CONTENTS

UKC ANSAWARE SURVIVAL GUIDE (FOR MODULA-3)

1 INTRODUCTON

So you need to write an ANSAware application in Modula-3 but don't know where to start? This document will try to break you in as gently as possible by explaining the concepts you need to understand and then showing you how ANSAware implements them. Although the concepts are general, the parts of this document that explain how to compile code are specific to UKC.

2 OVERVIEW OF DISTRIBUTED SYSTEMS

2.1 What is a Distributed System?

Ok, so you're used to one program which starts, runs and then stops. It contains procedures that perform some function and these are called in turn so that the program performs some required task.

In a "distributed system" we take a problem and we build a solution from several different programs that all talk to one another, when we can make these programs communicate over some network we are able to have each one run on a different machine. It is this that makes a distributed systems solution so useful; these programs may run on machines on other side of the room or the other side of the world.

Another important aspect of distributed systems is "relocation" — if one of our machines stops working then we might be able to start that part of the system on another machine somewhere else and carry on.

2.2 How Does ANSAware Help Build One?

ANSAware provides support for building client/server applications. A server capsule provides some "service" to the rest of the ANSA world, an example might be "I can tell you the temperature outside"; a client capsule will make use of this service by making a request to the server.

Here are two terms you will use a lot when talking about ANSA programs:

- An ANSA program is called a **capsule**.
- The request that a client makes to a server is implemented via a **Remote Procedure Call** (RPC).

Let's look at a normal piece of Modula-3 code; your program calls a procedure with some parameters and you may get some results back. An RPC is exactly the same except that the procedure is executed by the server which may be on another machine, ie. remote. You can pass parameters and get results back in just the same way, remember that passing pointers to data structures across machines isn't going to work!

This leaves us with two BIG issues to deal with:

- We now have all these server capsules that we can talk with and they can be running on different machines in several locations so how do we know what services are out there and how do we contact them?
- What do we do when our call to a server fails?

The second question will be answered later. Let's answer the first by considering the Yellow Pages.

I am a plumber, a good plumber who works at a reasonable rate and I know that everyone would like to use my plumbing service but first I need to tell the world that I exist. What do I do? I take out an advertisement in the Yellow Pages, the advertisement says I am a plumber and identifies a method of contact. Now someone with a leaking water pipe knows they need the services of a plumber so they can find me in the Yellow Pages and call me.

The ANSAware Trader is similar to our Yellow Pages. The Trader is an ANSA capsule that knows about all the visible services in our little world and how to contact the servers that provide them.

- When a server wishes to tell the world that it is around it **exports** a service offer to the trader. In our plumber example this occurs when I place my advertisement in the Yellow Pages.
- When a client wants o use a service it **imports** the location of the service from the Trader. What it gets back is a contact point for the service and can use this to talk to the server directly and perform the RPC.

Wildcards can be used so that the Trader performs a search and returns a list of matching services.

3 IMPLEMENTING THE CONCEPTS

When you write code for an ANSA capsule you write one or more modules of code which are then compiled and linked by the Modula-3 compiler. The best way to run several capsules is to run them in different windows on an X-Terminal.

Before we continue we need a little more terminology:

- Each server capsule has at least one **interface**. This is the point of contact with the client capsules.
- Each interface can support some number of **service operations**. Operations are usually grouped together using some criteria and each group appears on a different interface, one capsule can have several interfaces.

Each interface is represented by a Modula-3 ANSA Network Object. A client asks the Trader for a handle to a network object that provides a particular service and then invokes methods on the object. A server creates an object that inherits from the network object associated with the service it wants of provide and `OVERRIDES` the methods.

See [Har92a] for more details of Modula-3 objects.

3.1 Interface Description Language

The Interface Description Language (IDL) describes the operations available on each Ansa interface. There is at least one IDL file for each interface - if you are writing a client, read the IDL file for the interface you need to talk to to see what operations there are.

If you are writing a server then you must write an IDL file to describe the operations it will implement. You can also create your own types to make the operation signatures clearer.

You should see [APM93], section 3.1, for details of the IDL syntax.

3.2 stubm3 - the IDL Compiler

`stubm3` is the Modula-3 stub compiler. This takes IDL files and produces Modula-3 modules to implement the interface definitions.

For each `<interface>` in the IDL file being compiled, `stubm3` produces the following files:

- `<interface>.i3` - the M3 network object interface.

- `<interface>CRPC.{i3,m3}` - the client-side stubs.
- `<interface>SRPC.{i3,m3}` - the server-side stubs.

See [stu] for a full description, To write code, you will need to know how the stub compiler maps the IDL types onto Modula-3. These mappings are given in Appendix A.

Here follows a descrption of the functions provided by the gererated stubs:

Server-side Stub Routines

- ```
  PROCEDURE Export(
    ref: <interface>.T;
    intfSpecName: TEXT := "<interface>";
    context:      TEXT := "/";
    propList:     TEXT := "";
    concurrency:  CARDINAL := 16;
    typeTagged := FALSE)
    RAISES {Ansa.Failure};
  ```

`Export` registers the Ansa Network Object `ref` with the Trader in the associated trading context and with the supplied properties. See [APM93], section 3.11 for more details.

`concurrency` specifies the number of requests to the interface that will be queued while one is being processed.

If `intfSpecName` is not registered as a offer type with the Trader then the server capsule will fail with an error along the lines of:

```
(./plumber) :: warning, file '../capsule/src/generic/trading.c': line 104
(./plumber) :: capsule 1857 WARNING:
     binder_export - trader error 'unknownType': 1027 (bindFailure)
```

- `PROCEDURE Withdraw(ref: <interface>.T) RAISES {Ansa.Failure};`

`Withdraw` removes the specified Ansa Network Object from the Trader. It should be called when you no longer wish to provide the service ie. at server termination time.

Client-side Stub Routines

- ```
  PROCEDURE Import(
    intfSpecName: TEXT := "<interface>";
    context:      TEXT := "/";
    constraints:  TEXT := "";
    typeTagged         := FALSE): <interface>.T
    RAISES {Ansa.Failure};
  ```

`Import` searches the Trader for a handle to an Ansa Network Object which provides the specified interface, is in the specified trading context and that matches the given constraints.

- `PROCEDURE Discard(ref: plumber.T) RAISES {Ansa.Failure};`

`Discard` frees up any resources taken by the binding to the specified Network Object. This should be done when the binding is no longer required, ie. at client close-down time.

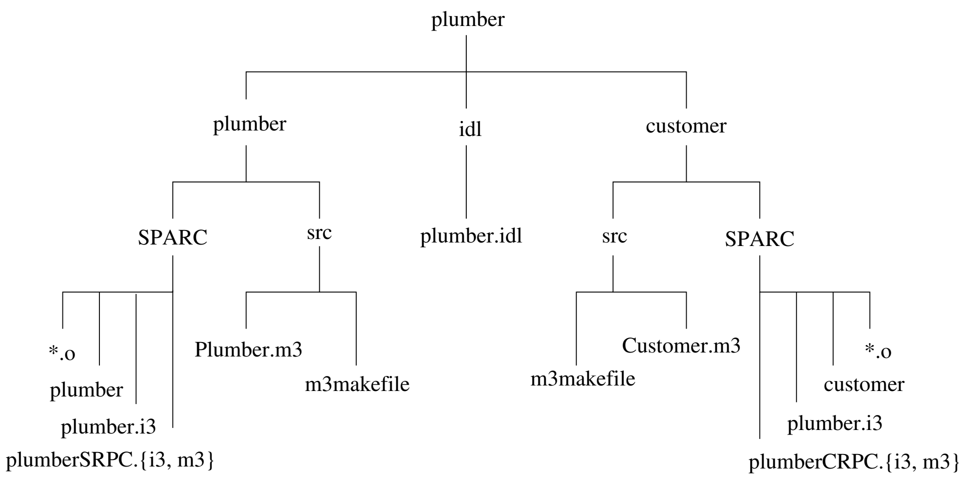

Figure 1: Directory layout for example programs

3.3 Building Applications with m3build

We use the `m3build` command to start the Modula-3 compiler running. `m3build` expects all source code to be in a directory called `src`, any object code and the finished executable are placed in a directory at the same as `src`, whos name is determined by the platform being built for.

We will often need to build a number of capsules so the best way to lay them out is shown in Figure 1.

`m3build` should be invoked in the directory that contains the src directory for the capsule you want to build. When started, `m3build` reads the commands in `src/m3makefile` to decide how to construct the capsule.

m3makefiles are written in a procedural language called Quake. There are a number of predefined procedures which allow you to build straight forward Modula-3 applications, see [m3m] for details. m3ansa provides a number of extra procedures for building Ansa applications (see Appendix B).

`m3build` builds the program, `m3build clean` removes all code derived from the build.

4 BUILDING AN APPLICATION ONE STEP AT A TIME

The easiest way to explain how to build an application is to go through a simple example one step at a time. We will build a plumber service who can give quotes for jobs and can be called out by a customer capsule which we shall also build[1].

The stages that we go through to build our application can be summarised by this list:

1. Configure our shell for an ANSA session.
2. Write an interface definition for our service.
3. Copy the default m3makefile and modify it for our interface.
4. Write the server capsule.
5. Flesh out the service routines.
6. Write the server body.
7. Write the client.

[1] The code for this example can be found in /proj/ansa/examples/m3ansa/plumber.

8. Build our application.

There is a logical order to these steps, use the above list as a reference when you come to writing your own programs. The following sections will explain each of these steps in detail.

4.1 Configuring the Shell for an ANSA Session

The ANSA system relies on certain environment variables being set so you must configure the shell for an ANSAware session before starting. If you use the csh shell then add the following lines to your `.cshrc` file:

```
setenv ANSA_VERSION m3
alias ansa source /proj/ansa/.ANSAwarerc
```

Once this has been sourced you can type `ansa` to set up all the library paths and other things that are necessary but which you really don't want to know about. At present Modula-3 is only installed on larch and mango so configuring your shell for an m3ansa session on any other machine will fail.

4.2 Defining the Interface for our Service

We need to tell ANSAware what operations are available on each interface, for this we use an Interface Description Language (IDL). There is at least one IDL file for each interface - if you are writing a client, read the IDL file for the interface you need to talk to to see what operations there are.

If you are writing a server then you must write an IDL file to describe the operations it will implement. You can also create your own types to make the operation signatures clearer.

Here is the IDL file for our plumber service (`plumber.idl`):

```
plumber :       INTERFACE =
BEGIN

-- Defines a simple plumber service
--              Ian Buckner - 13/03/95

--       Operation signatures

  -- client supplies the problem to the plumber who sends a quote back
  GetQuote : OPERATION [ problem : STRING ]
               RETURNS [ quote : CARDINAL ];

   -- supply your address to the plumber, he tells you if he can come out
   CallOut  : OPERATION [ address : STRING ]
                RETURNS [  booked_ok : BOOLEAN ];

  -- put the plumber out of business
  Sack : OPERATION [ ]
                        RETURNS [ ];

END.
```

We see that there are three operations, two of which take one parameter and return one result, see [APM93], section 3.1, for details of the IDL syntax. It is a good idea to comment what each operation will do to avoid searching through the server source when you forget!

4.3 Copying the Default M3makefile

The m3makefile explains how to build your ANSAware capsules, it specifies which files will be used in the build. Take a copy of `/proj/ansa/templates/m3makefile` which we will modify for our application.

See [m3m] for more details on writing m3makefiles.

4.4 Writing the Server

The first thing we need to do is update our `m3makefile` to indicate which interfaces our server will provide. As `plumber.idl` is held in a different directory, we use:

```
import_ansa_server("../../idl", plumber)
```

The main body of our server will be in `Plumber.m3` so we add:

```
implementation(Plumber)
```

Now let's look at how `Plumber.m3` is put together. All m3ansa capsules import the Ansa module which contains the Ansa type definitions. As we are a server of the plumber interface, we import the plumber module and associated module that contains the Server Remote Procedure Calls too:

```
IMPORT Ansa;
IMPORT plumber, plumberSRPC;
```

The next thing to do is to `OVERRIDE` the methods for the object associated with the service that we are providing - the plumber object in this case:

```
TYPE
  T = plumber.T OBJECT
      OVERRIDES
        GetQuote := DoGetQuote;
        CallOut  := DoCallOut;
        Sack     := DoSack;
      END;
```

We now write the service routines which will get called when a client makes a request on our plumber interface. Here is the code for the GetQuote operation:

```
PROCEDURE DoGetQuote (<*UNUSED*> self: T; problem: TEXT):
  plumber.GetQuoteResult_ RAISES {} =
  VAR res: plumber.GetQuoteResult_;
  BEGIN
    (*
     * the quote is completely unrelated to the work that needs doing
     *)
    WITH rand = NEW(Random.Default).init() DO
      res.quote := rand.integer(5, MAX_BILL);
    END;
    Say("asked for a quote on " & problem & ", reckon it will cost $"
          & Fmt.Int(res.quote));
    RETURN res;
  END DoGetQuote;
```

Two things to note about service methods are:

- The first parameter is always of type T and is a reference to the object that owns the method, the remaining parameters are the ones declared in the idl definition of the operation.
- The return type is always of type `<interface>.<operation>Result_`

Now, to create our service object, we create a variable of the service object type:

```
VAR
  plumberRef     : T;
```

Then we create an instance of the object:

```
plumberRef := NEW(T);
```

and tell the rest of the world about the service by exporting it to the Trader:

```
plumberSRPC.Export(plumberRef, context := CONTEXT);
```

`CONTEXT` is the trading context for this offer. You should have your own context space to work in and only import from other spaces if you are talking to servers not owned by you. Now we just provide the service until close down when we call:

```
plumberSRPC.Withdraw(plumberRef);
```

which will remove the offer from the trader.

The plumber example shows how the server can simply loop until it is asked to terminate. The application is prevented from using excessive CPU by making calls to

```
Thread.Pause();
```

Which puts the main capsule thread to sleep and allows other threads to run (see later).

4.5 Writing the Client

We shall define our client behaviour in `Customer.m3`. Firstly, we update the `m3makefile` to show that we will be a client of the plumber interface:

```
import_ansa_client("../../idl", plumber)
```

Two more lines have to be added to tell `m3build` where the main body of code is held and which program to build:

```
implementation(Customer)
program(customer)
```

This time our program is a client so we will need to import the plumber module and the client stub routines:

```
IMPORT Ansa;
IMPORT plumber, plumberCRPC;
```

To invoke operations on a server object, we first declare a variable of the server object type:

```
plumberClient: plumber.T;
```

We then perform an `Import` operation to bind us to the server:

```
plumberClient := plumberCRPC.Import(context := CONTEXT);
```

Remembering that result parameters are always of a type `<interface>.<operation>Result_` ie:

```
reply        : plumber.GetQuoteResult_;
book_reply   : plumber.CallOutResult_;
```

We can call operations on the server object:

```
reply := plumberClient.GetQuote(PROBLEM);
```

When we no longer need the server, we can release the binding:

```
plumberCRPC.Discard(plumberClient);
```

5 EXCEPTION HANDLING

Invoking a server method or calling one of the client/server stub routines can fail for one of several reasons. The server method may fail because the server is no longer running or because it is unreachable due to a network problem. We need to be able to trap failures when they occur so that we can take appropriate action.

Failure recovery is a very important part of writing distributed applications. We must make our code as robust as possible so that if part of our system fails, we can still continue.

The Modula-3 exception mechanism is used to indicate failure. Server methods and client/server stub routines raise the `Ansa.Failure` exception on failure and pass back a reason. The following extract of code shows how we might trap an exception caused by an Export failure:

```
TRY
  plumberSRPC.Export(plumberRef, context := CONTEXT);
  EXCEPT
  | Ansa.Failure (stat) =>
    Wr.PutText(Stdio.stderr, "Export failure. Reason: " & stat & "\n");
    Wr.Flush(Stdio.stderr);
    RETURN;
END;
```

In this example we simply return back to the calling procedure. Rather than do this, we might raise another exception for the calling procedure to pick up.

See [Har92b] for further information on exception handling in Modular-3.

6 PROGRAMMING WITH THREADS

Threads are often used in distributed applications to service a number of requests simultaneously or to perform background operations.

Some mention needs to be made of Modula-3's thread package, `Thread`. M3 used a pre-emptive sheduler so one thread can not monopolise on the processor. The thread package supports mutual exclusion to provide event synchronisation too.

See [Har92c] for a full description of M3 threads.

7 DEBUGGING

The best way to debug your programs is to run them through a debugger which will let you step through the code and print out variable contents.

The best program for the job is m3gdb, a modified version of gdb, the GNU debugger, which supports Modula-3. Copy the gdb configuration file to your home directory.

```
cp /proj/ansa/m3ansa/config/gdbinit ~/.gdbinit
```

m3gdb should be started in the build directory of the program you want to debug, in our example, to debug the customer program:

```
mango% cd example/plumber/customer/SPARC
mango% m3gdb customer
```

You nee to tell m3gdb that you will be debugging Modula-3:

```
(gdb) set la m3
```

Functions can be displayed with the `list` command. Procedures must be tied to the module they are declared in:

```
(gdb) list Customer.Say
```

- which will display the function if the file is on the current search path, this can be added to with the `directory` command:

```
(gdb) directory ../../idl
```

The gdb debugger is very powerful and can not be fully explained here. Read [m3g] and [gdb] for a more complete explanation. Gdb also has good help facilities, just do

```
{gdb} help
```

and follow the instructions. You might also want to consider running m3gdb inside GNU Emacs to debug your programs. This will give you a source and a debugger window and will highlight the current line as you step through your code.

8 TRCONTROL - A USEFUL TOOL

Trcontrol is an X-based application that allows you to visually see the services registered with the Trader, as well as add and delete service types and trading contexts. The application is started by typing `trcontrol`; Figure 2 shows the application window.

The main screen of Trcontrol is divided into seven areas. At the top is the message window showing the previous operations executed by Trcontrol, below this is an input bar into which you can type. Next are two large buttons which refresh the display and show the details of the Trader being managed.

The next row of buttons perform operations on the trading contexts that are shown in the window below. The trading contexts are hierarchical and ones ending in "/" can be expanded to reveal lower levels. All normal ANSAware users will have a trading context below "/ansa/users/" or "/ansa/groups/", followed by their login — contact the ANSAware administrator if you do not have one.

Once a trading context has been expanded, you will see a list of the service offers which have been exported in this context. Once you have highlighted an offer, the operations on the "Offers" button can be used to delete it or view it's details. Deleting services is useful for removing stale offers - when a server has died but has not withdrawn it's offer from the trader. Importing and using this service will fail as the server is no longer around.

The bottom part of the Trcontrol display is a button bar and a hierarchical display of the types registered with the trader. You can expand and move around the type hierarchy in the same way as you do in the context window. The "Add Type" option on the "Type" menu can be used to insert a type in the part of the hierarchy that is currently highlighted.

One word of warning, there is no sense of ownership on offers, contexts or types which means you are free to delete any ones you want. Be sensible and do not go around deleting other peoples offers or trading contexts as this is VERY annoying!

9 WHERE DO I GO IF I WANT TO KNOW MORE?

Configuring your shell for an ANSA session also makes some extra manual pages available. Many of the ANSA routines are documented and these pages help supplement the written material. It is probably worth playing around with `man -k` and seeing what you can find.

We have copies of the ANSAware manuals. Be warned however that these have been found to be incomplete and contain errors - don't take them as gospel. *Application Programming in ANSAware* is the reference manual for the language - three copies are available at Computing Reception but these can not be removed from the building. *Systems Programming in ANSAware* deals with the ANSAware implementation and is very low level, it will be of use if you need to play around with the ANSAware internals. The *Systems Managers Guide* explains how to install ANSAware and its day to day maintanance and *An Overview of ANSAware 4.1* describes the ANSAware approach and tries to cover the changes between versions.

UKC has been using ANSAware on research projects for several years now, many bugs have been fixed and sleep lost in the process. Most of the Networks and Distributed Systems Research Group have

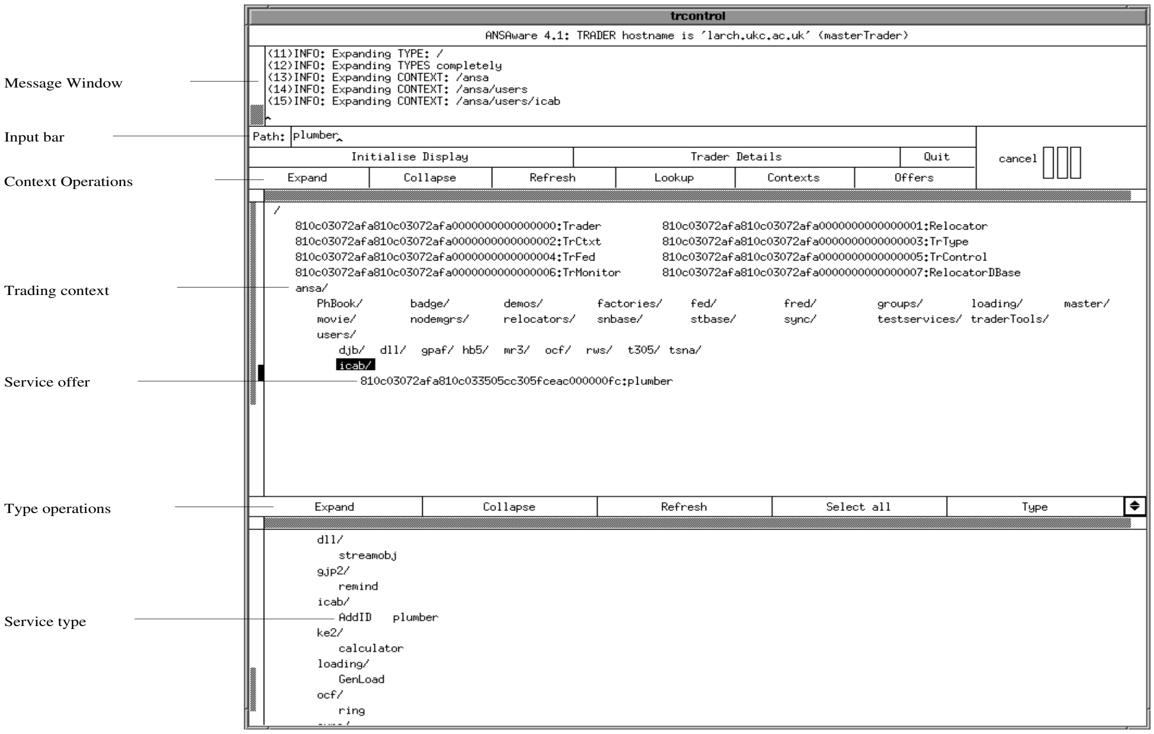

Figure 2: Trcontrol Display

used ANSAware and there is a lot of in-house expertise. Your initial port of call should be the current ANSAware administrator [2] who will be able to help with most problems.

[2]Email ansa@ukc.ac.uk

References

[APM93] APM. Extracts from the ANSAware reference manual. 1993.

[gdb] `gdb(1)` - on-line manual page for the GNU Debugger.

[Har92a] Samuel P Harbison. *Modula-3*, chapter 11 - Objects. Prentice Hall, 1992.

[Har92b] Samuel P Harbison. *Modula-3*, chapter 7 - Exceptions. Prentice Hall, 1992.

[Har92c] Samuel P Harbison. *Modula-3*, chapter 12 - Threads. Prentice Hall, 1992.

[m3g] http://stork.ukc.ac.uk/computer_science/Modula-3/html/modula-3/html/debugging.html.

[m3m] http://stork.ukc.ac.uk/computer_science/Modula-3/html/modula-3/html/m3build/m3makefile.html.

[stu] `stubm3(1)` - on-line manual page once configured for an m3ansa session.

APPENDIX A

IDL TO M3 TYPE MAPPING

When the stub compiler generates modules from an IDL file, it must translate the IDL types into their Modula-3 equivelents.

The basic IDL types are mapped with the same typenames as in IDL:

```
Boolean        = BOOLEAN;
ShortCardinal  = BITS 16 FOR [0..16_FFFF];
Cardinal       = Word.T;
LongCardinal   = Word.T;
ShortInteger   = BITS 16 FOR [-16_8000..16_7FFF];
Integer        = INTEGER;
LongInteger    = INTEGER;
Real           = REAL;
LongReal       = LONGREAL;
Octet          = CHAR;
Char           = BITS 8 FOR [0..16_7F];
String         = TEXT;
```

The constructed types are shown in Table 1.

IDL Type	*M3 Type*
`enum : TYPE = {a,b};`	`TYPE enum = a,b;`
`arr : TYPE = ARRAY n OF elt;`	`TYPE arr = ARRAY [1..n] OF elt;`
`seq : TYPE = SEQUENCE OF elt;`	`TYPE seq = REF ARRAY OF elt;`
`rec : TYPE = RECORD [f : ft, ...];`	`TYPE rec = RECORD f : ft; ... END;`
`iref : INTERFACEREF OFTYPE Foo;`	`TYPE iref = Foo.T;`

`chce : TYPE = CHOICE enum OF {`
`a => at,`
`b => bt`
`}`
	`TYPE`
	`chce = BRANDED OBJECT END;`
	`chce_a = chce BRANDED OBJECT f: at END;`
	`chce_b = chce BRANDED OBJECT f: bt END;`

Table 1: Mapping of constructed types.

The Modula-3 `TYPECASE` operation should be used to discriminate choice variants.

APPENDIX B

M3MAKEFILE MACROS

These are the extra macros for when you need to specify Ansa sources in you `m3makefile`. Add the following line first:

```
include("/proj/ansa/m3ansa/etc/include/ansa_macros.tmpl")
```

Here is a description of the procedures you will then be able to use:

- `ansa_interface(AnIDLIntf)`

The M3 program is interested in the types in `AnIDLIntf.idl`, which is in the source directory (the one containing this `m3makefile`). The corresponding M3 interface, `AnIDLIntf.i3`, will be generated in the same directory as the object files for this program, and is added to it's list of sources. No stubs will be generated.

- `ansa_server(AnIDLIntf)`

The M3 program is a server for the service defined in `AnIDLIntf.idl` in the source directory. The Network Object interface and the server stub module will be generated and added to this program's sources.

- `ansa_client(AnIDLIntf)`

The M3 program is a client of the service defined in `AnIDLIntf.idl` in the source directory. Generates `AnIDLIntf.i3` and the client-side stubs.

- `ansa_client_server(AnIDLIntf)`

The M3 program is both a client and a server for the service defined in `AnIDLIntf.idl` in the source directory. Generates `AnIDLIntf.i3` and both client and server-side studs.

What about IDL interfaces outside the source directory for this program?

For these, prefix the above functions with `import_` and insert the pathname to the IDL file as the first parameter, ie:

- `import_ansa_client(/a/path/name, AnIDLIntf)`

If you generate M3 interfaces or stubs from an IDL interface which `NEEDS` (ie. imports) another IDL interface which is not in the same directory, you need to supply a '`-I /the/other/dir`' argument to `stubm3`. The macros above include the value of the `m3makefile` variable `IDLDEFPATH` in their invocations of `stubm3` for this purpose, to be used as follows:

```
IDLDEFPATH = "-I /the/first/remote/dir -I /another/remote/dir"
import_ansa_interface (/the/main/dir,AnIDLIntf)
```

Don't forget about the template, `/proj/ansa/templates/m3makefile` which should be copied and modified for your capsules.