Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (01223) 515010**
INTERNATIONAL: **+44 1223 515010**
FAX: **+44 1223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Building DIMMA 2.0

Nicola Howarth

Abstract

This document describes how to build the DIMMA software, an implementation of the ANSA Distributed Interactive Multi-Media Architecture (DIMMA), and how to build applications that will run over DIMMA.

The tools required to build DIMMA are identified and the use of these together with the associated environment variables that must be set up prior to performing the build are described.

A description of the directory structure and how the associated files are used is provided for those who wish to understand how the process works.

APM.2036.01 **Approved** 10th July 1997

Request for Comments (confidential to ANSA consortium for 2 years)

Distribution:
Supersedes:
Superseded by:

Building DIMMA 2.0

Building DIMMA 2.0

Nicola Howarth

APM.2036.01

10th July 1997

The material in this Report has been developed as part of the ANSA Architecture for Open Distributed Systems. ANSA is a collaborative initiative, managed by APM Limited on behalf of the companies sponsoring the ANSA Workprogramme.

The ANSA initiative is open to all companies and organisations. Further information on the ANSA Workprogramme, the material in this report, and on other reports can be obtained from the address below.

The authors acknowledge the help and assistance of their colleagues, in sponsoring companies and the ANSA team in Cambridge in the preparation of this report.

APM Limited

Poseidon House
Castle Park
CAMBRIDGE
CB3 0RD
United Kingdom

TELEPHONE UK	(01223) 515010
INTERNATIONAL	+44 1223 515010
FAX	+44 1223 359779
E-MAIL	apm@ansa.co.uk

Contents

1 Introduction

This document describes how to build an implementation of the ANSA Distributed Interactive Multi-Media Architecture (DIMMA), and how to build applications which use DIMMA.

There are two aspects to the build process. The first is the “how to do it” approach, which is all that is needed for simple applications. The second is “why it is done this way”, which is helpful when building more complex applications.

Chapter 2 of this document explains how to build DIMMA 2.0, outlining the tools required, and the use of environment variables. After reading Chapter 2, you should be able to build DIMMA and to run the example programs. You should also be able to build simple client/server applications along the lines of the examples.

Chapter 3 provides further information as to how the DIMMA make system works. Read this if you want to understand the “why” behind Chapter 2.

Chapter 4 describes the directory structure in DIMMA: which files are where, and why they are there. It also describes in general terms what the file in each directory do. This information can be used to tailor your Makefiles for more complicated applications.

2 Building DIMMA and Simple Applications

2.1 What you need

DIMMA has been developed on the Solaris operating system, since this is both a popular environment, and more importantly provides an implementation of POSIX threads.

There are three tools necessary for building DIMMA:

1. Sparcworks “CC” C++ compiler. This was chosen as it supports both exceptions and templates. It is also used as a preprocessor for the IDL compiler.
2. GNU “gcc” C++ compiler. This is used to automate the dependency checking.
3. GNU “make”. This is a portable “make”, with advanced features such as VPATH, conditionals and string processing.

The source of GNU make (gmake) is distributed with DIMMA. If you already have a version of GNU make, then include it in your path. If not, then you will either need to obtain a copy from a GNU FTP mirror, or you can build the version supplied with DIMMA. Change directory to its location in the tools tree:

```
cd <dimma-top>/tools/src/gnu/gmake-3.74/src
```

and follow the installation instructions in file INSTALL. Typically, this will involve typing the following:

```
% ./configure
% make
% make binprefix=g install
```

This document assumes that the directory where GNU make is installed will be in your PATH.

Similarly you may need to built the GNU “gcc” C++ compiler and libraries. These are found under `gcc-2.7.0` and `libg++-2.7.1` under `tools/src/gnu`.

You now have all the tools necessary for building DIMMA. The next step is to set some environment variables.

2.2 Environment variables

2.2.1 DIMMA

Before building DIMMA, an environment variable must be set to the path where the DIMMA master directory is located. This variable is used by the makefiles, and should be set to the full pathname, with no abbreviations. One

way of doing this is to use the result of running pwd under the Bourne shell (sh) after first changing directory to the dimma root.

For example, if your normal shell is the C shell (csh), and the dimma master is /home/dimma, you will need to type:

```
% cd /home/dimma
% setenv DIMMA ‘/bin/sh -c “pwd”‘
% export DIMMA
```

or if your normal shell is bash, then you will need to type:

```
% cd /home/dimma
% DIMMA=‘/bin/sh -c “pwd”‘
% export DIMMA
```

If this sets DIMMA to an expanded path such as /net/host/disc/home/dimma, then that is the correct value; it should not be abbreviated to /home/dimma.

2.2.2 LD_LIBRARY_PATH

You will also need to set an environment variable to the location of the DIMMA libraries. This should be set as follows (here using sh - if using csh then use setenv as above):

```
% LD_LIBRARY_PATH=$DIMMA/<shadow>/lib
% export LD_LIBRARY_PATH
```

where <shadow> is the name of the shadow tree where the results of the build will go (see section 2.3).

Once you have set up these two environment variables, you are ready to build DIMMA.

2.3 Building DIMMA

The results of building DIMMA will go into a “shadow” tree under the dimma master directory. The name of this shadow directory will be ‘platform‘_build, where platform is a script in $DIMMA/tools/bin. On Solaris 2.5, for example, the shadow tree will be called “sun4_sos_5.5_build”. The structure of the shadow tree matches that of the source tree. The use of shadow trees keeps the source tree clean, and enables the use of a different shadow tree for each platform, and potentially for each user.

To build the IDL stub compiler, the dpe and other libraries and the examples, change directory to the dimma master directory and type:

```
% gmake
```

The shadow tree is constructed, matching the source tree. For example, the executables from $DIMMA/examples/Timing will be built in $DIMMA/<shadow>/examples/Timing.

The libraries are copied to <shadow>/lib, and the IDL compiler binary “jet_idl” is copied to <shadow>/bin.

If you wish to build DIMMA files in the same directory tree as the source files, and not use shadow trees, you can build with the command:

```
gmake BUILDDIR=.
```

When using this option, the libraries and IDL compiler are copied into sub-directories lib and bin respectively of ‘platform‘_build, for convenience.

If having built DIMMA you decide you want to put the built files somewhere else, then ensure that you clean the previous build, with

```
% gmake clean
```

or

```
gmake BUILDDIR=. clean
```

before rebuilding.

2.3.1 After building DIMMA

Once you have successfully built the complete DIMMA architecture, you may wish to install it on another machine of the same type, without re-building the entire system. The following steps should achieve this:

- Duplicate the directory structure of both source and shadow tree on the target machine.
- Copy all the files in the bin and lib directories in the shadow tree.
- Copy all the .hh files in the source tree.

You may also consider it worthwhile copying the examples directory (along with its sub-directories) in the source tree, as this provides useful instructional material and can be used to test that your trees are complete.

You may also wish to remove any unnecessary files from the original host machine. In the target tree, you can remove all .o files. In the source tree, you could remove all files except .hh files, but again you may choose to leave the examples.

2.3.2 Building a subdirectory

If you wish to build just one subdirectory, having already built the main DIMMA tree, then change directory to the required directory in the source tree, and run gmake as described above.

2.3.3 Other gmake targets

The following targets are supported:

- clean: removes all built files from the build directory and its subdirectories. Note that some files are not removed, in particular exported interface references.
- newstart: removes all the build files, and the build tree <shadow>. This should only be executed from the $DIMMA directory.
- buildtree: builds an empty build directory tree to contain built files. Normally you would never need to use this target, as the build directory tree is built by default in a normal build.
- depends: builds .d files with rules describing how build files depend on source files. Normally you would not need to use this since GNU Make should automatically build depend files if they do not exist when it tries to read them.

- templates: runs the IDL compiler to generate the template files _i.th and _i.tc without compiling. This is intended for use when developing a new application, and can only be used where the local Makefile includes examples.mk.

2.4 Building simple applications

This is fully described in [APM.2020] and hence only a brief overview is provided here.

Under $DIMMA/examples there is a collection of example programs which demonstrate features of DIMMA and of CORBA. Most of these will be built along with the DIMMA tree. Most of the examples generate two programs, for example in `$DIMMA/<shadow>/examples/Functional/Strings` you will find `echo_server` and `echo_client`. To run these you should run the server, then in a different window run the client.

In the source tree, you will find that each of the examples has a Makefile, and for simple applications you can model your own Makefiles on these. Most of the work for the Makefile is done in the file `examples.mk`, which is included into the Makefile, to enable the statements within the Makefile to be kept to a minimum.

A typical Makefile for a simple CORBA application called `echo` would look something like this:

```
ROOT := echo
CLIENT_OBJS := echo_client.o
SERVER_OBJS := echo_server.o $(ROOT)_i.o
include $(DIMMA)/config/examples.mk
```

ROOT defines the name of the IDL source file; currently only one IDL file is automatically handled in one directory/Makefile. In this example, the IDL file is `echo.idl`.

CLIENT_OBJS lists the source files which make up the client, other than those generated by the IDL compiler. Here there is a single source file for the client: `echo_client.cc`. Note that it is the `.o` file which is listed in the `Makefile`.

SERVER_OBJS lists the source files which comprise the server, other than those generated by the IDL compiler. Here there are two source files, `echo_server.cc` and `echo_i.cc`. Again it is the `.o` files which are listed in the Makefile. Note the use of `$(ROOT)` when listing the files.

Finally, the line including the `examples.mk` file must be inserted.

Note that although the IDL compiler generates a dummy `$(ROOT)_i.tc` for the server, the expanded version of this must be included explicitly. This is to provide greater flexibility in the use of file names on the server side. (The .tc file is a template implementation file - it must be copied to the source directory and renamed as .cc when the implementation is added).

The Makefile described above covers a simple application which generates a separate server and client. Colocation within a single capsule of client and server is also possible, and can be done without including the code needed for remote access. The standard examples.mk cannot be used, but must in part be included in the Makefile. An example is in $DIMMA/examples/Functional/Colocate/Makefile.

2.5 Building non-CORBA applications

For non-CORBA applications, stubs are handwritten, and not generated by an IDL compiler. This means that the Makefile cannot include examples.mk, but must include all the necessary steps directly. The result is effectively a simplified version of examples.mk. Two examples exist, in $DIMMA/examples/ODP/tiny_bank/Makefile and $DIMMA/examples/ODP/uni_flow/Makefile, and these can be used as templates for applications which do not use CORBA IDL. If many such applications are to be written, then it may be worth-while developing an “odp_examples.mk” to reduce the effort involved in producing each Makefile. Further information on the .mk files is given in section 3.1.

2.6 Building with ClearCase clearmake

If you do not use ClearCase, then you can ignore this section.

As with building using GNU make, this assumes that the DIMMA source and necessary tools have been installed, and that the environment variables have been set.

The Makefiles in the DIMMA source are GNU Make compatible. As it was not straightforward to pass the gnu compatibility flag recursively to clearmake, a further environment variable is required:

```
% setenv CCASE_MAKE_COMPAT gnu
```

or

```
% CCASE_MAKE_COMPAT=gnu
% export CCASE_MAKE_COMPAT
```

Shadow trees are used in the same way as with gmake.

To build the jet_idl compiler, dpe and other libraries and example programs, use

```
% clearmake
```

instead of gmake, as before.

To build DIMMA files in the same directory tree as the source files, the macro definition BUILDDIR=. must be put in a Build Options Specification (BOS) file, and not on the command line, that is:

```
% clearmake BUILDDIR=.# DOES NOT WORK
```

It is suggested that the environment variable CCASE_OPTS_SPECS is used so that clearmake will find the BOS file, e.g.

```
% setenv CCASE_OPTS_SPECS $DIMMA/ClearMakefile
```

where the view private ClearMakefile consists of the single line:

```
BUILDDIR=.
```

Again, if you change your mind about where to put the built files, ensure that you enter

```
% clearmake clean
```

before setting or unsetting `CCASE_OPTS_SPECS`.

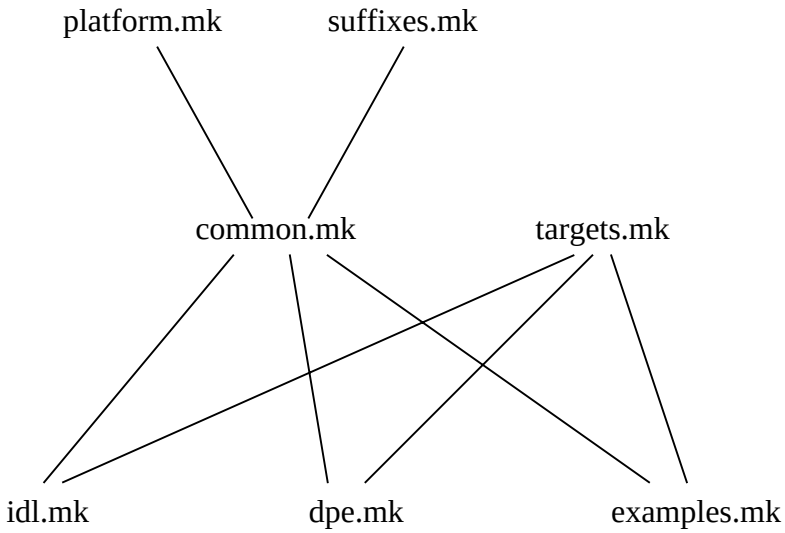
Beware of mixing GNU gmake and clearmake, as this can cause problems. It is recommended that the appropriate make clean command be issued before changing the make used.

3 DIMMA make - how it works

3.1 Makefiles in $DIMMA/config

In order to simplify the makefiles in each individual directory, the bulk of the work is specified out in files included by each makefile from `$DIMMA/config`. The hierarchy of these files in shown in Figure 3.1, where files lower in the hierarchy include those above them.

Figure 3.1: Hierarchy of .mk files



Each directory in the source tree has a makefile. The overall structure of each makefile is as follows:

- Define SUBDIRS to be recursively made (if undefined, every subdirectory containing a Makefile is used).
- Include $DIMMA/config/common.mk, which defines and documents common variables, such as SRCTREE, BUILDTREE, VPATH, SOURCES, OBJECTS, and common command sequences, such as visit-subdirs, make-from-builddir.
  - — Common.mk includes platform.mk, which defines platform-specific tools, such as the C++ compiler CXX, and variables such as OSV.
  - — Common.mk also includes suffixes.mk, which defines common implicit suffix rules for building objects and dependency files.
- Define some extra variables, or re-define existing variables, if required.

- Define a rule for the phony target ‘default’, for whatever is to be made by default (e.g. a library, subdirectories or some executable).
- Include $DIMMA/config/targets.mk, which defines rules for the standard DIMMA targets, such as clean, newstart and depends.
- Include dependency makefiles (.d files) if applicable.

The resulting makefile for $DIMMA/dpe and all its subdirectories is $DIMMA/config/dpe.mk, which is included by all the Makefiles in the dpe directory and all its subdirectories. The default behaviour is to build a library from the object files in the current source directory, then build the subdirectories.

The resulting makefile for all the subdirectories of $DIMMA/Jet-idl is $DIMMA/config/idl.mk, which is included by all the Makefiles of these subdirectories. The default behaviour here is to build a library from the object files in the current source directory.

The resulting makefile for most example directories under $DIMMA/examples is $DIMMA/config/examples.mk. This is written to build a client and server executable with certain assumptions about the source files, for example that they contain a single IDL file with all the IDL code.

3.2 Other Makefiles in the DIMMA source tree

- $DIMMA/Makefile. This is the top level makefile. The default behaviour is to build the jet_idl compiler from sources in $DIMMA/Jet-idl, then to build the dpe libraries from source in $DIMMA/dpe, then to build the example programs from source in $DIMMA/examples.
- $DIMMA/Jet-idl/Makefile. This is the makefile which builds the IDL compiler. The default behaviour is to build libraries from source in the subdirectories, then to build the jet_idl compiler.
- $DIMMA/examples/Makefile and other similar ones below it. Example directories which only contain other directories and have no code, for example Functional, have the default behaviour to build some or all of their subdirectories.
- $DIMMA/examples/ODP/tiny_bank/Makefile and $DIMMA/examples/ODP/uni_flow/Makefile. These are similar to examples.mk, but are simpler because the stubs for these examples (which do not use CORBA) are hand-written.

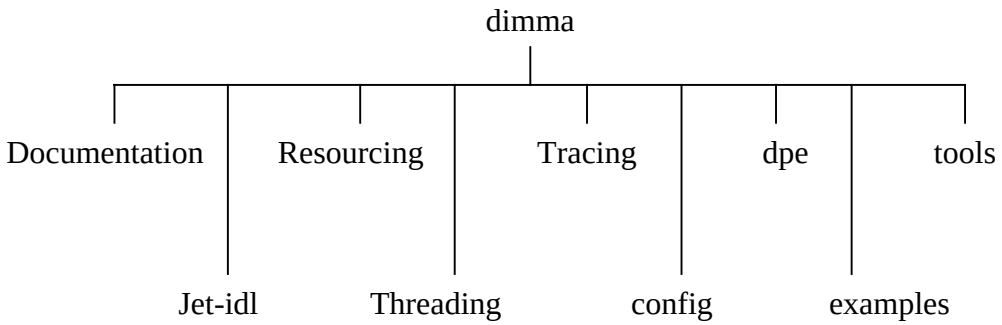
4 DIMMA directory structure

This section briefly outlines the hierarchy of the DIMMA source tree, with a brief description of each directory.

In general, the object files in each subdirectory are assembled into a shared library, so that there is a library for each subdirectory. The subdirectories are organised so that an application need only include the libraries it requires, for example the code for the Flow protocol and for the IIOP protocol are in different subdirectories, generating separate libraries, and it is only necessary to link with the appropriate protocol library.

The outline of the DIMMA tree is shown in figure 4.1. Each subdirectory is then described, together with expanded sub-trees where appropriate.

Figure 4.1: DIMMA structure - top level



4.1 Documentation

This directory holds postscript versions of the documents associated with DIMMA (such as this one).

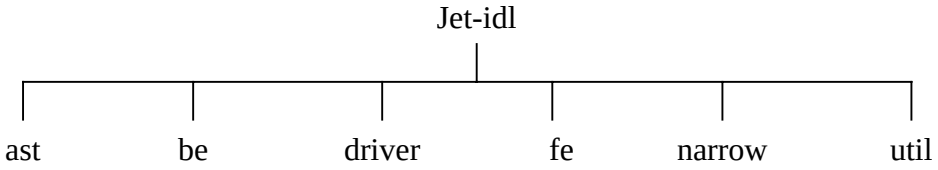
4.2 Jet-idl

This directory holds the source of the CORBA stub compiler. It's structure is shown in figure 4.2.

Each of the six subdirectories holds the source of a particular functinal area of the compiler, and a shared library is generated from each. These are then linked to build the jet_idl executable which is used to compile CORBA IDL files.

Header files which are common to more than one subdirectory appear in the top level of the Jet-idl directory.

Figure 4.2: DIMMA structure - Jet-idl



The subdirectories comprise the following:

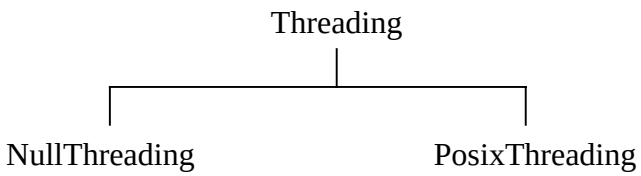
- ast: constructors, destructors and methods for generating the abstract syntax tree.
- be: the back end produces the C++ code from the ast, including client and server stubs and so on.
- driver: controls each stage of the compiler's operation.
- fe: the front end parses the IDL file and generates the ast.
- narrow: generic narrowing methods.
- util: utilities used by the other parts of the compiler.

4.3 Resourcing

This directory holds the generic source code which controls dynamic resource allocation for DIMMA applications.

4.4 Threading

Figure 4.3: DIMMA structure - Threading



The Threading directory holds the code which is common to all types of threads, while the subdirectories contain specific implementations such as Posix threads or null threading.

4.5 Tracing

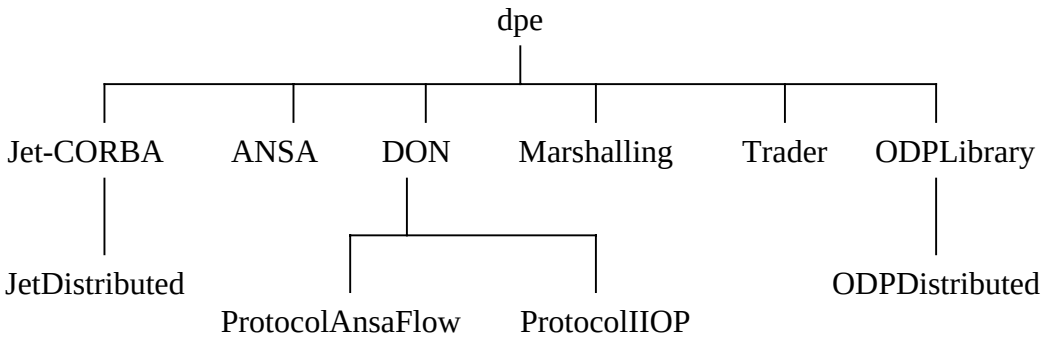
This directory holds the code for DIMMA's comprehensive tracing system, described in APM.1980 "DIMMA Tracing" and included in the Documentation directory.

4.6 config

The config directory holds all the .mk files which are included by the make system. These were described in an earlier section.

4.7 dpe

Figure 4.4: DIMMA structure - dpe



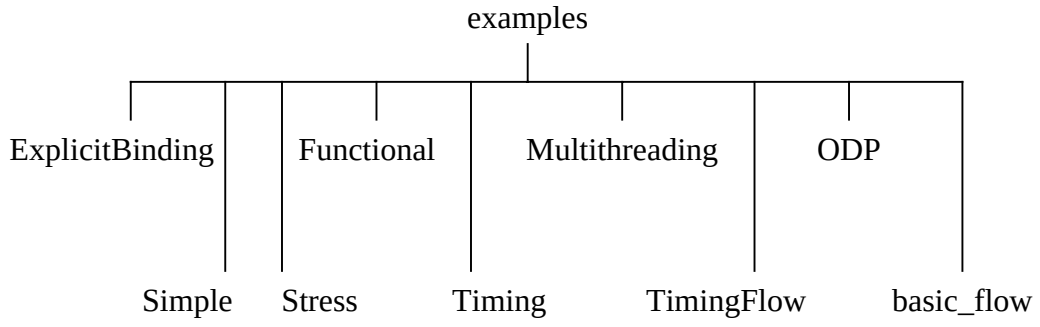
The dpe directory holds the major part of the dimma infrastructure. The main components are:

- Jet-CORBA: header files used by the rest of the CORBA API, and source code for exception classes which are also used in other areas of DIMMA.
- JetDistributed: the implementation of the distribution aspects of the CORBA API.
- ANSA: a mechanism for using ANSAware; this is not currently supported.
- DON: the dimma nucleus itself.
- ProtocolAnsaFlow: support for flow protocol.
- ProtocolIIOP: support for IIOP.
- Marshalling: marshalling classes.
- Trader: a simple trading “application” based on NFS.
- ODPLibrary: header files and some support code which is used outside of the ODP API.
- ODPDistributed: the implementatino of the distribution aspects of ODP.

4.8 examples

The examples directory contains example applications which demonstrate many of the features of DIMMA and of CORBA. Most of the subdirectories hold further subdirectories, but it is left to the reader to explore this area (see Figure 4.2)/

Figure 4.5: DIMMA structure - examples



4.9 tools

The tools directory holds the various tools provided with DIMMA, includingGNU make.

References

[APM.1995]

Macmillan, I. A., *An Introduction to DIMMA*; **APM.1995**, APM Ltd., Cambridge U.K., May 1997.

[APM.2020]

Howarth, N. J., *Writing a DIMMA Application*; **APM.2020**, APM Ltd., Cambridge U.K., June 1997.

[APM.1980]

Hayton, R. J., *DIMMA Tracing*; **APM.1980**, APM Ltd., Cambridge U.K., April 1997.

[APM.1994]

Macmillan, I. A., *DIMMA Design and Implementation;* **APM.1994**, APM Ltd., Cambridge U.K., June 1997.