completed (`wait=TRUE`).

Activation Mode *Persistent*

Persistent servers are just like shared servers, except that the BOA daemon does not activate them. Instead they have to be started by means outside of the BOA, e.g. by a system administrator or a shell script. The code of a persistent server looks exactly like that of a a shared server. But note that once `deactivate_impl()` and `shutdown()` are called the server will *not* be restarted by the BOA daemon.

That means persistent servers do not need a running BOA daemon. Instead clients can connect directly to the object implementation, giving you better performance. See section 3.3.3 for an example. However, there is a reason to have even persistent servers register with the BOA daemon: you can do a `bind()` using the address of the BOA daemon, that

[7] Usually one would define a new interface `ManagedObject` that contains the management operations and derive `Account` from `ManagedObject`. We don't do this here for ease of exposition.

is you do not need to know the address of the persistent server. Making a persistent server register with the BOA daemon is done like this:

```
some_server -OARemoteAddr <micod-address> -ORBImplRepoAddr <micod-address> \
  -OAImplName <impl-name>
```

where <micod-address> is the address micod is bound to[8]. This is usually the same address you used as an argument to -ORBIIOPAddr when starting micod. See section 3.3.3 for details on addresses, sections 4.1.1 and 4.3.1 for details on command line arguments. <impl-name> is the name of the entry in the implementation repository the corresponds to the server.

Activation Mode *Unshared*

Unshared servers are similar to shared servers. The difference is that each instance of an unshared server can only serve *one* object instance. That is for N objects you need N running instances of an unshared server.

Furthermore you cannot use impl_is_ready() and deactivate_impl() but have to use obj_is_ready() and deactivate_obj() instead. Here is the main() function of an unshared account server:

```
// file account_server2.cc

#include "account.h"

class Account_impl : virtual public Account_skel
{
  // unchanged, see section "MICO Application"
  // ...
};


int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  Account_impl* server = new Account_impl;

  boa->obj_is_ready (server, CORBA::ImplementationDef::_nil());
  orb->run ();
  CORBA::release( server );
  return 0;
}
```

The exit() method would look like this in an unshared server:

[8]The -ORBImplRepoAddr option is usually already in your .micorc file, so you do not have to specify it.

```
// account.idl
class Account_impl : virtual public Account_skel {
  ...
public:
  ...
  virtual void exit ()
  {
    CORBA::BOA_var boa = _boa();
    CORBA::ORB_var orb = _orb();
    boa->deactivate_obj (this);
    orb->shutdown (TRUE);
  }
};
```

Although an unshared server instance can only *serve* one object instance it can *create* more than one object instance. Imagine for instance a bank object

```
// bank.idl
interface Bank {
  Account create ();
  void destroy (in Account account);
};
```

that can create new account objects and destroy account objects that are no longer needed[9]. The implementation of the `create()` method in an unshared server would look like this:

```
 1:  // bank_server.cc
 2:  class Bank_impl : virtual public Bank_skel {
 3:    ...
 4:  public:
 5:    ...
 6:    virtual Account_ptr create ()
 7:    {
 8:      Account_ptr account = new Account_impl;
 9:
10:      CORBA::BOA_var boa = _boa();
11:      boa->deactivate_obj (account);
12:
13:      return Account::_duplicate (account);
14:    }
15:  };
```

Note that line 11 calls `deactivate_obj()` on the newly created object[10]. This will tell the BOA daemon that you are not going to serve this object, instead a new server instance has to be activated for serving the newly created account object. For this to work you must of course implement saving and restoring for your objects as described in section 4.3.5.

If you need access to the newly created account object from within the server where it was first created you need to take special actions. The reason for this is that the

[9] Such a design pattern is called a *factory*.

[10] If you delete lines 10 and 11 you will get the code for `create()` in a shared or persistent server.

created account object is initially an account object implementation (`Account_impl`), but in order to access the moved account object in the other server you need an account stub (`Account_stub`). Here is how to create this stub:

```
// bank_server.cc
class Bank_impl : virtual public Bank_skel {
  ...
public:
  ...
  virtual Account_ptr create ()
  {
    CORBA::BOA_var boa = _boa();
    CORBA::ORB_var orb = _orb();

    Account_ptr account = new Account_impl;
    boa->deactivate_obj (account);

    // turn 'account' into a stub
    CORBA::String_var ref = orb->object_to_string (account);
    CORBA::release (account);
    CORBA::Object_var obj = orb->string_to_object (ref);
    account = Account::_narrow (obj);

    // now you can invoke methods on (the remote) 'account'
    account->deposit (100);

    return Account::_duplicate (account);
  }
};
```

The `demo/account3` directory contains a complete example for an unshared server that creates more than one object.

Activation Mode *Per Method*

Per Method servers are similar to unshared servers, except that a new server instance is launched for each method invocation. The code for a per method server looks the same as for an unshared server. But note that `run()` will return after the first method invocation, whereas in an unshared server `run()` will not return until you call `shutdown()`.

Activation Mode *Library*

All activation modes discussed up until now assume client and server are different programs that run in separate processes. This approach has the advantage that client and server can be bound to each other dynamically during runtime. The drawback is the overhead for doing method invocations across process boundaries using some kind of IPC. The activation mode *library* eliminates this drawback while still allowing runtime binding. This is achieved by loading an object implementation (called a *module* from now on) into the running client. Invoking methods on an object loaded this way is as fast as a C++ method invocation.

A client that wants to use this feature does not differ from other clients, only the loadable module requires special code and you have to create a special entry in the implementation repository. To give you an example we want to change the bank account example from section 3.3.3 to make use of dynamic loading. The only change in the client is the address specified in the call to `bind()`: we have to use `"local:"` instead of `"inet:localhost:8888"`, because we want to bind to the dynamically loaded object running in the same process:

```
// file account_client2.cc

#include "account.h"


int main( int argc, char *argv[] )
{
  // ORB initialization
  CORBA::ORB_var orb = CORBA::ORB_init( argc, argv, "mico-local-orb" );
  CORBA::BOA_var boa = orb->BOA_init( argc, argv, "mico-local-boa" );

  CORBA::Object_var obj
    = orb->bind ("IDL:Account:1.0", "local:");
  if (CORBA::is_nil (obj)) {
     // no such object found ...
  }
  Account_var client = Account::_narrow( obj );

  client->deposit( 700 );
  client->withdraw( 250 );
  cout << "Balance is " << client->balance() << endl;

  return 0;
}
```

Here is the code for the loadable module:

```
// file module.cc

#include "account.h"
#include <mico/template_impl.h>

class Account_impl : virtual public Account_skel
{
  // unchanged, see section "MICO Application"
  // ...
};

static Account_ptr server = Account::_nil();

extern "C" CORBA::Boolean
mico_module_init (const char *version)
{
  if (strcmp (version, MICO_VERSION))
    return FALSE;
```

```
18:   server = new Account_impl;
19:   return TRUE;
20: }
21:
22: extern "C" void
23: mico_module_exit ()
24: {
25:   CORBA::release (server);
26: }
```

Lines 13–20 define a function `mico_module_init()` that is called when the module is loaded into the running client. Note that this function must be declared as `extern "C"` to avoid C++ name mangling. The `version` argument to `mico_module_init()` is a string specifying the Mico–version of the client the module is loaded into. Lines 16 and 17 check if this version is the same as the Mico–version the module was compiled with and make module initialization fail by returning `FALSE` if they differ. Otherwise a new account object is created and `TRUE` is returned indicating successful module initialization. Note that `mico_module_init()` must not perform ORB and BOA initialization since the client the module is loaded into did this already. The function `mico_module_exit()` is called just before the module is unloaded from the client and should release all allocated resources: in our example the account object created in `mico_module_init()`. `mico_module_exit()` is only called if `mico_module_init()` returned `TRUE`. Modules have to be compiled as a shared library, see section 4.6 for details and an example.

Although communication does not go through the BOA daemon when using loadable modules you need a running `micod` because you have to create an implementation repository entry for the module. See section 4.3.3 for details. The directory `demo/shlib` contains a complete example.

There is currently one problem with loadable modules: throwing exceptions from a loadable module into non–loadable module code results in a segmentation fault. This is not a bug in Mico but in the GNU–C++ compiler and/or dynamic loader.

4.3.5 Making Objects Persistent

In the last section we saw two cases where an object had to be "moved" between two different instances of a server[11]:

- if an unshared or per method server creates a second object it has to be moved to a new server instance.
- if a server terminates and is restarted later all the objects of the terminated server have to be moved to the restarted server.

In all these cases the state of the moved object has to be saved before and restored after moving. Because the BOA has no information about the internal state of an object

[11] Note that the CORBA 2 specification only gives you some vague idea of object persistence but omits any implementation details. That is why everything explained in this section is Mico–specific and will not work with other CORBA implementations.

the user has to provide code for saving and restoring. However, the BOA offers you some support methods.

Saving is done in the `_save_object()` method of the object implementation. If you do not provide this method for an object, `_save_object()` from the base class will be used, which will cause the object to be treated as transient (i.e., it will not be restored later). Let us again consider the account example. The internal state of an account object consists of the current balance. Here is how to save the state:

```
// account_server3.cc

#include "account.h"
#include <iostream.h>
#include <fstream.h>

class Account_impl : virtual public Account_skel {
  CORBA::Long _current_balance;
public:
  ...
  virtual CORBA::Boolean _save_object ()
  {
     ofstream out (_ident());
     out << _current_balance;
     return TRUE;
  }
};
```

Pretty simple, eh? We just open a file and write the balance into it. The only noteworthy thing is the file name, which is obtained by using the `_ident()` method. The returned string is guaranteed to be unique among all objects managed by a single BOA daemon. If you use multiple BOA daemons or use persistent servers that do not register with the BOA you have to make sure no name clashes occur. One way to do this is to create a new directory where all the files are created, in our example `/tmp/account/` would be appropriate. Another way to distinguish different instances (objects) of on interface (class) is to use `BOA::ReferenceData`. See `demo/account2` for an example.

Restoring the state takes a bit more code. You need to subclass the abstract baseclass `CORBA::BOAObjectRestorer` providing an implementation for the `restore()` method:

```
// account_server3.cc

class AccountLoader : public CORBA::BOAObjectRestorer {
public:
  CORBA::Boolean restore (CORBA::Object_ptr obj)
  {
     if (!strcmp (obj->_repoid(), "IDL:Account:1.0")) {
       new Account_impl (obj);
       return TRUE;
     }
     // dont know about such objects
     return FALSE;
  }
};
```

`restore()` receives an object reference for the object that has to be restored. We use the `_repoid()` method to find out the repository id[12] of the object to be restored. If it is equal to the repository id of account objects (`"IDL:Account:1.0"`) we can go on with restoring, otherwise we just return `FALSE` indicating that we cannot restore the object.

Restoring the object is now just a matter of calling a special `Account_impl` constructor which we still have to define:

```
 1:  // account_server3.cc
 2:
 3:  class Account_impl : virtual public Account_skel {
 4:    CORBA::Long _current_balance;
 5:  public:
 6:    ...
 7:    Account_impl (CORBA::Object_ptr obj)
 8:      : Account_skel (obj)
 9:    {
10:      ifstream in (obj->_ident());
11:      in >> _current_balance;
12:    }
13:  };
```

The constructor is basically the counterpart to `_save_object()`. It uses `_ident()` to obtain the identification string of the object to be restored, opens the associated file and reads in the current balance. Note the invocation of the base class constructor in line 8, which is very important. If you forget this line the code will still compile but will give you strange results, because the default `Account_skel` constructor will be used, which is an error.

Note that we have omitted error handling for the ease of exposition. Usually one would check if the file exists and its contents are valid. If an error is detected you should make `AccountLoader::restore()` return `FALSE`[13].

Now what is left to do is to create an instance of the `AccountLoader` class. Note that you have to create at least one such instance *before* you do ORB and BOA initialization, because restoring can already occur during BOA initialization. Of course you can create serveral different `BOAObjectRestorer` subclasses each of which handles special kinds of objects. When an object has to be restored the `restore()` methods of the existing restorer objects are called until eventually one returns `TRUE`. Note that you should not create new objects if any objects are being restored, because otherwise you would get an infinitely growing number of objects over time. The BOA method `restoring()` returns `TRUE` if objects are being restored, `FALSE` otherwise. Here is the `main()` function:

```
// account_server3.cc

int main (int argc, char *argv[])
{
  // create loader *before* BOA initialization
  AccountLoader loader;

```

[12] See section 3.3.3 for details on repository ids.

[13] For instance by throwing an exception that is caught in `restore()`.

```
  CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
  CORBA::BOA_var boa = orb->BOA_init (argc, argv, "mico-local-boa");

  if (!boa->restoring()) {
    // create new objects only if not restoring
    new Account_impl;
  }
  boa->impl_is_ready (CORBA::ImplementationDef::_nil());
  orb->run ();
  return 0;
}
```

In an unshared or per method server you would call

```
boa->obj_is_ready (CORBA::Object::_nil(),
                   CORBA::ImplementationDef::_nil());
```

instead of `impl_is_ready()`. The sources for a complete example can be found in `demo/account2`.

Sometimes it is handy to know when saving of objects can occur. But you cannot rely on this being the only occurences of object saving:

1. Just before a server is exiting all the objects that have not been released are saved. If you do not want an object to be saved you must make its `_save_object()` method return `FALSE` or do not provide a `_save_object()` method at all. The object will then be treated as transient (i.e., it will not outlive the process it was created in).

2. When you call `deactivate_obj()` on an object in an unshared or per method server saving is done during the call to `deactivate_obj()`. Objects saved this way will *not* be saved again at server exit according to 1.

3. When you call `deactivate_impl()` in a shared or persistent server saving of all currently activate objects is done during the call to `deactivate_impl()`. Objects saved this way will *not* be saved again at server exit according to 1.

4. When you migrate an object saving of it is done during the call to `change_implementation()`, see section 4.3.6 for details. Objects saved this way will *not* be saved again at server exit according to 1.

Note that it is quite likely that invocations on objects will occure after a call to `deactivate_obj()`, `deactivate_impl()`, or `change_implementation()` because the server has to execute all (buffered) invocations that arrived up until your call to one of the above mentioned methods. So your code must be prepared to handle this.

Although the actual code for saving and restoring the state of an account object are two–liners each real world applications often require complex code for making objects persistent. Therefore the OMG has specified the *Persistent Object Service (POS)*, an implementation of which is not yet provided by MICO.

4.3.6 Migrating Objects

Up until now we described how objects are moved between different *instances* of the same server. Here we explain how to move objects between two completely different servers. This is for example useful if a server has to be replaced by a new version without interrupting usual business.

Recall that we augmented the account object by a management interface in section 4.3.4. The management interface offered a method `exit()` that terminates the server when invoked. Now let us add a method `migrate()` that migrates an account object to a new server. The new server is specified through an implementation repository entry.

```
// account.idl
interface Account {
  ...
  void migrate (in CORBA::ImplementationDef destination);
};
```

Here is the implementation of the `migrate()` method:

```
 1:  #include "account.h"
 2:
 3:  class Account_impl : virtual public Account_skel {
 4:    ...
 5:  public:
 6:    ...
 7:    virtual void migrate (CORBA::ImplementationDef_ptr dest)
 8:    {
 9:       CORBA::BOA_var boa = _boa();
10:       boa->change_implementation (this, dest);
11:    }
12:  };
```

The `change_implementation()` in line 10 does the whole job. It will save the object's state as described in section 4.3.4 and tell the BOA daemon to use the new implementation from now on. See `demo/account4` for an example.

The current version of MICO can only perform the migration when the destination implementation is not currently active, which means that:

- you cannot migrate an object to a persistent server
- you cannot migrate an object to a shared server that is already running

This limitation will be removed in a future version of MICO.

4.4 POA

The Basic Object Adapter provides a bare minimum of functionality to server applications. As a consequence, many ORBs added custom extensions to the BOA to support more complex demands upon an object adapter, making server implementations incompatible among different ORB vendors. In CORBA 2.2, the new *Portable Object Adapter* was introduced. It provides a much-extended interface that addresses many needs that were wished for, but not available with the original BOA specification. POA features include:

- Support for transparent activation of objects. Servers can export object references for not-yet-active servants that will be incarnated on demand.
- Allow a single servant to support many object identities.
- Allow many POAs in a single server, each governed by its own set of *policies*.
- Delegate requests for non-existent servants either to a default servant, or ask a servant manager for an appropriate servant.

These features, make the POA much more powerful than the BOA and should fulfill most server applications' needs. As an example, object references for some million entries in a database can be generated, which are all implemented by a single default servant.

4.4.1 Architecture

The general idea is to have each server contain a hierarchy of POAs. Only the *Root POA* is created by default; a reference to the Root POA is obtained using the `resolve_initial_references()` operation on the ORB. New POAs can be created as the child of an existing POA, each with its own set of policies.

Each POA maintains an *Active Object Map* that maps all objects that have been activated in the POA to a servant. For each incoming request, the POA looks up the object reference in the Active Object Map and tries to find the responsible servant. If none is found, the request is either delegated to a default servant, or a servant manager is invoked to activate or locate an appropriate servant.

Associated with each POA is a *POA Manager* object. A POA Manager can control one or many POAs. For each incoming request to an object, the POA Manager's state is checked, which can be one of the following:

Active
: Requests are performed immediately.

Holding
: Incoming requests are queued. This is the initial state of a POA Manager; to perform requests, the POA Manager must be explicitely set to the *Active* state.

Discarding
: Requests are discarded. Clients receive a `TRANSIENT` exception.

Inactive
: This is the "final" state of a POA Manager, which is entered prior to destruction of the associated POAs. Clients receive an `OBJ_ADAPTER` exception.

Before continuing, we should more precisely define a few terms that have already been freely used.

Object Reference

On the client side, an object reference encapsulates the identity of a distinct abstract object. On the server side, an object reference is composed of the POA identity in which the object is realized, and a *Object Id* that uniquely identifies the object within the POA.

Object Id

An Object Id is an opaque sequence of octets. Object Ids can be either system generated (the POA assigns a unique Id upon object activation), or user generated (the user must provide an Id upon object activation). The object's Object Id cannot be changed through the object's lifetime.

In many cases, object references and Object Id can be used synonymously, since an object reference is just an Object Id with opaque POA-added "internal" information.

Servant

A servant provides the implementation for one or more object references. In the C++ language mapping, a servant is an instance of a C++ class that inherits from `PortableServer::ServantBase`. This is true for dynamic skeleton implementations (DSI), or for classes that inherit from IDL-generated skeletons.

The process of associating a servant with an Object Id is called *activation* and is performed using POA methods. A servant can be activated more than once (to serve many different Object Ids) and can be activated in many POAs. After activation, object references can be obtained using other POA methods.

Servants are *not* objects and do not inherit from `CORBA::Object`. It is illegal to perform operations directly upon a servant – all invocations must be routed through the ORB. Also, memory management of servants is entirely left to the user. POAs keep only a pointer to a servant, so they must not be deleted while being activated.

Server

"Server" refers to a complete process in which servants exist. A server can contain one or more POAs, each of which can provide zero, one or more active servants. Each active servant can then serve one or more object references.

4.4.2 Policies

We have already mentioned the *policies* that control various aspects of POA behaviour. POA policies do not change over the POA's lifetime. When creating a new POA as a child of an existing POA, policies are not inherited from the parent, but instead each POA is assigned a set of default policies if not explicitely defined.

Thread Policy

`ORB_CTRL_MODEL` (default)

Invocations are performed as scheduled by the ORB. Potentially, many upcalls are perfomed simultaneously.

`SINGLE_THREAD_MODEL`
: Invocations are serialized. At most a single upcall is performed at any time.

Non-reentrant servants should only be activated in POAs with the `SINGLE_THREAD_MODEL` policy.

As the current version of MICO is not multithreaded, this policy is not yet evaluated.

Lifespan Policy

`TRANSIENT` (default)
: Objects activated in this POA cannot outlive the server process.

`PERSISTENT`
: Objects can outlive the server process

Id Uniqueness Policy

`UNIQUE_ID` (default)
: Servants can be activated at most once in this POA.

`MULTIPLE_ID`
: Servants can be activated more than once in this POA and can therefore serve more than one object reference.

Id Assignment Policy

`SYSTEM_ID` (default)
: Object Ids are assigned by the POA upon object activation.

`USER_ID`
: Upon activation, each servant must be provided with a unique Id by the user.

Servant Retention Policy

`RETAIN` (default)
: The POA maintains a map of active servants (the Active Object Map).

`NON_RETAIN`
: The POA does not maintain an Active Object Map.

Request Processing Policy

`USE_ACTIVE_OBJECT_MAP_ONLY` (default)
: To process an incoming request, the object reference is looked up in the Active Object Map only. If no active servant serving the reference is found, the request is rejected, and an `OBJECT_NOT_EXIST` exception is returned.

`USE_DEFAULT_SERVANT`
: The object reference is looked up in the Active Object Map first. If no active servant is found to serve the reference, the request is delegated to a default servant.

`USE_SERVANT_MANAGER`
: The object reference is looked up in the Active Object Map first. If no active servant is found to serve the reference, a servant manager is invoked to locate or incarnate an appropriate servant.

Implicit Activation Policy

`IMPLICIT_ACTIVATION`
: If an inactive servant is used in a context that requires the servant to be active, the servant is implicitely activated.

`NO_IMPLICIT_ACTIVATION` (default)
: It is an error to use an inactive servant in a context that requires an active servant.

The Root POA has the `ORB_CTRL_MODEL`, `TRANSIENT`, `UNIQUE_ID`, `SYSTEM_ID`, `RETAIN`, `USE_ACTIVE_OBJECT_MAP_ONLY` and `IMPLICIT_ACTIVATION` policies.

4.4.3 Example

As an example, let's write a simple POA-based server. You can find the full code in the `demo/poa/hello-1` directory in the MICO distribution. Imagine a simple IDL description in the file "hello.idl":

```
interface HelloWorld {
  void hello ();
};
```

The first step is to invoke the IDL to C++ compiler in a way to produce skeleton classes that use the POA:

```
idl --poa --no-boa hello.idl
```

The first option, `--poa`, turns on code generation for POA-based skeletons. The second option, `--no-boa` optionally turns off code generation for the old BOA-based skeletons. Next, we rewrite the server.

```
// file server.cc

#include "hello.h"

class HelloWorld_impl : virtual public POA_HelloWorld
{
  public:
    void hello() { printf ("Hello World!\n"); };
};


int main( int argc, char *argv[] )
{
```

```
14:   CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
15:   CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
16:   PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
17:   PortableServer::POAManager_var mgr = poa->the_POAManager();
18:
19:   HelloWorld_impl * servant = new HelloWorld_impl;
20:
21:   PortableServer::ObjectId_var oid = poa->activate_object (servant);
22:
23:   mgr->activate ();
24:   orb->run();
25:
26:   poa->destroy (TRUE, TRUE);
27:   delete servant;
28:   return 0;
29: }
```

The object implementation does not change much with respect to a BOA-based one, the only difference is that `HelloWorld_impl` does not inherit from the BOA-based skeleton `HelloWorld_skel` any more, but from the POA-based skeleton `POA_HelloWorld`.

In `main()`, we first initialize the ORB, then we obtain a reference to the Root POA (lines 15–16) and to its POA Manager (line 17).

Then, we create an instance of our server object. In line 21, the servant is activated. Since the Root POA has the `SYSTEM_ID` policy, a unique Object Id is generated automatically and returned. At this point, clients can use the MICO binder to connect to the HelloWorld object.

However, client invocations upon the HelloWorld object are not yet processed. The Root POA's POA Manager is created in the holding state, so in line 23, we transition the POA Manager, and therefore the Root POA, to the active state. We then enter the ORB's event loop in 24.

In this example, `run()` never returns, because we don't provide a means to shut down the ORB. If that ever happened, lines 26–27 would first destroy the Root POA. Since that deactivates our active HelloWorld object, we can then safely delete the servant.

Since the Root POA has the `IMPLICIT_ACTIVATION` policy, we can also use several other methods to activate the servant instead of `activate_object()`. We could, for example, use `servant_to_reference()`, which first implicitely activates the inactive servant and then returns an object reference pointing to the servant. Or, we could invoke the servant's inherited `_this` method, which also implicitely activates the servant and returns an object reference.

4.4.4 Using a Servant Manager

While the previous example did introduce the POA, it did not demonstrate any of its abilities – the example would have been just as simple using the BOA.

As a more complex example, we want to show a server that generates "virtual" object references that point to non-existent objects. We then provide the POA with a servant manager that incarnates the objects on demand.

We continue our series of "Account" examples. We provide the implementation for a Bank object with a single "create" operation that opens a new account. However, the Account object is not put into existence at that point, we just return a reference that will cause activation of an Account object when it is first accessed. This text will only show some code fragments; find the full code in the `demo/poa/account-2` directory.

The implementation of the Account object does not differ from before. More interesting is the implementation of the Bank's `create` operation:

```
Account_ptr
Bank_impl::create ()
{
  CORBA::Object_var obj = mypoa->create_reference ("IDL:Account:1.0");
  Account_ptr aref = Account::_narrow (obj);
  assert (!CORBA::is_nil (aref));
  return aref;
}
```

The `create_reference()` operation on the POA does not cause an activation to take place. It only creates a new object reference encapsulating information about the supported interface and a unique (system-generated) Object Id. This reference is then returned to the client.

Now, when the client invokes an operation on the returned reference, the POA will first search its Active Object Map, but will find no servant to serve the request. We therefore implement a servant manager, which will be asked to find an appropriate implementation.

There are two types of servant managers: a `Servant Activator` activates a new servant, which will be retained in the POA's Active Object Map to serve further requests on the same object. A `Servant Locator` is used to locate a servant for a single invocation only; the servant will not be retained for future use. The type of servant manager depends on the POA's Servant Retention policy.

In our case, we use a servant activator, which will incarnate and activate a new servant whenever the account is used *first*. Further operations on the same object reference will use the already active servant. Since the `create_reference()` operation uses a unique Object Id each time it is called, one new servant will be incarnated for each Account – this represents the BOA's *Unshared* activation mode.

A servant activator provides two operations, `incarnate` and `etherealize`. The former one is called when a new servant needs to be incarnated to serve a previously unknown Object Id. `etherealize` is called when the servant is deactivated (for example in POA shutdown) and allows the servant manager to clean up associated data.

```
class AccountManager : public virtual POA_PortableServer::ServantActivator
{ /* declarations */ };

PortableServer::Servant
AccountManager::incarnate (/* params */)
{
  return new Account_impl;
}
```

```
void
AccountManager::etherealize (PortableServer::Servant serv,
                             /* many more params */)
{
  delete serv;
}
```

Our servant activator implements the `POA_PortableServer::ServantActivator` interface. Since servant managers are servants themselves, they must be activated like any other servant (see below).

The `incarnate` operation has nothing to do but to create a new Account servant. `incarnate` receives the current POA and the requested Object Id as parameters, so it would be possible to perform special initialization based on the Object Id that is to be served.

`etherealize` is just as simple, and deletes the servant. In "real life", the servant manager would have to make sure that the servant is not in use anywhere else before deleting it. Here, this is guaranteed by our program logic.

The `main()` code is a little more extensive than before. Because the Root POA has the `USE_ACTIVE_OBJECT_MAP_ONLY` policy and does not allow a servant manager, we must create our own POA with the `USE_SERVANT_MANAGER` policy.

```
CORBA::ORB_var orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
CORBA::Object_var poaobj = orb->resolve_initial_references ("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow (poaobj);
PortableServer::POAManager_var mgr = poa->the_POAManager();

CORBA::PolicyList pl;
pl.length(1);
pl[0] = poa->
  create_request_processing_policy (PortableServer::USE_SERVANT_MANAGER);
PortableServer::POA_var mypoa = poa->create_POA ("MyPOA", mgr, pl);
```

Note that we use the Root POA's POA Manager when creating the new POA. This means that the POA Manager has now control over both POAs, and changing its state affects both POAs. If we passed `NULL` as the second parameter to `create_POA()`, a new POA Manager would have been created, and we would have to change both POA's states separately.

We can now register the servant manager.

```
AccountManager * am = new AccountManager;
PortableServer::ServantManager_var amref = am->_this ();
mypoa->set_servant_manager (amref);
```

After creating an instance of our servant manager, we obtain an object reference using the inherited `_this()` method. This also implicitely activates the servant manager in the Root POA.

```
Bank_impl * micocash = new Bank_impl (mypoa);
PortableServer::ObjectId_var oid = poa->activate_object (micocash);
mgr->activate ();
orb->run();
```

Now the only thing left to do is to activate a Bank object, to change both POAs to the active state, and to enter the ORB's event loop.

4.5 IDL Compiler

MICO offers its own IDL–compiler called `idl` which is briefly described in this section. The tool is used for translating IDL–specifications to C++ as well as feeding IDL–specifications into the interface repository. The `idl` tool takes its input either from a file or an interface repository and generates code for C++ or CORBA–IDL. If the input is taken from a file, the `idl` tool can additionally feed the specification into the interface repository. The synopsis for `idl` is as follows:

```
idl [--help] [--version] [-D<define>] [-I<path>] \
    [--no-exceptions] [--codegen-c++] [--no-codegen-c++] \
    [--codegen-c++] [--no-codegen-c++] [--codegen-idl] \
    [--no-codegen-idl] [--c++-suffix=<suffix>] [--c++-impl] \
    [--h-suffix=<suffix>] [--absolute-paths] [--emit-repoids] \
    [--query-server-for-narrow] [--feed-ir] [--feed-included-defs] \
    [--repo-id=<id>] [--name=<prefix>] [--pseudo] \
    [--poa] [--no-poa] [--boa] [--no-boa] [--no-poa-ties] [<file>]
```

In the following a detailed description of all the options is given:

`--help`
: Gives an overview of all supported command line options.

`--version`
: Prints the version of MICO.

`-D<define>`
: Defines a preprocessor macro. This option is equivalent to the `-D` switch of most C–compilers.

`-I<path>`
: Defines a search path for `#include` directives. This option is equivalent to the `-I` switch of most C–compilers.

`--no-exceptions`
: Tells `idl` to disable exception handling in the generated code. Code for the exception classes is still generated but throwing exceptions will result in an error message and abort the program. This option can only be used in conjunction with `--codegen-c++`. This option is off by default.

`--codegen-c++`
: Tells `idl` to generate code for C++ as defined by the language mapping IDL to C++. The `idl` tool will generate two files, one ending in `.h` and one in `.cc` with the same basenames. This option is the default.

`--no-codegen-c++`
: Turns off the code generation for C++.

`--codegen-c`
: Tells `idl` to generate code for C as defined by the language mapping IDL to C. The `idl` tool will generate three files, ending in `-skel.cc`, `-stub.cc` and `-c.h` with the same basenames.

`--no-codegen-c`
: Turns off the code generation for C. This is the default.

`--codegen-idl`
: Turns on the code generation for CORBA–IDL. The `idl` tool will generate a file which contains the IDL specification which can again be fed into the `idl` tool. The basename of the file is specified with the `--name` option.

`--no-codegen-idl`
: Turns off the code generation of CORBA–IDL. This option is the default.

`--c++-suffix=<suffix>`
: If `--codegen-c++` is selected, then this option determines the suffix for the C++ implementation file. The default is “`cc`”.

`--c++-impl`
: This option will cause the generation of some default C++ implementation classes for all interfaces contained in the IDL specification. This option requires `--codegen-c++`.

`--hh-suffix=<suffix>`
: If `--codegen-c++` is selected, then this option determines the suffix for the C++ header file. The default is “`h`”.

`--h-suffix=<suffix>`
: If `--codegen-c` is selected, then this option determines the suffix for the C header file. The default is “`h`”.

`--relative-paths`
: If selected, included files (via the `#include` directive) will be referenced in a relative way (i.e. `#include` <...>).

`--emit-repoids`
: This option will cause `#pragma` directives to be emitted, which associate the repository id of each IDL construct. This option can only be used in conjunction with the option `--codegen-idl`.

`--query-server-for-narrow`
: This option can only be used in conjunction with the `--codegen-c++` switch. If it is used, the IDL compiler will insert special code for all `_narrow()` methods for querying the server at runtime. See `test/idl/26/README` for further comments.

`--feed-ir`
: The CORBA–IDL which is specified as a command line option is fed into the *interface repository*. This option requires the `ird` daemon to be running.

`--feed-included-defs`
: This option can only be used in conjunction with `--feed-ir`. If this option is used, IDL definitions located in included files are fed into the interface repository as well. The default is to feed only the definitions of the main IDL file into the IR.

`--repo-id=<id>`
: The code generation is done from the information contained in the *interface repository* instead from a file. This option requires the `ird` daemon to be running. The parameter `id` is a repository identifier and must denote a CORBA module.

`--name=<prefix>`
: This option controls the prefix of the file names if a code generation is selected. This option is mandatory if the input is taken from the interface repository. If the input is taken from a file, the prefix is derived from the basename of the file name.

`--pseudo`
: Generates code for "pseudo interfaces". No stubs, skeletons or code for marshalling data to and from "any" variables is produced. Only supported for C++ code generation.

`--poa`
: Turns on generation of skeleton classes based on the Portable Object Adapter (POA).

`--no-poa`
: Turns off generation of POA-based skeletons. This is the default.

`--no-poa-ties`
: When using `--poa`, this option can be used to turn off generation of Tie classes if not needed.

`--boa`
: Turns on generation of skeleton classes using the Basic Object Adapter (BOA). This is the default.

`--no-boa`
: Turns off generation of BOA-based skeletons.

Here are some examples on how to use the `idl` tool:

`idl account.idl`
: Translates the IDL–specification contained in account.idl according to the C++ language mapping. This will generate two files in the current directory.

`idl --feed-ir account.idl`
: Same as above but the IDL–specification is also fed into the interface repository.

`idl --feed-ir --no-codegen-c++ account.idl`
: Same as above but the generation of C++ stubs and skeletons is omitted.

`idl --repo-id=IDL:Account:1.0 --no-codegen-c++ --codegen-idl --name=out`
: This command will generate IDL–code from the information contained in the interface repository. This requires the `ird` daemon to be running. The output is written to a file called `out.idl`.

`idl --no-codegen-c++ --codegen-idl --name=out account.idl`
: This command will translate the IDL–specification contained in `account.idl` and into a semantical equivalent IDL–specification in file `out.idl`. This could be useful if you want to misuse the IDL–compiler as a pretty printer.

4.6 Compiler and Linker Wrappers

It can be quite complicated to compile and link MICO applications because you have to specify system dependent compiler flags, linker flags and libraries. This is why MICO provides you with four shells scripts:

`mico-c++`
: should be used as the C++ compiler when compiling the C++ source files of a MICO–application.

`mico-ld`
: should be used as the linker when linking together the `.o` files of a MICO–application.

`mico-shc++`
: should be used as the C++ compiler when compiling the C++ source files of a MICO dynamically loadable module. `mico-shc++` will not be available unless you specified the `--enable-dynamic` option during configuration.

`mico-shld`
: should be used as the linker when linking together the `.o` files of a MICO dynamically loadable module. `mico-shld` will not be available unless you specified the `--enable-dynamic` option during configuration.

The scripts can be used just like the normal compiler/linker, except that for `mico-shld` you do not specify a file name suffix for the output file because `mico-shld` will append a system dependent shared object suffix (`.so` on most systems) to the specified output file name.

4.6.1 Examples

Let us consider building a simple MICO–aplication that consists of two files: `account.idl` and `main.cc`. Here is how to build `account`:

```
idl account.idl
mico-c++ -I. -c account.cc -o account.o
mico-c++ -I. -c main.cc -o main.o
mico-ld account.o main.o -o account -lmico2.2.3
```

As a second example let us consider building a dynamically loadable module and a client program that loads the module. We have three source files now: `account.idl`, `client.cc`, and `module.cc`:

```
idl account.idl
mico-shc++ -I. -c account.cc -o account.o
mico-shc++ -I. -c module.cc -o module.o
mico-shld -o module module.o account.o -lmico2.2.3

mico-c++ -I. -c client.cc -o client.o
mico-ld account.o client.o -o client -lmico2.2.3
```

Note that

- all files that go into the module must be compiled using `mico-shc++` instead of `mico-c++`.
- `module` was specified as the output file, but `mico-shld` will generate `module.so` (the extension depends on your system).
- `account.o` must be linked both into the module and the client but is compiled only once using `mico-shc++`. One would expect that `account.cc` had to be compiled twice: once with `mico-c++` for use in the client and once with `mico-shc++` for use in the module. The rule is that using `mico-shc++` where `mico-c++` should be used does not harm, but *not* the other way around.

Chapter 5

C++ mapping

This chapter features some highlights of the IDL to C++ mapping. Sometimes we just quote facts from the CORBA standard, sometimes we describe some details which are specific to MICO.

5.1 Using strings

Strings have always been a source of confusion. The CORBA standard adopts a not necessarily intuitive mapping for strings for the C++ language. The following description is partially taken from chapter [16.7] in the CORBA 2.2 specification.

As in the C mapping, the OMG IDL string type, whether bounded or unbounded, is mapped to `char*` in C++. String data is null–terminated. In addition, the CORBA module defines a class `String_var` that contains a `char*` value and automatically frees the pointer when a `String_var` object is deallocated. When a `String_var` is constructed or assigned from a `char*`, the `char*` is consumed and thus the string data may no longer be accessed through it by the caller. Assignment or construction from a `const char*` or from another `String_var` causes a copy. The `String_var` class also provides operations to convert to and from `char*` values, as well as subscripting operations to access characters within the string. The full definition of the `String_var` interface is given in appendix C–2 of the CORBA 2.2 specification.

For dynamic allocation of strings, compliant programs must use the following functions from the CORBA namespace:

```
// C++
namespace CORBA {
  char *string_alloc( ULong len );
  char *string_dup( const char* );
  void string_free( char * );
  ...
}
```

The `string_alloc` function dynamically allocates a string, or returns a null pointer if it cannot perform the allocation. It allocates `len+1` characters so that the resulting string has enough space to hold a trailing NULL character. The `string_dup` function dynamically allocates enough space to hold a copy of its string argument, including the NULL character, copies its string argument into that memory, and returns a pointer to the new string. If allocation fails, a null pointer is returned. The `string_free` function deallocates a string that was allocated with `string_alloc` or `string_dup`. Passing a null pointer to `string_free` is acceptable and results in no action being performed.

Note that a static array of char in C++ decays to a `char*`, so care must be taken when assigning one to a `String_var`, since the `String_var` will assume the pointer points to data allocated via `string_alloc` and thus will eventually attempt to `string_free` it:

```
// C++
// The following is an error, since the char* should point to
// data allocated via string_alloc so it can be consumed
String_var s = "static string"; // error

// The following are OK, since const char* are copied,
// not consumed
const char* sp = "static string";
s = sp;
s = (const char*)"static string too";
```

See the directory `mico/test/idl/5` for some examples on how to use strings in conjunction with operations.

5.2 Untyped values

The handling of untyped values is one of CORBAs strengths. The pre–defined C++ class `Any` in the namespace `CORBA` provides this support. An instance of class `Any` represents a value of an arbitrary IDL–type. For each type, the class `Any` defines the overloaded operators `>>=` and `<<=`. These two operators are responsible for the insertion and extraction of the data values. The following code fragment demonstrates the usage of these operators:

```
// C++
CORBA::Any a;

// Insertion into any
a <<= (CORBA::ULong) 10;

// Extraction from any
CORBA::ULong l;
a >>= l;
```

At the end of this example the variable `l` should have the value 10. The library of Mico provides overloaded definitions of these operators for all basic data types. Some of these data types are ambiguous in the sense that they collide with other basic data types. This is true for the IDL–types `boolean`, `octet`, `char` and `string`. For each of these IDL–types, CORBA prescribes a pair of supporting functions which help to disambiguate the type clashes. For the type `boolean` for example the usage of these supporting function is:

```
CORBA::Any a;

// Insertion into any
a <<= CORBA::Any::from_boolean( TRUE );

// Extraction from any
CORBA::Boolean b;
a >>= CORBA::Any::to_boolean( b );
```

The usage of the other supporting functions for `octet`, `char` and `string` is equivalent. For bounded strings the supporting functions `from_string` and `to_string` accept an additional `long`–parameter which reflects the bound.

For each type defined in an IDL specification, the IDL–compiler generates an overloaded version of the operators `>>=` and `<<=`. For example given the following IDL specification:

```
// IDL
struct S1 {
  long x;
  char c;
};

struct S2 {
  string str;
};
```

The Mico IDL–compiler will automatically generate appropriate definitions of `>>=` and `<<=` for the IDL types `S1` and `S2`. The following code fragment demonstrates the usage of these operators:

```
void show_any( const CORBA::Any& a )
{
  S1 s1;
  S2 s2;

  if( a >>= s1 ) {
    cout << "Found struct S1" << endl;
    cout << s1.x << endl;
    cout << s1.c << endl;
  }
  if( a >>= s2 ) {
    cout << "Found struct S2" << endl;
    cout << s2.str << endl;
  }
}

int main( int argc, char *argv[] )
{
```

```
19:   //...
20:   CORBA::Any a;
21:
22:   S2 s2;
23:   s2.str = (const char *) "Hello";
24:   a <<= s2;
25:   show_any( a );
26:
27:   S1 s1;
28:   s1.x = 42;
29:   s1.c = 'C';
30:   a <<= s1;
31:   show_any( a );
32: }
```

The main program first initializes an instance of a `S2` (lines 22–24) and then calls the function `show_any`. Function `show_any` tries to extract the value contained in the any. This example also demonstrates how to tell whether the extraction was successful or not. The operator `>>=` returns true, iff the type of the value contained in the any matches with the type of the variable of the right side of `>>=`. If the any should contain something else than `S1` or `S2`, then `show_any` will fall through both `if`–statements in lines 6 and 11. The complete sources for the above example can be found in `mico/test/idl/14`.

5.2.1 Unknown Constructed Types

MICO's `Any` implementation offers an extended interface for typesafe insertion and extraction of constructed types that were not known at compile time. This interface is also used by the `<<=` and `>>=` operators generated by the IDL compiler for constructed types. Lets look at the generated operators for a simple structure:

```
// IDL
struct foo {
  long l;
  string s;
};

// C++
CORBA::Boolean operator<<= ( CORBA::Any &a, const foo &s )
{
  a.type( _tc_foo );
  return a.struct_put_begin() &&
         (a <<= s.l) &&
         (a <<= s.s) &&
         a.struct_put_end();
}

CORBA::Boolean operator>>=( const CORBA::Any &a, foo &s )
{
```

```
19:    return a.struct_get_begin() &&
20:           (a >>= s.l) &&
21:           (a >>= s.s) &&
22:           a.struct_get_end();
23:  }
```

The `<<=` operator tells the `Any` the `TypeCode` (`_tc_foo`) of the to be inserted structure in line 10. Those `_tc_*` constants are generated by the IDL compiler as well. If you want to insert a constructed type that was not known at compile time you have to get the `TypeCode` from somewhere else (e.g., from the interface repository) or you have to create one using the `create_*_tc()` ORB methods.

After telling the `Any` the `TypeCode` the `<<=` operator opens a structure in line 11, shifts in the elements of the struct in lines 12–13 and closes the struct in line 14. While doing so the `Any` checks the correctness of the inserted items using the `TypeCode`. If it detects an error (e.g., the `TypeCode` says the first element of the struct is a short and you insert a float) the corresponding method or `<<=` operator will return FALSE. If the structure contained another constructed type you had to make nested calls to `struct_put_begin()` and `struct_put_end()` or the corresponding methods for unions, exceptions, arrays, or sequences.

The `>>=` operator in lines 17–23 has the same structure as the `<<=` operator but uses `>>=` operators to extract the struct elements and `struct_get_begin()` and `struct_get_end()` to open and close the structure. There is no need to specify a `TypeCode` before extraction because the `Any` knows it already.

5.2.2 Subtyping

Another feature of Mico's `Any` implementation is its subtyping support. The extraction operators of type `Any` implement the subtyping rules for recursive types as prescribed by the *Reference Model for Open Distributed Processing* (RM–ODP), see [1, 2, 3, 4] for details. The idea behind subtyping is the following: Imagine you want to call a CORBA method

```
void bar (in long x);
```

but want to pass a `short` as an argument instead of the required `long`. This should work in theory since each possible `short` value is also a `long` value which means `short` is a subtype of `long`. More generally speaking a type T_1 is a subtype of type T_2 if you could pass T_1 as an input parameter where a T_2 is expected. This means for basic types such as `long`: a basic type T_1 is a subtype of a basic type T_2 iff the set of possible values of T_1 is a subset of the set of possible values of T_2. Figure 5.1 shows the subtype relations between CORBA's basic data types. In C++ the compiler can automatically convert types along a chain of arrows, but in a distributed CORBA application this can't be done by the compiler alone because binding between client and server is performed at runtime using a trader or a naming service. That is the subtype checking must be done at runtime as well.
In Mico the `Any` type performs subtype checking at runtime. For example:

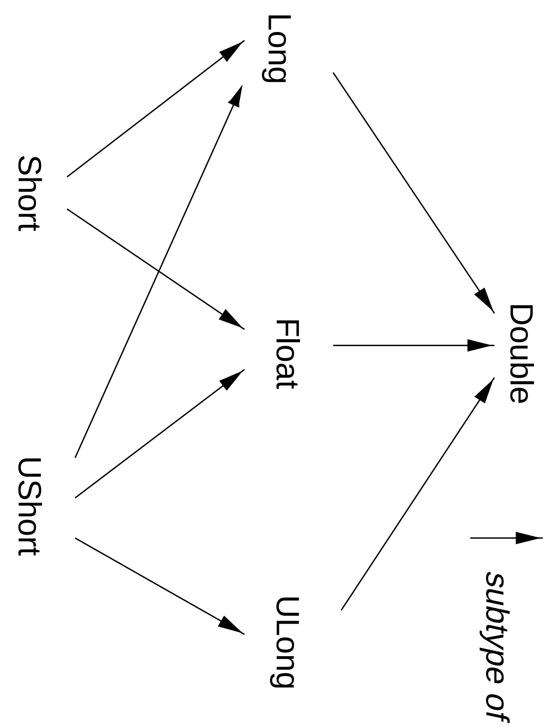

Figure 5.1: Subtype relations between basic CORBA types.

```
// C++
CORBA::Any a;
a <<= (CORBA::Short) 42;
...
CORBA::Double d;
a >>= d;
```

will work because `short` is a subtype of `double` according to figure 5.1 but:

```
// C++
CORBA::Any a;
a <<= (CORBA::Long) 42;
...
CORBA::ULong d;
a >>= d;
```

will fail because `long` is not a subtype of `unsigned long`. There is a special subtyping rule for structured types: A struct type T_1 is a subtype of a struct type T_2 iff the elements of T_2 are supertypes of the first elements of T_1. `struct S1` is for example a subtype of `struct S2`:

```
struct S1 {
  short s;
  long l;
```

```
};

struct S2 {
  long s;
};
```

That is you can put a `struct S1` into an `Any` and unpack it as a `struct S2` later:

```
// C++
CORBA::Any a;
S1 s1 = { 10, 20 };
a <<= s1;
...
S2 s2;
a >>= s2;
```

There are similar rules for the other constructed types.

5.3 Arrays

Arrays are handled somewhat awkwardly in CORBA. The C++ mapping for the declaration of an array is straight forward. Things are getting a bit more complicated when arrays are being passed around as parameters of operations. Arrays are mapped to the corresponding C++ array definition, which allows the definition of statically–initialized data using the array. If the array element is a string or an object reference, then the mapping uses the same type as for structure members. That is, assignment to an array element will release the storage associated with the old value.

```
// IDL
typedef string V[10];
typedef string M[1][2][3];

// C++
V v1; V_var v2;
M m1; M_var m2;

v1[1] = v2[1]; // free old storage, copy
m1[0][1][2] = m2[0][1][2]; // free old storage, copy
```

In the above example, the two assignments result in the storage associated with the old value of the left–hand side being automatically released before the value from the right–hand side is copied.

Because arrays are mapped into regular C++ arrays, they present special problems for the type–safe `Any` mapping described in [16.14]. To facilitate their use with the type `Any`, Mico also provides for each array type a distinct C++ type whose name consists of the array name followed by the suffix `_forany`. Like `Array_var` types, `Array_forany` types allow access to the underlying array type. The interface of the `Array_forany` type is identical to that of the `Array_var` type.

```
// IDL
typedef string V[10];

// C++
V_forany v1, v2;
v1[0] = ...;  // Initialize array

CORBA::Any any;
any <<= v1;
any >>= v2;   // v1 and v2 now have identical contents
```

Besides the `Array_forany` mapping the CORBA standard also describes a mapping for an *array slice*. A slice of an array is an array with all the dimension of the original but the first. Output parameters and results are handled via pointers to array slices. The array slice is named like the array itself plus appending the suffix `_slice`. For the declaration of type `M` in the example above, the IDL compiler would generate the following type definition:

```
// Generated by IDL compiler, C++
typedef M M_slice[2][3];
```

Let's consider the following IDL specification (see also `mico/test/idl/18`):

```
// IDL
// Note: long_arr is an array of fixed length data type
typedef long long_arr[ 10 ];

// Note: SS is an array of variable data type
typedef string SS[ 5 ][ 4 ];

interface foo {
  SS bar( in SS x, inout SS y, out SS z, out long_arr w );
};
```

The implementation of interface `foo` will look like this:

```
class foo_impl : virtual public foo_skel
{
  //...
  SS_slice* bar( const SS ss1, SS ss2, SS_slice*& ss3, long_arr arr )
  {
    //...
    ss3 = SS_alloc();
    SS_slice *res = SS_alloc();
    return res;
  };
};
```

Note that the result value of the operation bar is a pointer to an array slice. Output parameters where the type is an array to a variable length data type, are handled via a reference to a pointer of an array slice. In order to facilitate memory management with array slices, the CORBA standard prescribes the usage of special functions defined at the same scope as the array type. For the array SS, the following functions will be available to a program:

```
// C++
SS_slice *SS_alloc();
SS_slice *SS_dup( const SS_slice* );
void SS_free( SS_slice * );
```

The SS_alloc function dynamically allocates an array, or returns a null pointer if it cannot perform the allocation. The SS_dup function dynamically allocates a new array with the same size as its array argument, copies each element of the argument array into the new array, and returns a pointer to the new array. If allocation fails, a null pointer is returned. The SS_free function deallocates an array that was allocated with SS_alloc or SS_dup. Passing a null pointer to SS_free is acceptable and results in no action being performed.

5.4 Unions

Unions and structs in the CORBA–IDL allow the definition of constructed data types. Each of them is defined through a set of members. Is a struct used as an input parameter of an operation, all of its members will be transmitted, whereas for a union at most one of its members will actually be transmitted. The purpose of an IDL–union is similar to that of a C–union: reduction of memory usage. This is especially important in a middleware platform where less memory space for a data type also means less data to transfer over the network. One must carefully consider, when structs or unions should be used.

A special problem arises with unions when they are being used as parameters of operation invocations: how does the receiving object know which of the different members holds a valid value? In order to make a distinction for this case, the IDL–union is a combination of a C–union and a C–switch statement. Each member is clearly tagged with a value of a given discriminator type (see also mico/test/idl/21):

```
// IDL
typedef octet Bytes[64];
struct S { long len; };
interface A;

union U switch (long) {
  case 1: long x;
  case 2: Bytes y;
  case 3: string z;
  case 4:
  case 5: S w;
```

```
  default: A obj;
};
```

In the union U as shown above, long is the discriminator type. The values following the case label must belong to this discriminator type. All integer types and enums are valid discriminator types. Unions map to C++ classes with access functions for the union members and discriminant. The default union constructor performs no application–visible initialization of the union. It does not initialize the discriminator, nor does it initialize any union members to a state useful to an application. It is therefore an error for an application to access the union before setting it. The copy constructor and assignment operator both perform a deep–copy of their parameters, with the assignment operator releasing old storage if necessary. The destructor releases all storage owned by the union. The following example helps illustrate the mapping for union types for the union U as shown above:

```
// Generated C++ code
typedef CORBA::Octet Bytes[64];
typedef CORBA::Octet Bytes_slice;
template<...> Bytes_forany;
struct S { CORBA::Long len; };
typedef ... A_ptr;

class U {
  public:
    //...
    void _d( CORBA::Long );
    CORBA::Long _d() const;

    void x( CORBA::Long );
    CORBA::Long x() const;

    void y( Bytes );
    Bytes_slice *y() const;

    void z( char* );              // free old storage, no copy
    void z( const char* );        // free old storage, copy
    void z( const String_var& ); // free old storage, copy
    const char *z() const;

    void w( const S & ); // deep copy
    const S &w() const;  // read-only access
    S &w();              // read-write access

    void obj( A_ptr ); // release old objref, duplicate
    A_ptr obj() const; // no duplicate
};
```

The union discriminant access functions have the name _d to both be brief and avoid name conflicts with the members. The _d discriminator modifier function can only be used to set the discriminant to a value within the same union member. In addition to the `_d` accessors, a union with an implicit default member provides a `_default()` member function that sets the discriminant to a legal default value. A union has an implicit default member if it does not have a default case and not all permissible values of the union discriminant are listed.

Setting the union value through an access function automatically sets the discriminant and may release the storage associated with the previous value. Attempting to get a value through an access function that does not match the current discriminant results in undefined behavior. If an access function for a union member with multiple legal discriminant values is used to set the value of the discriminant, the union implementation will choose the value of the first case label in the union (e.g. value 4 for the member `w` of union `U`), although it could be any other value for that member as well.

The restrictions for using the `_d` discriminator modifier function are shown by the following examples, based on the definition of the union `U` shown above:

```
// C++
S s = ...;
A_ptr a = ...;
U u;

u.w( s );   // member w selected, discrimintator == 4
u._d( 4 );  // OK, member w selected
u._d( 5 );  // OK, member w selected
u._d( 1 );  // error, different member selected
u.obj( a ); // member obj selected
u._d( 7 );  // OK, member obj selected
u._d( 1 );  // error, different member selected
```

As shown here, the `_d` modifier function cannot be used to implicitly switch between different union members. The following shows an example of how the `_default()` member function is used:

```
// IDL
union Z switch(boolean) {
  case TRUE: short s;
};

// C++
Z z;
z._default();  // implicit default member selected
CORBA::Boolean disc = z._d(); // disc == FALSE
U u;           // union U from previous example
u._default();  // error, no _default() provided
```

For union `Z`, calling the `_default()` member function causes the union's value to be composed solely of the discriminator value of `FALSE`, since there is no explicit default member. For union `U`, calling `_default()` causes a compilation error because `U` has an explicitly declared default case and thus no `_default()` member function. A `_default()` member function is only generated for unions with implicit default members.

For an array union member, the accessor returns a pointer to the array slice, where the slice is an array with all dimensions of the original except the first (see section 5.3 for a discussion on array slices). The array slice return type allows for read–write access for array members via regular subscript operators. For members of an anonymous array type, supporting typedefs for the array are generated directly into the union. For example:

```
// IDL
union U switch (long) {
  case 1: long array[ 3 ][ 4 ];
};

// Generated C++ code
class U {
  public:
  // ...
  typedef long _array_slice[ 4 ];
  void array( long arg[ 3 ][ 4 ] );
  _array_slice* array();
};
```

The name of the supporting array slice typedef is created by prepending an underscore and appending `_slice` to the union member name. In the example above, the array member named `_array` results in an array slice typedef called `_array_slice` nested in the union class.

5.5 Interface inheritance

The CORBA standard prescribes that IDL–interfaces need to be mapped to C++ classes for the C++ language binding. The question arises, how things are handled when interface inheritance is used. MICO offers two alternatives for implementing the skeletons when using interface inheritance. Consider the following IDL definitions:

```
interface Base {
  void op1();
};

interface Derived : Base {
  void op2();
};
```

`Base` is an interface and serves as a base for interface `Derived`. This means that all declarations in `Base` are inherited to `Derived`. As we have seen before, the `idl` tool creates stub– and skeleton–classes for each interface. The operations map to pure virtual functions which have to be implemented by the programmer. For the interface `Base` this is straight forward:

```
class Base_impl : virtual public Base_skel
{
public:
  Base_impl()
  {
  };
  void op1()
  {
    cout << "Base::op1()" << endl;
  };
};
```

The skeleton for `Derived` allows two different possible ways to implement the skeleton. The difference between the two is, whether the implementation of `Derived` inherits the implementation of `Base` or not. Let's take a look on how this translates to lines of code. Here is the first alternative:

```
class Derived_impl :
  virtual public Base_impl,
  virtual public Derived_skel
{
public:
  Derived_impl()
  {
  };
  void op2()
  {
    cout << "Derived::op2()" << endl;
  };
};
```

In the code fragment above, the implementation of `Derived` inherits the implementation of `Base`. Note that `Derived_impl` inherits from `Base_impl` and therefore needs only to implement `op2()` since `op1()` is already implemented in `Base_impl`.

Important note: when implementing a class `X_impl` that inherits from multiple base classes you have to ensure that the `X_skel` constructor is the last one that is called. This can be accomplished by making `X_skel` the rightmost entry in the inheritance list:

```
class X_impl : ..., virtual public X_skel {
  ...
};
```

Now comes the second alternative (note that the skeleton classes are still the same; there is no particular switch with the `idl` tool where you have to decide between the two alternatives):

```
class Derived_impl :
  virtual public Base_skel,
```

```
    virtual public Derived_skel
  {
  public:
    Derived_impl()
    {
    };
    void op1()
    {
      cout << "Derived::op1()" << endl;
    };
    void op2()
    {
      cout << "Derived::op2()" << endl;
    };
  };
```

You should notice two things: first of all `Derived_impl` is no longer derived from `Base_impl` but rather from `Base_skel`. For this reason the class `Derived_impl` needs to implement the operation `op2()` itself. Figure 5.2 shows the inheritance hierarchy for the classes generated by the IDL–compiler and their relationship to the classes contained in the MICO library. Compare this with figure 3.3 on page 15. This example can also be found in the directory `mico/test/idl/15`.

5.6 Modules

In contrast to other middleware platforms, CORBA does not assign an universal unique identifier (UUID) to an interface. To avoid name clashes, CORBA offers a structured name space, similar to the directory structure of a UNIX file system. Within an IDL a scope is defined by the keyword `module`. For example the following IDL–code excerpt defines two modules called `Mod1` and `Mod2` on the same level:

```
  module Mod1 {
    //...
    interface foo;
  };

  module Mod2
  {
    //...
  };
```

Module declarations can be nested which leads to the above mentioned hierarchical namespace. The IDL to C++ mapping offers different alternatives on how to map a module to C++. Those C++ compilers which support the namespace feature of the C++ language, IDL–modules are directly mapped to C++ namespaces. Unfortunately the GNU compiler currently does not support namespaces. In this case the CORBA specification offers two alternatives: either do some name mangling such that a name reflects

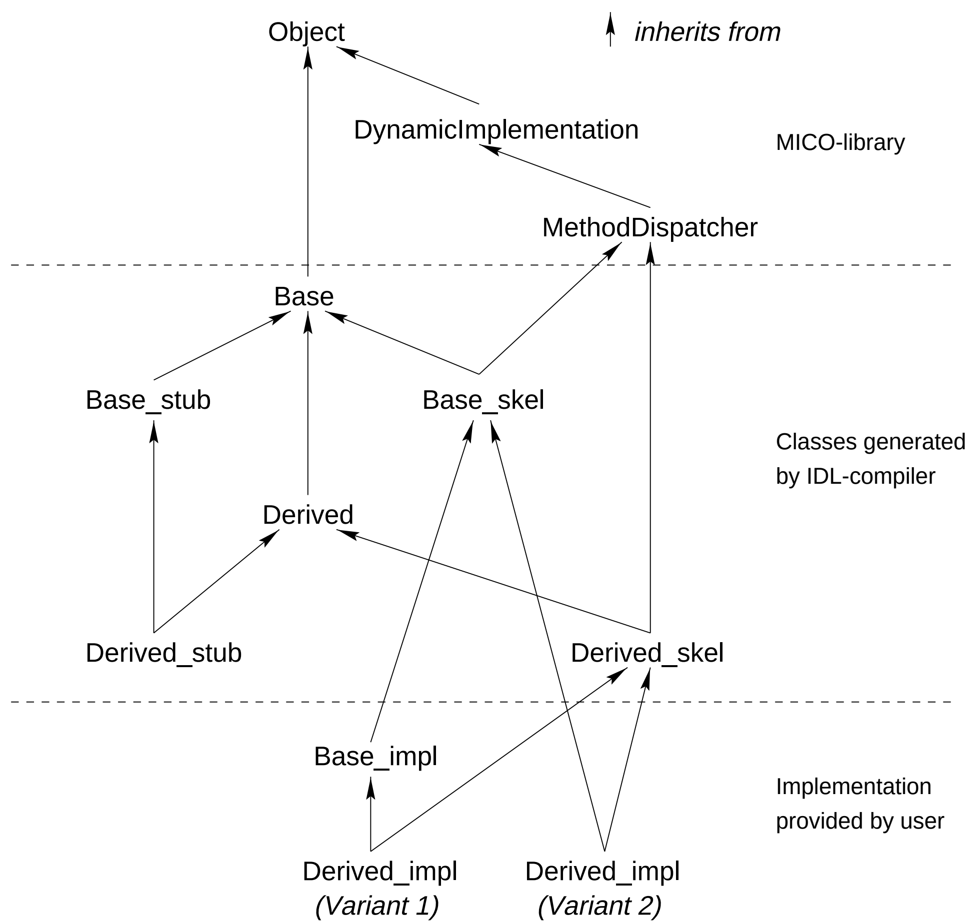

Figure 5.2: C++ class hierarchy for interface inheritance.

the absolute name of the IDL–identifier where the names are separated by undersores (e.g. `Mod1_foo`). The second alternative is to map an IDL–module to a C++ `struct`.

The second alternative has two drawbacks: without a proper support for namespaces all names have to be referenced by their absolute names, i.e. there is no C++ keyword `using` (note that this is also true for the first alternative). The second drawback has to do with the possibility to re–open CORBA–modules which allows cyclic definitions:

```
module M1 {
  typedef char A;
};

module M2
{
  typedef M1::A B;
};

module M1 {  // re-open module M1
{
  typedef M2::B C;
};
```

The declaration of a C++ `struct` has to occur in one location (i.e. a `struct` can not be re–opened). Mapping IDL–modules to C++ structs therefore implies, that re–opening of modules can not be translated to C++. However, if the C++ compiler supports namespaces, Mico's IDL–compiler allows the re–opening of modules. The backend of Mico's IDL–compiler generates a dependency graph to compute the correct ordering of IDL definitions. Figure 5.3 shows the dependency graph for the IDL specification shown above. The correct ordering of IDL definitions is done by doing a left–to–right, depth–first, post–order traversal of the dependency graph starting from `_top`, and omitting previously visited nodes of the graph.

Sometimes it is necessary to have some control over the top–level modules. This for example is used in `CORBA.h` where some definitions have to be read in one at a time. The IDL–compiler inserts some `#define` in the generated `.h` file. Setting and unsetting these defines allows to read the module definitions one at a time. Given the two modules `Mod1` and `Mod2` as above, the following C++ code fragment demonstrates how to do this:

```
// These #includes need to be done manually if
// MICO_NO_TOPLEVEL_MODULES is defined
#include <CORBA.h>
#include <mico/template_impl.h>

#define MICO_NO_TOPLEVEL_MODULES

// Get module Mod1
#define MICO_MODULE_Mod1
struct Mod1 {
  #include "module.h"
```

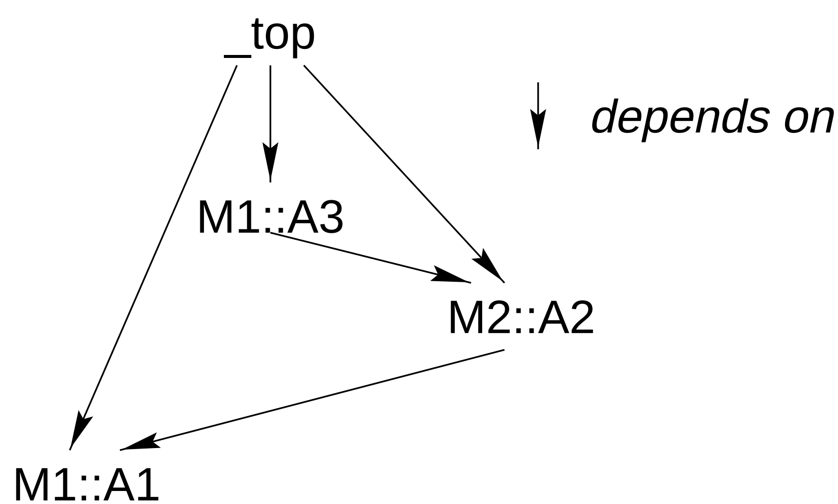

Figure 5.3: Dependency graph.

```
11:  };
12:  #undef MICO_MODULE_Mod1
13:
14:  // Get module Mod2
15:  #define MICO_MODULE_Mod2
16:  struct Mod2 {
17:    #include "module.h"
18:  };
19:  #undef MICO_MODULE_Mod2
20:
21:  // Get global definitions in module.h
22:  #define MICO_MODULE__GLOBAL
23:  #include "module.h"
24:  #undef MICO_MODULE__GLOBAL
25:  #undef MICO_NO_TOPLEVEL_MODULES
```

In this example we assume that the definitions are located in a file called `module.h`. First of all you need to define `MICO_NO_TOPLEVEL_MODULES` which simply means that you wish to read in the definitions yourself (line 6). For each toplevel module `XYZ` in an IDL–file there exists a define called `MICO_MODULE_XYZ`. Setting this define will activate all definitions which belong to module `XYZ` (see lines 9 and 15). Do not forget to undefine these definitions after the definitions are read in (lines 12 and 19). There are some global definitions which do not belong to any module. For these definitions there in a special define called `MICO_MODULE__GLOBAL` (see line 22; the two underscores are no typo). The last thing we need to do is to undefine `MICO_MODULE__GLOBAL` and `MICO_NO_TOPLEVEL_MODULE` (see lines 24 and 25). This example can also be found in the directory `mico/test/idl/10`.

5.7 Exceptions

Mico's exceptions handling capabilities suffer a lot from the limited exceptions handling support in the GNU C++ compiler, namely:

- exception handling is only supported on very few platforms, notably on Sun SPARC, Intel x86, Motorola 68k, and IBM RS/6000.
- throwing class `X` and catching a base class of `X` does only work if you turn on *Runtime Type Information (RTTI)* support by using the `-frtti` option. But if you compile one file using `-frtti` you have to compile all files using this option, including the iostream– and all other C++ libraries. Besides this g++'s RTTI support ist still very buggy.
- you cannot throw exceptions from a shared library into user code (trying this gives you a segmentation fault).
- there seems to be a memory leak in the compiler generated exception handling code.

These limitations are the reason for the somewhat strange exception handling design in the current version of Mico.

5.7.1 Throwing Exceptions

You must not use the `throw` operator directly, instead you should use the function `mico_throw()` defined in `mico/throw.h`, which is automatically included by IDL compiler generated code:

```
// ok
mico_throw (CORBA::UNKNOWN());

// wrong
throw CORBA::UNKNOWN();
```

will throw the CORBA system exception `UNKNOWN`. User defined exceptions are thrown the same way.

5.7.2 Catching Exceptions

Exceptions are always caught by reference using the `_var` types. System exceptions must be caught by `SystemException_var`:

```
// ok
try {
  ...
  mico_throw (CORBA::UNKNOWN());
  ...
} catch (CORBA::SystemException_var &ex) {
  ...
}

// wrong
try {
  ...
  mico_throw (CORBA::UNKNOWN());
  ...
} catch (CORBA::UNKNOWN_var &ex) {
  ...
}

// wrong
try {
  ...
  mico_throw (CORBA::UNKNOWN());
  ...
} catch (CORBA::Exception_var &ex) {
  ...
}
```

Sometimes it is necessary to know exactly which system exception has been thrown:

```
// ok
try {
  ...
```

```
    mico_throw (CORBA::UNKNOWN());
    ...
  } catch (CORBA::SystemException_var &sys_ex) {
    if (CORBA::UNKNOWN *ukn_ex = CORBA::UNKNOWN::_narrow (sys_ex)) {
      // something1
    } else {
      // something2
    }
  }

  // wrong
  try {
    ...
  } catch (CORBA::UNKNOWN_var &ukn_ex) {
    // something1
  } catch (CORBA::SystemException_var &other_ex) {
    // something2
  }
```

In contrast to system exceptions a user exception `X` must be caught by `X_var` (i.e., not by `UserException_var`):

```
  // ok
  try {
    ...
    mico_throw (SomeExcept());
    ...
  } catch (SomeExcept_var &some_ex) {
    ...
  }

  // wrong
  try {
    ...
    mico_throw (SomeExcept());
    ...
  } catch (CORBA::UserException_var &usr_ex) {
    ...
  }

  // wrong
  try {
    ...
    mico_throw (SomeExcept());
    ...
  } catch (CORBA::Exception_var &ex) {
    ...
  }
```

If an exception is thrown but not caught MICO will print out a short description of the exception and terminate the process. On systems where g++ does not support exception handling throwing an exception will always result in such a message and termination of the process.

Chapter 6

Java Interface

We have implemented a generic user interface to MICO's dynamic invocation interface. The interface is written is Java and allows the invocation of arbitrary operations. The specification of an operation invocation is done with the help of a knowledge representation technique called *conceptual graphs*. This chapter gives an overview of this interface. The outline of this chapter is as follows: in section 6.1 be provide a brief introduction to the theory of conceptual graphs. In section 6.2 we describe CORBAs dynamic invocation interface and the problems related to a generic user interface which allows run–time access to this interface. In section 6.3 we present the anatomy of an operation declaration as defined by the CORBA standard. In section 6.4 we finally present our solution for a generic user interface to CORBAs dynamic invocation interface based on an interactive conceptual graph editor. In section 6.5 we finally show how to run the Java applet using standard JDK tools in conjunction with a graphical browsing tool for the contents of the interface repository. The work in this chapter has been presented in [7].

6.1 Conceptual Graphs

The theory of *conceptual graphs* (CG) has been developed to model the semantics of natural language (see [8]). Specifications based on conceptual graphs are therefore intuitive in the sense that there is a close relationship to the way human beings represent and organize their knowledge. From a mathematical point of view a conceptual graph is a finite, connected, directed, bipartite graph. The nodes of the graph are either *concept* or *relation nodes*. Due to the bipartite nature of the graphs, two concept nodes may only be connected via a relation node. A concept node represents either a concrete or an abstract object in the world of discourse whereas a relation nodes defines a context between two or more concepts.

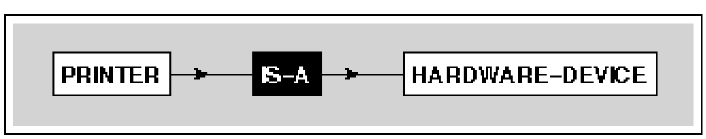

Figure 6.1: A simple conceptual graph with two concepts and one relation.

A sample CG is depicted in figure 6.1. This CG consists of two concepts (white nodes) and one relation (black node). This CG expresses the fact that a printer is a hardware device. The two concepts — `PRINTER` and `HARDWARE-DEVICE` — are placed in a semantical context via the binary relation `IS-A`. The theory of CGs defines a mapping from conceptual graphs to first–order calculus. This mapping, which is described in [8], would map the CG depicted in figure 6.1 to the first order formula $\exists x \exists y : \texttt{PRINTER}(x) \wedge \texttt{HARDWARE-DEVICE}(y) \wedge \texttt{IS-A}(x, y)$. As can be seen, the variables x and y form the link between the two concepts via the predicate `IS-A`.

Given a conceptual and relational catalogue, one can express arbitrary knowledge. For this reason the theory of CG represents a *knowledge representation technique*. The work done in [8] focuses on the representation of natural language. We have shown, that with a suitable conceptual and relational catalogue one can translate operational interface specifications to conceptual graphs (see [6]). We have written translators which translate arbitrary DCE and CORBA–IDL specifications to CGs. Thus we have already demonstrated that an implementation of an interface repository, which is based on such a meta–notation, can be used in different middleware platforms. In the following we show how a meta–notation can also be exploited for the construction of a generic user interface to CORBAs *dynamic invocation interface* (DII).

6.2 Dynamic Invocation Interface

In this section we present a description for CORBAs DII. For the following discussions we refer to the interface `Account` as specified in section 3.3.2. A client application written in C++ might for example use this interface in the following way:

```
Account_ptr acc = ...;  // Obtain a reference to an Account-object

acc->deposit( 100 );
acc->withdraw( 20 );

cout << "Total balance is " << acc->balance() << endl;
```

If we assume that the current balance of the server object was 0 when the variable `acc` was bound with a refence to this object, then this program fragment prints out *“Total balance is 80”*. It should be clear that this program fragment requires the definition of the class `Account_ptr`. This class, which allows a type safe access to a CORBA object implementing the interface `Account`, is generated using an IDL compiler. Thus the type of the operational interface of the server object is known at compile time. But what if we did not know about the interface `Account` at compile–time? The only possible way to access the object in this case is to use CORBA’s *dynamic invocation interface* (DII). This interface to an ORB offers the possibility to invoke operation calls whose signature was not known at compile time. The following code excerpt shows the usage of the DII:

```
CORBA::Object_ptr obj = ...;
CORBA::Request_ptr req = obj->_request( "deposit" );
req->add_in_arg( "amount" ) <<= (CORBA::ULong) 100;
req->invoke();
```

Note that the variable `obj` is of type `Object_ptr` and not `Account_ptr`. The code fragment demonstrates how to model the operation call `acc->deposit( 100 )` from the code fragment above[1]. It does not require the `Account_ptr` client stub as in the last example. Despite the generic manner how the operation is invoked, the problem remains how to write a generic user interface to access CORBAs DII. Such an interface would allow a user to invoke arbitrary operations of *a priori* unknown interfaces. The next section gives a brief overview of the specific details of an operation invocation.

6.3 Anatomy of an operation declaration

Section 3.10 of the CORBA 2.2 specification describes the syntax of an *operation declaration* (see [5]). The syntax is part of the Interface Definition Language (IDL). The grammar presented in that section describes the syntax which induces a formal language. In figure 6.2 the anatomy of an operation declaration is given, using a graphical representation of the grammar where the arrows denote "consists of" relations. Thus, according to the CORBA standard, an operation declaration consists of a result type, an ordered list of parameters and so on. A parameter declaration itself consists of a directional attribute (`in`, `out` or `inout`), a parameter type and an identifier.

Note that the "graph" depicted in figure 6.2 already has some resemblance to a conceptual graph. We propose to model the information pertinent to an operation invocation through a CG. The anatomy of an operation declaration as depicted in figure 6.2 provides a hint on how to accomplish this task.

6.4 A generic DII interface

Just consider if we had an application which allowed the browsing of an interface repository. A user would find a suitable interface at *run–time* and decide to invoke operations without having to write a specific client object. What would be nice to have is a *generic client* which could cope with *a priori* unknown operational interfaces. As we have seen in figure 6.2 and from the discussion of the previous section, an *operation invocation* consists of the following elements:

- a name of the operation
- a return type
- an ordered list of actual parameters

With this "anatomy" of an operation invocation we can assemble a domain–specific conceptual and relational catalogue. We have developed such a catalogue which provides the "vocabulary" to express the information needed for the specification of an operation invocation. The conceptual graph depicted in figure 6.3 shows how to translate the operation invocation for `deposit( 100 )` using the DII (again concept nodes are denoted by white rectangles and relation nodes by black rectangles). As can be seen, a meta–notation

[1] Note that the code generated by the IDL compiler makes use of the DII interface

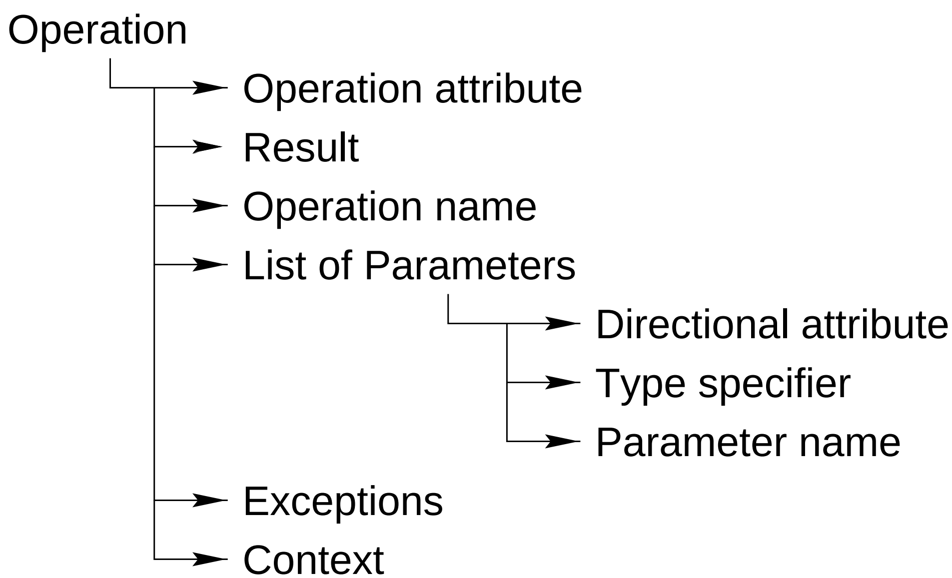

Figure 6.2: Syntax of an operation declaration.

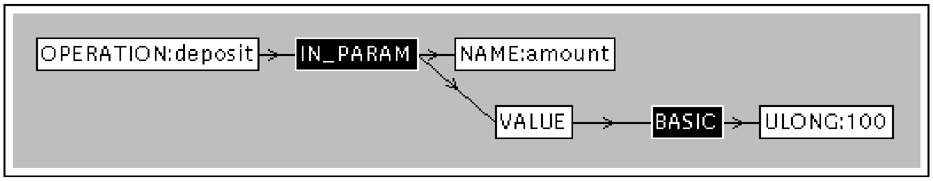

Figure 6.3: Conceptual graph representing the specification of the operation `deposit()`.

based on CG provides an easy readable, formal specification of an operation invocation. It should be clear that the CG template can be extended arbitrarily to cover the specifics of the CORBA–IDL like complex type definitions or sequences of arbitrary types.

6.5 Running the example

The MICO sources include an interactive conceptual graph editor written in Java. The sources of the example are located in the directory `mico/tools/ir-browser`. Note that you need the Java Developers Kit 1.1.5 as well as a parser generator for Java called JavaCUP (see chapter 2 on where to obtain these tools). We assume that you have succefully compiled the MICO sources contained in the aforementioned directory. Alternatively you can run the Java applet from your favorite WWW browser by visiting the MICO–homepage.
Two files in the `ir-browser` directory are of importance to run the example:

- `runproxy`: this shell script starts `diiproxy` and the interface repository. The IR server is then feed with some IDL's so you have something to browse.
- `dii.html`: a HTML page which makes reference to the main Java–class `DII` implementing the interactive interface repository browser.

In order to run the demonstration, you first have to run the shell script `runproxy`. You simply do this by starting it from an UNIX shell:

```
./runproxy
```

After this you can load the applet by either using a Java capable browser or the appletviewer tool which is part of the JDK. You can run the applet be running the following command from an UNIX shell:

```
appletviewer dii.html
```

Once the applet has been loaded, click on the button called *Start IR browser*. A new window opens. The right side of this window shows all top–level objects contained in the interface repository. For each object there is one icon. If you click on one of these icons using the left mouse button, the IDL source code of that object is shown in the left side of the window. You can "enter" an object using the right mouse button (this of course works only on container objects like interfaces or modules). If you press the right mouse button on an operation object, another window will open containing a conceptual graph representing this operation. You can change the input parameters of that CG before invoking it on an object.

Here is a short step–by–step tour:

1. click with the left mouse button on the *Account* icon
2. click with the right mouse button on the *Account* icon
3. click on the *deposit* icon with the right mouse button to invoke the `deposit()` method
4. click on the `ULONG:0` node while holding down the shift key, enter 100 into the appearing entry box and press return
5. use *Server/Invoke* to do the actual invocation
6. click on the *withdraw* icon with the right mouse button in the browser window to invoke the `withdraw()` method
7. click on the `ULONG:0` node while holding down the shift key, enter 20 into the appearing entry box and press return
8. use *Server/Invoke* to do the actual invocation
9. click on the *withdraw* icon with the right mouse button in the browser window to invoke the `withdraw()` method
10. use *Server/Invoke* to do the actual invocation
11. the rightmost node of the graph should change to `LONG:80`

HINT: If you move the pointer over a node of the graph the status line will show you the actions possible on this node. For example *Shift–Button1: edit* means: To edit the contents of the node press the left mouse button while holding down the SHIFT key.

6.6 Using the CG–editor

The CG–editor allows the insertion, editing and removal of nodes. The editor supports the following actions on conceptual graph nodes:

left mouse button
: If the working area was empty before this will insert a new root node, otherwise if you click on a node you can drag it around.

shift + left mouse button
: Edit the contents of conceptual graph node currently pointed at.

control + shift + left mouse button

Remove the node (and all its descendents) currently pointed at.

right mouse button

Bring up a context sensitive popup menu. Selecting an entry from it will add a corresponding subtree to the node currently pointed at.

Not all of the above functions work on all conceptual graph nodes. If you move the pointer over a node, the status line will show you the actions which are possible for that node.

The order of the child nodes of a conceptual graph node is determined by their Y–positions. The first child node is the one with the smallest Y–position (with Y–position increasing from top to bottom). So if you want to swap nodes A and B, just move A below B (if A was above B before).

The *Edit* menu offers you some functions which come in handy: *New graph* will delete the current graph, *Arrange graph* will layout the nodes of the graph currently being edited and *Linear from...* will show you the textual representation of the conceptual graph.

Chapter 7

LICENSE

This chapter contains the license conditions for MICO. All libraries are covered by the GNU Library General Public License (LGPL) version 2 or later, code generated by the IDL compiler is not copyrighted, everything else is covered by the GNU General Public License (GPL) version 2 or later.

The idea behind this is that MICO can be used for developing commercial applications without requiring the manufacturer of the commercial application to put the application under (L)GPL. On the other hand it is not possible to derive commercial applications from MICO without putting that application under (L)GPL.

Section 7.1 contains terms and conditions of the LGPL, section 7.2 contains terms and conditions of the GPL.

7.1 GNU Library General Public License

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well–defined independent of the application. Therefore, Subsection 2d requires that any application–supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine–readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine–readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine–readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side–by–side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty–free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Library General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE

COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

7.2 GNU General Public License

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine–readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine–readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty–free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix A

Frequently Asked Questions About MICO

Q: *During compilation my gcc 2.7.2.x dies with an "internal compiler error". What is going wrong?*

A: Some Linux distributions (noteably Suse Linux 5 and Red Hat) shipped broken gcc binaries. You have to recompile gcc 2.7.2.x, or better yet, install egcs 1.x or gcc 2.8.x.

Q: *I have installed gcc 2.8 or egcs, and it still dies with an "internal compiler error".*

A: You are encouraged to submit a bug report to the appropriate compiler's mailing list. In the meantime, disabling optimization usually works.

```
./configure --disable-optimize
```

Q: *During compilation gcc dies with a "virtual memory exhausted" error. What can I do?*

A: Add more swap space. Under Linux you can simply create a swap file:

```
su
dd if=/dev/zero of=/tmp/swapfile bs=1024 count=64000
mkswap /tmp/swapfile
swapon /tmp/swapfile
```

There are similar ways for other unix flavors. Ask your sys admin. If for some reason you cannot add more swap space, try turning off optimization by rerunning configure: `./configure --disable-optimize`.

Q: *I use Cygnus CDK and gcc dies with a "virtual memory exhausted" error. How to fix this?*

A: There seems to be a bug Cygnus CDK beta19 that prevents gcc from using swap space. The only workaround is to disable optimization by rerunning configure: `./configure --disable-optimize`.

Q: *I configured for namspace support but MICO doesn't compile?*

A: Earlier versions of gcc and egcs (up to gcc 2.8 and egcs 1.0) have very limited namespace support. The tests configure does to check for working namespaces pass, but MICO itself fails to compile. Rerun configure without `--enable-namespace`.

Q: *Why do MICO programs fail with a COMM_FAILURE exception when running on 'localhost'?*

A: Because MICO requires using your 'real' host name. Never use 'localhost' in an address specification.

Q: *MICO programs crash. Why?*

A: There is no easy answer (what did you expect?). But often this is caused by linking in wrong library versions. For example people often install egcs as a second compiler in their system and set PATH such that egcs will be picked. But that is not enough: You have to make sure that egcs' C++ libraries (esp. libstdc++) will be linked in. One way to make MICO use an egcs installed in `/usr/local/egcs` is:

```
export PATH=/usr/local/egcs/bin:$PATH
export CXXFLAGS=-L/usr/local/egcs/lib
export LD_LIBRARY_PATH=/usr/local/egcs/lib:$LD_LIBRARY_PATH
./configure
```

If that is not the cause you probably found a bug in MICO. Write a mail to `mico@vsb.cs.uni-frankfurt.de` containing a description of the problem, along with

- the MICO version (make sure it is the latest by visiting http://www.cs.uni-frankfurt.de/ mico/)
- the operating system you are running on
- the hardware you are running on
- the compiler type and version you are using
- a stack trace
- To get a stack trace run the offending program in the debugger:

```
gdb <prog>
(gdb) run <args>
program got signal ???
(gdb) backtrace
```

and include the output in your mail.

Q: *After creating Implementation Repository entiries with imr create imr list does not show the newly created entries. What is going wrong?*

A: You must tell `imr` where `micod` is running, otherwise imr will create its own implementation repository which is destroyed when imr exits. You tell `imr` the location of the implementation repository by using the `-ORBImplRepoAddr` option, e.g.:

```
micod -ORBIIOPAddr inet:jade:4242 &
imr -ORBImplRepoAddr inet:jade:4242
```

Q: *I'm using egcs 1.x. When I turn off MiniSTL compliation aborts with*

```
/usr/ccs/bin/as: error: can't compute value of an expression
involving an external symbol
```

A: This is a bug in the assember. One solution is to enable debugging:

```
./configure --enable-debug
```

The preferred solution is to install GNU as (in the binutils package). See also the discussion on the egcs FAQ (the `-fsquangle` option).

Q: *Why don't exceptions work on Linux?*

A: They do. You are experiencing a bug in the assembler. Upgrade to binutils-2.8.1.0.15 or newer and recompile MICO.

Bibliography

[1] ITU.TS Recommendation X.901 — ISO/IEC 10746–1: Basic Reference Model of Open Distributed Processing Part 1: Overview and Guide to the use of the Reference Model, July 1994.

[2] ITU.TS Recommendation X.902 — ISO/IEC 10746–2: Basic Reference Model of Open Distributed Processing Part 2: Descriptive Model, 1994.

[3] ITU.TS Recommendation X.903 — ISO/IEC 10746–3: Basic Reference Model of Open Distributed Processing Part 3: Prescriptive Model, February 1994.

[4] ITU.TS Recommendation X.904 — ISO/IEC 10746–4: Basic Reference Model of Open Distributed Processing Part 4: Architectural Semantics, 1994.

[5] Object Management Group (OMG), The Common Object Request Broker: Architecure and Specification, Revision 2.2, February 1998.

[6] A. Puder. Introduction to the AI–Trader Project. http://www.vsb.informatik.uni–frankfurt.de/projects/aitrader/, Computer Science Department, University of Frankfurt, 1995.

[7] A. Puder and K. Römer. Using a Meta–Notation in a CORBA environment. In *CORBA: Implementation, Use and Evaluation, ECOOP*, Jyväskylä, Finland, June 1997.

[8] J.F. Sowa. *Conceptual Structures, information processing mind and machine.* Addison–Wesley Publishing Company, 1984.