Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

TELEPHONE: **Cambridge (0223) 323010**
INTERNATIONAL: **+44 223 323010**
FAX: **+44 223 359779**
E-MAIL: **apm@ansa.co.uk**

ANSA Phase III

Type Conformance and IDLs

Yigal Hoffner, Mike Beasley

Abstract

This paper starts by considering what agreement is necessary to achieve interworking in each of the five projections, and goes on to contrast abstract and concrete interface definitions, and to compare existing Interface Definition Languages (IDLs).

APM.1042.00.01 **Draft** 16 August 1995

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

Distribution:
Supersedes:
Superseded by:

Type Conformance and IDLs

Type Conformance and IDLs

Yigal Hoffner, Mike Beasley

APM.1042.00.01

16 August 1995

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 Architecture Projects Management 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.

Architecture Projects Management 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	**1**	**Interworking**
1	1.1	Introduction
1	1.2	Type systems in a distributed system
1	1.3	Federation of type systems
2	1.4	Interworking in the computational projection
2	1.4.1	Interaction conformance
2	1.4.2	Interface type conformance
3	1.5	Specifying interfaces in the computational and engineering projections
3	1.6	Abstract data types (ADTs)
3	1.6.1	Interface definition languages (IDL)
3	1.6.2	Abstract interface definition languages (AIDLs)
3	1.6.3	Using ADTs
4	1.7	Concrete data types (CDTs)
4	1.7.1	Concrete interface definition languages (CIDLs)
4	1.7.2	CIDL syntax
5	1.7.3	CIDL language binding
5	1.7.4	CIDL and OTWF
5	1.7.5	RPC and interworking
5	1.8	Advantages of using ADTs in programming languages
6	1.9	Reverse engineered ADTs
7	1.10	ADTs and AIDLs, CDTs and CIDLs
9	**2**	**Review and examples of IDLs**
9	2.1	Review of various concrete IDLs
9	2.2	General Features of IDLs
9	2.2.1	Types
9	2.2.2	Type Constructors
9	2.2.3	Constants
10	2.2.4	Scoping
10	2.2.5	Interface References
10	2.2.6	Structure of Invocations
10	2.3	ANSAware IDL
10	2.3.1	Types
10	2.3.2	Examples of Constructed Types
11	2.3.3	Constants
11	2.3.4	Scoping
11	2.3.5	Interface References
12	2.3.6	Structure of Invocations
12	2.3.7	Example of an interface declaration
12	2.4	CORBA IDL
12	2.4.1	Types
12	2.4.2	Examples of Constructed Types

13 2.4.3 Scoping
13 2.4.4 Interface References
14 2.4.5 Structure of Invocations
14 2.4.6 Other features
15 2.4.7 Example of an interface declaration
15 2.5 DCE IDL
15 2.5.1 Types
15 2.5.2 Type Constructors
15 2.5.3 Examples of Constructed Types
16 2.5.4 Scoping
16 2.5.5 Interface References
17 2.5.6 Structure of Invocations
17 2.5.7 Example of an interface declaration
17 2.5.8 Other Interface Properties
17 2.5.9 Attribute Configuration Language
18 2.6 Conclusions from IDL survey

19 3 DPL compared with concrete IDLs
19 3.1 DPL
19 3.1.1 Introduction
19 3.1.2 Types
20 3.1.3 Type Constructors
20 3.1.4 Examples of constructed types
22 3.1.5 Scoping
22 3.1.6 Interface References
22 3.1.7 Structure of Invocations
22 3.1.8 Example of an interface declaration
22 3.2 Conclusions

25 4 IDL and inter-working
25 4.1 Interworking conformance checks
25 4.1.1 Levels of type checking
26 4.2 Two extreme worlds: passing references versus passing values
26 4.2.1 Passing references only
26 4.2.2 Passing values only
26 4.2.3 Trade-off between the extremes
27 4.3 Unresolved issues
27 4.4 Crossing domain boundaries

29 5 Appendix
29 5.1 Detailed examples
29 5.1.1 Implementation of an enumeration type
29 5.1.2 Implementation of a record type
29 5.1.3 Alternative definition of ‘choice’
30 5.1.4 Implementation of alternative ‘choice’
30 5.1.5 Avoidance of ‘choice’

1 Interworking

1.1 Introduction

One of the main goals of the ANSA architecture is to provide support for connecting together heterogeneous computer systems, and allowing them to interwork in a way which is transparent to the application programmer.

This document provides an investigation of the problems which arise in the context of multiple distributed platforms which support different:

- concrete IDLs
- different OTWFs

The problems involved with clients and servers defined using different IDLs and OTWFs where interworking is required:

- type checking
- translation (stubs and gateways)

Note: a discusiion of more general problems of conformance is also necessary - eg. static versus dynamic type conformance checking.

1.2 Type systems in a distributed system

A type system is a categorization of entities according to properties which they have in common [RC.258]. A type system which is based on conformance rather than equality and on the the encapsulation concept will support:

- implementation independent specification of interfaces
- trading, late binding and dynamic configuration
- service evolution
- the distributed development process

Note: above is in teh context of the same type system ie. here we are defining a type system boundary. Next section will specify how different type systems are characterized and hence how they can differ from each other. this also specifies what federation has to overcome.

1.3 Federation of type systems

In addition to the problems of type matching which arise in systems with a homogeneous type system, the following problems arise when federating systems with different type systems:

- means of specifying semantics - these can range from:
  - formal to informal
  - within to outside the system

- different computational models
- static versus dynamic type checking
- reflections of different engineering concerns:
  - not abstract enough to deal with engineering heterogeneity
  - different ways of representing the engineering concerns
- different syntax of ADT's
- different concrete IDLs
- different OTWFs

Additional difficulties arise when federating distributed systems. For example, federation makes type checking based on programmer asserted type relations impractical (because merging the type relationship graphs of two systems would require much human intervention) and poses the requirement for computing automatically the relationship between interface types [RC.258.00 92].

In addition, merging different type systems requires two sets of information:

- type system information about interfaces - in terms of interface signature
- information about service semantics: the semantic information about the entities being represented behind the interface. This is necessary in order to check that services actually do what clients intend them to do, as interface signatures can only ensure that there will be no interaction faults.

1.4 Interworking in the computational projection

Interworking in the computational projection entails two kinds of conformance: conformance to the interaction model and conformance of the interface type for potential interactions.

1.4.1 Interaction conformance

Interaction conformance requires interactions to follow the rules of an interaction model (for example, the interaction part of the ANSA computational model as described in [APM.1001.1.93]). This relates to things like the structure of invocations in terms of operations, arguments, terminations and results.

In cases where interaction models are different, translation may be used to bridge the gap between them.

1.4.2 Interface type conformance

Interface type conformance applies to potential interaction between clients and servers. An interface type conformance statement for a potential interaction is based on the “no surprise” principle which asserts that neither party to the interaction will attempt to interact in a way that the other does not expect. This subject is covered in [APM.1001.1.93], [RC.258.00 92] and [RC.339.01 92].

Although the problem of type conformance spans the entire range of projections [RC.258.00 92], this paper is mainly concerned with the computational and the engineering projections.

1.5 Specifying interfaces in the computational and engineering projections

Unless working within a homogeneous environment, an abstract representation of interfaces is essential as a starting point before engineering considerations cause designs and implementations to diverge. One possible way of describing interfaces is by using the abstract data types concept.

1.6 Abstract data types (ADTs)

ADTs are a specification of data entirely in terms of operations which can be performed on the data. It is a specification which abstracts from the concrete representation of the data.

The principles behind ADTs are [APM.1020.1 93], [APM.1001.1.93]:

1. the ability to use interface references for invocation
2. the ability to select an operation using these interface references
3. the ability to select an outcome (also referred to in literature as exceptions or terminations)
4. the ability to refer to other ADTs in operation and outcome parameters.

Some proponents of ADTs propose the inclusion of base types such as `INTEGER, REAL, BOOLEAN` within the ADT model. It can be shown that it is possible to construct all the necessary types by using the ability to select outcomes. Thus, the ability to distinguish outcomes without the need for base types is regarded as a more basic model. Also, with outcomes, the resulting computational model is symmetric between operations and outcomes.

1.6.1 Interface definition languages (IDL)

An interface definition language (IDL) is a language used to describe interfaces. We distinguish between concrete and abstract IDLs. Abstract IDLs (AIDL) are used to describe ADTs. Concrete IDLs (CIDL) are used to describe interfaces in terms of built-in (and constructed) types resulting in concrete data types (CDTs).

1.6.2 Abstract interface definition languages (AIDLs)

An AIDL is a language used to describe ADTs. Such a language must support:

1. ability to specify interfaces or closures (e.g. classes in C++)
2. referring to external interfaces from within interface definitions
3. multiple outcomes.

1.6.3 Using ADTs

The ADT concept can be used in three different ways:

1. to describe an idealized world where everything is done by passing interface references

2. to provide an abstract specification which can be implemented in different ways depending on the target environment and engineering trade-off
3. to ‘reverse engineer’ ADT from a CDT.

1.7 Concrete data types (CDTs)

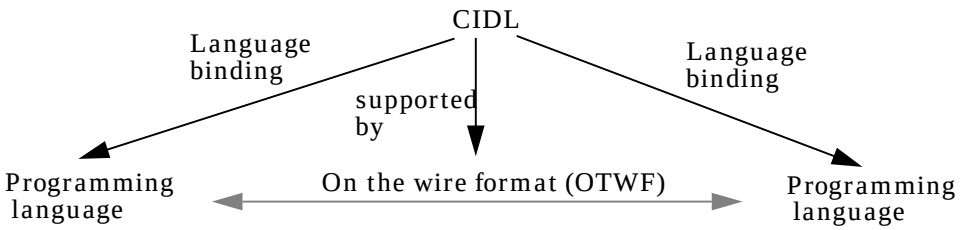
Like ADTs, CDTs are interface types which are defined in terms of the operations which can be performed on them. Unlike ADTs, the arguments and results must usually be of basic and constructed data types.

1.7.1 Concrete interface definition languages (CIDLs)

There are three issues concerning CIDLs (Figure 1.1):

1. syntax: what can be described and how by the CIDL
2. binding to specific programming languages such as C, C++. A CIDL description of an interface has to be translatable into at least one programming language
3. relation to “on the wire format” (OTWF): a presentation layer issue in terms of comms

Figure 1.1: CIDL, language binding and OTWF



The use of a CIDL to specify a refinement of an ADT in concrete terms introduces some engineering constraints on ADTs. CIDLs can be regarded as a compromise between ADTs which avoid a concrete representation of data, and programming languages which define the concrete representation in a specific environment.

1.7.2 CIDL syntax

CIDL syntax includes base types. Thus, notions of size and ranges are introduced, in terms of `SHORT INT, INT, LONG INT`, for example.

Some CIDLs include interface references as an explicit type. Although interface references may be composed of records of base types[1] they need to be treated as a special case because translating them from one CIDL to another involves more than just base type conversions. There is a need to take into account the semantics of interface references and their underlying engineering model (issues of relocation, binding, protocols, groups, etc. are relevant here).

1. This is true of ANSAware IDL, but not of CORBA or DCE, where the types ‘Object’ and ‘handle_t’ (respectively) are opaque.

1.7.3 CIDL language binding

CIDL types must be mappable to the client/server implementation language in terms of:

- base types and the operations which can be performed on them
- constructors: language facilities for specifying composite data structures such as arrays and records. Some CIDL languages also allow self referring data such as graphs
- interface references

1.7.4 CIDL and OTWF

On the wire format (OTWF) describes the data structures which are supported by the protocol and their format and encoding rules.

The OTWF has to support the set of base types and constructors chosen by the respective CIDL and vice versa. This mapping allows the automated creation of stub generators.

The exact way in which the information is formatted/encoded on the wire is not in itself important as long as both interacting sides agree on it. In practice this means that:

- the CIDL types can be mapped to the OTWF and vice versa
- the CIDL types can be supported by the environments of the interacting objects and their programming languages

There does not have to be a one to one mapping between CIDLs and OTWFs. A CIDL can be supported by a number of OTWFs but in practice, most CIDLs (such as DCE and ANSAware, but not CORBA) have an associated specification of a single OTWF.

The requirement for generating stubs in a particular environment is that the programming language used by a client or a server can support the structures defined by the CIDL and OTWF specification of the object's interface.

1.7.5 RPC and interworking

RPC support for multiple terminations, for example, depends on whether the IDL type system supports a symmetric model of invocation operations and terminations; if it does so, then the session protocol (request/reply) will necessarily do so.

1.8 Advantages of using ADTs in programming languages

There are a number of advantages to using ADTs in programming languages [APM.1020.1 93]. The advantages seem to lie in a number of areas: portability, type checking, interworking and evolution.

Portability: mapping onto different environments (Figure 1.2)

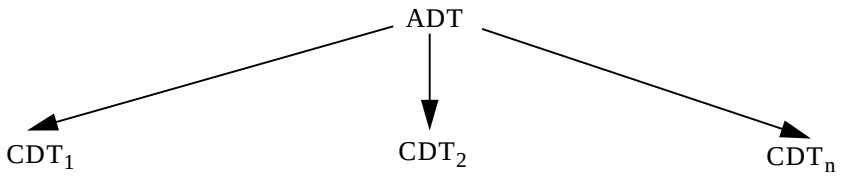
1. not being tied to a specific CIDL or OTWF means that the same program can be mapped to different environments without the need to change the program
2. facilitates the use of multiple OTWFs in the same object through use of multiple stubs, thus allowing interworking between different environments

3. because of the higher level of abstraction, ADT definitions are more suitable for transformations to add quality of service and transparencies, for example transactions, groups and migration
4. when transforming between a large number of CIDLs, fewer transformers are required when going through an ADT representation than when transforming directly

Evolution: dealing with change

1. dealing with heterogeneity and also future evolution is simpler if the starting point is abstracted away from implementation details
2. using ADTs allows implementation choices with regard to the degree of value passing or reference passing to suit different environments and conditions

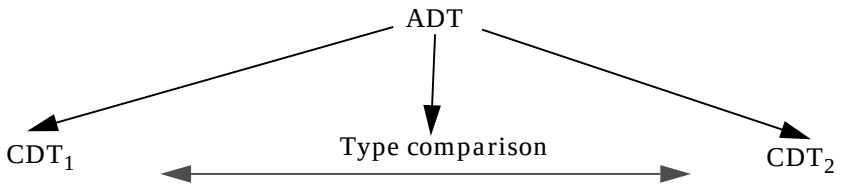
Figure 1.2: Transforming ADTs into CDT's



Type checking (Figure 1.3):

1. simplify type checking: comparison of CIDLs can be complex or impossible without human intervention as base types cannot always be matched or easily translated between CIDLs. Use of ADT together with the transformation decisions taken in the implementation process can simplify the comparisons of concrete types
2. ease of translation between CIDLs: translation implies possible change of semantics and the need for testing which is likely to be a complex task. Using ADTs can simplify the task.

Figure 1.3: Type comparison



Interworking (Figure 1.4):

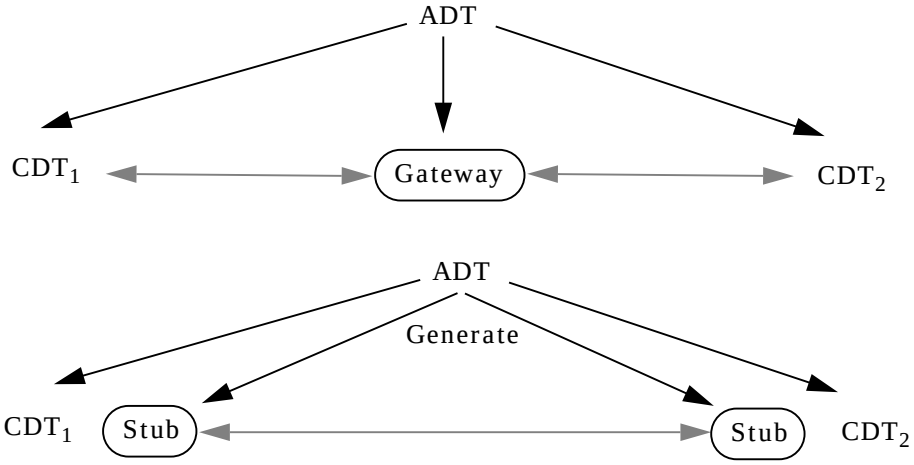
1. the explicit stage of going from ADTs to CDTs simplifies the generation of tools such as stub generators and gateways.

1.9 Reverse engineered ADTs

This entails constructing an ADT description of a given CDT (Figure 1.5). The ADT can then be used to ease the task of (Figure 1.4):

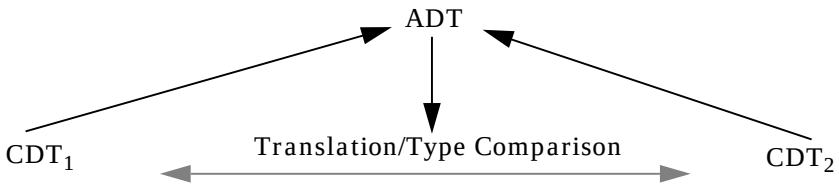
- type conformance checking

Figure 1.4: Translation between CDTs using gateways or stubs



- building of translators between services and client described in different CDTs: stubs and gateways.

Figure 1.5: Re-engineering the ADT from a CDT



1.10 ADTs and AIDLs, CDTs and CIDLs

IDLs evolved as languages for specifying the signatures of interfaces and were originally heavily influenced by programming languages which resulted in the inclusion of base types and constructors. These IDLs are referred to as concrete IDLs (CIDLs) in this paper.

Some early IDLs did not include the ability to define and refer to interfaces from within other interfaces. Such IDLs cannot be used to describe ADTs. Also, in some early IDLs the explicit passing of an interface references as parameters was not possible.

Currently there is a trend to include additional facilities in CIDLs to enable ADTs to be described. The result is that some current IDLs are a hybrid between AIDLs and CIDLs in that they provide sufficient facilities to allow them to describe ADTs. However, the use of base types is not prohibited thus allowing a mixture of ADT and CDT like descriptions of interfaces.

In ADTs the way interfaces are referred to is by name. The concept of interface references is an engineering concept introduced at the CIDL level.

Ultimately, neither IDLs nor translation to a resident language like C will be necessary. There is no reason why a compiler cannot translate directly from ADT to a concrete representation, provided the mapping to the concrete representation is explicit. There has to be a match with an OTWF and the relevant stubs built.

Note: ajh didn't understand this paragraph, so it needs sorting out.

However, in practice, the emergence of current platforms such as DCE and CORBA necessitates the use of their IDLs and also raises the need to be able to map between them.

Note: We need to say something about the two sorts of type (data type and interface type) in concrete IDLs compared with the one in ADT systems.

2 Review and examples of IDLs

2.1 Review of various concrete IDLs

In this section we first present some general features of concrete IDLs, and then compare the features of ANSAware, CORBA and DCE IDLs with these. For more information on the specific IDLs, see [ARM], [OMG 92] and [OSF 92} respectively.

Note that *DPL* is a distributed programming language developed and used by the ANSA team (see for example the DPL Programmers' Manual, [APM.1014.1 93]). It must not be confused with *dpl*, which is the language accepted by the *prepc* pre-processor, as part of the development process in ANSAware.

2.2 General Features of IDLs

Some general features of concrete IDLs are outlined below, as background against which to compare the specific examples.

2.2.1 Types

There is a set of base types, including:

- boolean
- signed and unsigned integers of various lengths, including 16 (short) and 32 (long)
- short and long floating-point numbers
- char
- octet or byte (8-bit quantity not requiring translation)
- string

2.2.2 Type Constructors

There is a set of type constructors, which can be used to construct more complex data types. Typically, these constructors include:

- enumeration
- array
- sequence (variable-length array)
- record
- choice (discriminated union)
- 'alias' (another name for an existing type)

2.2.3 Constants

- it is possible to define constants of any base type

2.2.4 Scoping

- there are rules about names (of types, constants, interfaces etc.) and where in an IDL file they can be referred to

2.2.5 Interface References

- there is a data type which represents an interface or object.
- this data type may have a special value 'nil' which refers to no interface (object).

2.2.6 Structure of Invocations

- an interface type consists of a set of operations.
- each operation has arguments, sent from client to server, and results sent from server to client. These must be of a basic or constructed type, or an interface reference.
- each operation may have a number of *terminations*, by which various possible outcomes can be distinguished. Most concrete IDLs do not have this feature.
- there are two kinds of operations; the IDL has some syntactic construct(s) to distinguish the two:
  (i) the caller waits for the call to complete (in ANSAware this is called an *interrogation*)
  (ii) the caller does not wait, and therefore cannot receive any results (in ANSAware this is called an *announcement).*

2.3 ANSAware IDL

2.3.1 Types

- base types are called `BOOLEAN`, `SHORT CARDINAL`, `CARDINAL`, `LONG CARDINAL`, `SHORT INTEGER`, `INTEGER`, `LONG INTEGER`, `REAL`, `LONG REAL`, `OCTET`, `CHAR`, `STRING`
- integers are 16 bits (`SHORT`) or 32 bits (unspecified or `LONG`).

2.3.2 Examples of Constructed Types

2.3.2.1 Enumeration

```
Colour: TYPE = {Red, Green, Blue};
```

2.3.2.2 Array

```
Address: TYPE = ARRAY 16 OF OCTET;
```

2.3.2.3 *Sequence*

```
Vector: TYPE = SEQUENCE OF REAL;

/* generated 'C' typedef */
typedef struct Vector {
     ansa_Cardinal length;
     ansa_Real *data;
} Vector;
```

2.3.2.4 *Record*

```
HostEntry: TYPE = RECORD
     H_Name: STRING,
     H_Type: INTEGER,
     H_Addr: Address
]
```

2.3.2.5 *Choice*

```
Result: TYPE = CHOICE Colour OF {
     Red => INTEGER,
     Green => STRING,
     Blue => REAL
};
```

2.3.2.6 *Alias*

```
MyAddr: TYPE = Address;
```

2.3.3 Constants

- there are no *constant* declarations in ANSAware IDL.

2.3.4 Scoping

- there is no real concept of scoping in ANSAware IDL, as an IDL file defines exactly one interface. However, interfaces can be referred to from other interface definitions by using the '`INTERFACEREF OFTYPE`' facility, as shown below.

2.3.5 Interface References

- the type `INTERFACEREF` refers to an *interface instance*. An interface instance may include some state, and so is like an interface to a specific object instance.
- it is possible to define a type which is a reference to an interface instance of a specific type:

```
MyRef: INTERFACEREF OFTYPE InterfaceTypeName;
```

- for each interface defined in IDL, a corresponding interface reference type is automatically defined.
- the internal structure of an interface reference can be visible to the programmer; it is defined in terms of the base types using the constructors. However, programmers are not usually expected to deal with the internal structure of an interface reference explicitly. When this is necessary, special facilities are provided by the infrastructure.

- there is no defined ‘nil’ value, which refers to no interface instance.

For further information about interface references, see [RC.268.01 91] and also the ANSAware Reference Manual [APM.1014.1 93].

2.3.6 Structure of Invocations

- *arguments* go from client to server, and multiple *results* come back from server to client.
- the two kinds of operation are called *interrogation* and *announcement*.
- there are no multiple terminations.

2.3.7 Example of an interface declaration

```
example: INTERFACE =
BEGIN
     op1:OPERATION [str:STRING] RETURNS [INTEGER];
     op2:ANNOUNCEMENT OPERATION [str:STRING];
END.
```

2.4 CORBA IDL

2.4.1 Types

- base types are called *boolean, short, long, unsigned short, unsigned long, float, double, octet, char, string* and *any*
- integers can be 16 or 32 bits
- there is a type *any*, which can represent any base or constructed type. The ‘C’ language mapping of *any* is a structure with a *type_code* field and a value.
- the *sequence* constructor allows recursion, so it is possible to construct a list type.

2.4.2 Examples of Constructed Types

2.4.2.1 Enumeration

```
enum colour {red, green, blue};
```

2.4.2.2 Array

```
typedef octet address[16];
```

2.4.2.3 *Sequence*

```
typedef sequence<float> vector;

/* 'C' language mapping */
typedef struct {
     unsigned long _maximum;
     unsigned long _length;
     float *_buffer;
} _IDL_SEQUENCE_float;
typedef _IDL_SEQUENCE_float vector;

// 'C++' language mapping is not yet fixed
```

2.4.2.4 *Record*

```
Record
struct host_entry {
     string h_name,
     long h_type,
     address h_addr
};
```

2.4.2.5 *Choice*

```
union result switch(colour) {
     case red:
          long l_result;
     case green:
          string s_result;
     case blue:
          float f_result;
}
```

2.4.2.6 *Alias*

```
typedef address my_addr;
```

2.4.3 Scoping

- the entire IDL file, and each *module* and *interface* form naming scopes.
- the only point of the *module* construct is for scoping. A *module* can contain type, constant, exception, interface and module definitions.
- names of types, constants, enumeration values, exceptions, interfaces and attributes are subject to scoping.
- a name can be used 'unqualified' within the scope where it is defined, or any scope within it; a qualified name (containing '::') can be used elsewhere.

2.4.4 Interface References

- an IDL parameter or function result can be of an interface type. This means that an object reference is passed.
- there is a predefined C type/C++ class *Object*, which is opaque, and is the type of an object reference for any type of object.
- there is a literal `OBJECT_NIL` of type *Object*. This denotes no object.

- if an interface 'fred' is declared in IDL, a type 'fred' is automatically declared in 'C' as an alias of *Object*. This enables the programmer to use 'fred', though correct usage cannot be checked. In all the currently proposed C++ language mappings, however, conversion of an *Object* to an object reference of a specific type is done via a special member function *_narrow*, which performs a check.

2.4.5 Structure of Invocations

- CORBA IDL is syntactically like C++, so arguments and results are implemented as:

  (i) parameters, which can be *in*, *out* or *inout*, and

  (ii) a single function result.

- CORBA has *exception* declarations. These permit the declaration of exception names, each with an associated structure type. These are analogous to the 'named terminations' of DPL.
- the two kinds of operation are distinguished by the presence or absence of the *oneway* attribute. A *oneway* operation is an *announcement*.
- there is a mechanism for specifying that part of a client's environment should be passed to the server.

Note: passing client's environment information can have implications on management, monitoring, accounting, security, fault tolerance, transactions, and should be further explored.

- there is an alternative syntax for specifying operations as *attribute*s, which are equivalent to 'get' and 'set' operations; *readonly* can be specified, in which case there is no 'set'. This is purely a syntactic feature: the following IDL definitions:

```
interface example {
     attribute int att1;
     readonly attribute float att2;
};
```

and

```
interface example {
     int _get_att1();
     void _set_att1(in int i);
     float _get_att2();
}
```

are equivalent as far as their 'C' language mappings are concerned:

2.4.6 Other features

- interfaces can *inherit* operations (and types, constants, exceptions and attributes) from each other. This is only inheritance of definitions, not inheritance of implementations.
- there is a *module* keyword for scoping

2.4.7 Example of an interface declaration

```
interface example {
     void op1 (in string str, out long res);
     oneway void op2 (in string str);
};
```

2.5 DCE IDL

2.5.1 Types

- the base types are called *boolean, [unsigned] {hyper | long | small | short}, float, double, char, byte, void*
- integers can be 8 (*small*), 16 (*short*), 32 (*long*) or 64 (*hyper*) bits
- *string* is not a base type; there is a *string* attribute which can be applied to arrays of, or pointers to, *char* or *byte*
- there is a *void* type, used like in 'C' for an operation with no result, and 'void *' for context handle parameters and null pointer constants.
- there are predefined types for error reporting (*error_status_t*) and for internationalisation (`ISO_LATIN_1`, `ISO_MULTI_LINGUAL`, `ISO_UCS`)
- a type may be represented in different ways in the application and 'on the wire', with automatic conversions.

2.5.2 Type Constructors

- arrays can be *fixed*, *conformant* or *varying*. A *conformant* array is truly varying, i.e. its size is determined by another structure member or a parameter. A *varying* array is actually a part of an array, and the part that is transferred depends on a structure member or parameter.
- there is no explicit *sequence* constructor, but a structure containing a length and a pointer to a conformant array can be used to achieve the same effect.
- there is a *pipe* constructor. A *pipe* is an open-ended sequence of elements of the same type. Pipes are more messy to program than conformant arrays, and their main advantageous use is where the data does not exist in the required form in memory (e.g. it is being obtained from a file).
- pointers can be used. When a pointer is passed, the object pointed to is marshalled and sent across the wire. This process continues recursively, so lists and trees can be sent. No pointer is expanded more than once.
- structures, arrays, strings and enumerations can optionally be transmitted in the format used by a previous version of the RPC mechanism.

2.5.3 Examples of Constructed Types

2.5.3.1 String

```
typedef [string] char *string;
```

2.5.3.2 *Enumeration*

```
typedef enum {red, green, blue} colour;
```

2.5.3.3 *Array*

```
typedef octet address[16];
```

2.5.3.4 *Sequence*

```
/* Explicitly specified as a record consisting of the length of
 * the sequence and a pointer to the data items, as in the 'C'
 * type definition generated from a 'sequence' constructor in
 * ANSAware IDL. We could follow CORBA and have a 'max' field too.
 */
typedef struct {
     unsigned long len;
     [size_is(len)] float *data;
} vector;
```

2.5.3.5 *Record*

```
typedef struct {
     string h_name;
     long h_type;
     adddress h_addr;
} host_entry;
```

2.5.3.6 *Choice*

```
typedef union switch (colour col) {
     case red:
          long l_result;
     case green:
          string s_result;
     case blue:
          float f_result;
};
```

2.5.3.7 *Alias*

```
typedef address my_addr;
```

2.5.4 Scoping

- there is no real concept of scoping in DCE IDL, as an IDL file defines exactly one interface.

2.5.5 Interface References

There is no single type in DCE IDL which represents an interface; the information held in an interface reference is contained in multiple *binding handle*s and a *context handle.*

- a *binding handle* represents the client's current relationship with the server, and may include addressing information for the protocol currently being used, an object identifier to distinguish servers providing the same service, and security information. The set of information which is present will change with time as the client and server establish communication.

- a *context handle* allows a reference to the server's state to be passed around, and can therefore be used to distinguish between different instances of the same interface.
- a *string binding* is a string representation of a binding handle.

2.5.6 Structure of Invocations

- DCE IDL resembles 'C' syntactically, so arguments and results are implemented as
  (i) parameters, which can have the *in* and/or *out* attributes; parameters with the *out* attribute must (following C) have an explicit '*', unless they are arrays
  (ii) a single function result.
- the two kinds of operation are distinguished by the presence or absence of the *maybe* attribute. A *maybe* operation is an *announcement*.
- other operation attributes are *idempotent* (can execute safely more than once) and *broadcast*. The *idempotent* attribute may seem like introducing a transparency issue into IDL, but often the semantics of an operation imply that it is idempotent, so idempotence could be thought of as a concern of the information projection that is also of interest in the engineering projection and is therefore legitimately brought in at the IDL level.
- there are no multiple terminations.

2.5.7 Example of an interface declaration

```
interface example {
     void op1 (
          [in] string str,
          [out]long *l
     )
     [maybe] void op2 (
          [in] string str
     )
};
```

2.5.8 Other Interface Properties

- unique identifier (UUID)
- version number
- optional comms endpoint address (e.g. IP port)

2.5.9 Attribute Configuration Language

The DCE IDL compiler processes not only an IDL file, but also a corresponding ACL file, which modifies the interaction between the application code and the stubs.

- controls generation of client stub code
- marshalling code in-line or out-of-line
- binding can be automatic, or via an explicit handle, passed as a parameter or as a global variable

- types can be represented differently locally and on the network
- communications and server failures can be reported back via a parameter or result.

2.6 Conclusions from IDL survey

The following observations are made:

- Implications of different language levels. As far as translation is concerned, it is generally easier to translate from a higher-level language to a lower one. The fact that we have the Generic Stub Compiler (GSC) makes it even easier to translate ANSAware IDL to the other two, and indeed IDL translation has been done using the GSC.
- ANSAware is the highest level, designed without thinking too much about the sort of languages which would be used to implement distributed applications. CORBA IDL is next, resembling C++. DCE is the lowest level of the three, resembling C
- all the IDLs have similar sets of base types and of constructors
- none of the IDLs have a proper implementation of DPL terminations, though CORBA comes closest with its 'exception' facility
- only CORBA allows multiple interfaces to be defined in the same IDL file. This is useful and ANSAware IDL should be extended to allow this
- none of the IDLs has proper type checking of interface references, but to some extent this is caused by the limitations of the C language; the proposals for the C++ language mapping of CORBA IDL are more satisfactory in this respect
- DCE and CORBA both have constant declarations.The purpose of these seems to be entirely for use by the client and server code - they certainly cannot be used in the IDL
- DCE and CORBA, being based on C and C++, seem to have an excessive number of ways of returning values from interfaces. The function return value and 'inout' parameters seem to be an unnecessary complication
- DCE's arrays seem unnecessarily complex
- DCE's pipes do not seem well thought out
- DCE's pointer facilities require a lot of implementation support to provide facilities that most programmers will not use most of the time.

Suggested extensions to ANSAware IDL:

1. support named terminations
2. allow multiple interfaces to be defined in the same IDL file
3. define a C++ language mapping
4. define a 'nil' interface reference value, or at least an 'is nil' test that can be performed on an interface reference
5. allow interface types to be specified as arguments and results of IDL operations. This would make the '`INTERFACEREF OFTYPE`' construct obsolete, and it would be possible to make interface references and their internal structure invisible at the IDL level.

3 DPL compared with concrete IDLs

This is a comparison of DPL with typical concrete IDLs such as ANSAware, CORBA and DCE.

3.1 DPL

3.1.1 Introduction

DPL is a distributed programming language developed and used by the ANSA team (see for example the DPL Programmers' Manual, [APM.1014.1 93]). It must not be confused with *dpl*, which is the language accepted by the *prepc* pre-processor, as part of the development process in ANSAware.

DPL is more than an IDL, because it includes the implementation as well as the definition of interfaces. However, a subset of DPL can be regarded as equivalent to an IDL. This subset consists of type definitions only.

3.1.2 Types

- in DPL, all types are defined in terms of the operations that they support.
- DPL's so-called 'built-in' types are a part of the language in the same way as standard library functions like 'printf' are part of the 'C' language.
  - (i) boolean - this is not in the standard library, but the implementation is easy and is given in the DPL Programmer's Manual [APM.1014.1 93].
  - (ii) integers - an implementation is given in the standard library. This uses 'C' and is therefore an implementation of signed 32-bit integers. Variations which implement different lengths, and/or unsigned integers, are easily constructed.
  - (iii) string - this is in the standard library.
  - (iv) floating point - this is not in the standard library, but could be implemented easily following the model of integers. The theoretical basis would be the (countable) rationals rather than the (uncountable) reals.
  - (v) char, octet/byte - these are not in the standard library, but they are basically just 8-bit integers and could be easily added.
  - (vi) any other base types thought desirable could be added.

Example of how a floating point type might be defined:

```
real = type (
     add(r:real) ->(real)
     subtract(r:real) ->(real)
     multiply(r:real) ->(real)
     divide(r:real) ->(real) ->divideByZero()
     equal(r:real) ->true() ->false()
     notEqual(r:real) ->true() ->false()
     greaterThan(r:real) ->true() ->false()
     lessThan(r:real) ->true() ->false()
     greaterOrEqual(r:real) ->true() ->false()
     lessOrEqual(r:real) ->true() ->false()
     toString() ->(String)
     print(s:OutputStream) ->()
)
```

3.1.3 Type Constructors

Most of the constructors supplied with concrete IDLs are not available for DPL. However, the DPL standard library includes constructors for lists, stacks and sets of existing types.

- enumeration - a specific enumeration type just needs an ‘equal’ operation and some means of distinguishing the values. The latter can be done (elegantly) using terminations or (efficiently) using integers.
- array - this needs ‘get’ and ‘set’ operations to access an element (with a named termination ‘boundsError’), and perhaps an operation to obtain the length.
- sequence - the ‘sequence’ constructor in the DPL standard library is more like an associative array - the only constraint on the subscript type is that it has a ‘greaterThan’ operation.
  A ‘sequence’ like that found in concrete IDLs would follow the definition of ‘array’ above, with an additional operation to set the length.
- record - this just needs one ‘get’ operation and one ‘set’ operation per field.
- choice - this could be defined like a record, including the discriminator as a field. An alternative would be to have a single ‘get’ which returned the ‘active’ field and indicated by means of a termination which field was present, together with a ‘set’ per field. However, DPL does not have the same need for ‘choice’ as concrete IDLs do, because it has named terminations.
- ‘alias’ - implemented trivially by writing ‘type2 = type1’.

3.1.4 Examples of constructed types

3.1.4.1 Enumeration

Example of how an enumeration type might be defined. The ‘select’ operation returns a named termination for each possible enumeration value, thus allowing the values to be distinguished

```
Colour = type (
     equal(c:Colour) ->true() ->false()
     select() ->Red() ->Green() ->Blue()
);
```

3.1.4.2 Array

Example of how an array type might be defined, This is an array of octet.

```
address = type (
     get_length() ->(Integer)
     get_element(subscript:Integer) ->(octet) ->boundsError()
     set_element(newval:octet subscript:Integer)
          ->()
          ->boundsError()
);
```

3.1.4.3 Sequence

Example of how a sequence type might be defined.

```
vector = type (
     get_length() ->(Integer)
     set_length(Integer) ->()
     get_element(subscript:Integer) ->(real) ->boundsError()
     set_element(newval:real subscript:Integer)
          ->()
          ->boundsError()
);
```

3.1.4.4 Record

Example of how a record type might be defined.

```
host_entry = type (
     get_h_name() ->(String)
     get_h_type() ->(Integer)
     get_h_addr() ->(String)
     set_h_name(name:String) ->()
     set_h_type(type:Integer) ->()
     set_h_addr(addr:String) ->()
);
```

3.1.4.5 Choice

Example of how a ‘choice’ type might be defined. The discriminator is of the enumeration type ‘Colour’ defined above. This definition suffers from the disadvantage of allowing the ‘wrong’ field to be ‘set’ or ‘got’.

```
result = type (
     get_discriminator() ->(Colour)
     get_red() ->(Integer)
     get_blue() ->(String)
     get_green() ->(Integer)
     set_discriminator(Colour) ->()
     set_red(Integer) ->()
     set_blue(String) ->()
     set_green(Integer) ->()
```

3.1.4.6 Constants

- DPL has ‘fixed bindings’, though these need not necessarily be compile-time constants.

3.1.5 Scoping

- operation parameters are in scope for the operation body
- fixed and variable bindings are in scope for the block in which they are defined, if the evaluation order is sequential.
- names of types and interfaces are in scope for the whole type or interface expression.

3.1.6 Interface References

- conceptually everything that is passed is an interface reference, so the question ‘what type represents an interface reference ?’ does not arise.

3.1.7 Structure of Invocations

- an object has one or more interfaces
- an interface has a set of operations
- each operation has arguments, and a set of terminations
- each termination has a name (at most one termination may be anonymous) and results
- there is no distinction between data types and interface types, and so the arguments and results can be of any type
- in previous versions of DPL, an operation could be an announcement, in which case it had no terminations; announcements are being removed from the language and replaced by *activity constructor*s.

3.1.8 Example of an interface declaration

```
example = interface
(
     op1 (str:String) ->(Integer)
     op2 (str:String) % announcement - being removed from DPL
)
```

3.2 Conclusions

DPL, considered as an AIDL, has a single concept of ‘type’; there is no distinction between what we might call ‘data types’ and ‘interface types’, and therefore no need for the concept of an ‘interface reference’. ANSAware and DCE IDLs have a distinction between data types and interface types; CORBA IDL does not.

To be able to transform concrete IDLs into DPL, there would need to be an additional standard library built for DPL. It would contain:

- additional base types
  - — boolean, as in the DPL Programmer’s Manual
  - — floating point, as given above (with an optimised implementation like integer, if it were ever going to be used)
  - — char, octet/byte
- additional type constructors
  - — array and sequence, as given above

— record, choice

— enumeration

It is not entirely obvious how the constructors for record, choice and enumeration would be defined in DPL (because of the variable number of arguments to the constructor) and it may be that some sort of preprocessing might be the only way to construct the definitions.

4 IDL and inter-working

Inter-working between systems which use the same CIDL:

- type checking from the point of view of naming is straightforward since base types, constructors and their names match
- OTWF is agreed and reflected in stubs.

Inter-working between systems which use different CIDLs:

- name mismatch can cause problems when type checking (same name for different types or different names for same type). Constructors may also be difficult to compare and translate
- agreement on OTWF.

Semantics of operation and termination names also need to be agreed.

4.1 Interworking conformance checks

The transition from ADT to a concrete language representation requires a number of decisions to be made and is dependent on the environment for which the implementation is intended. As much as possible, this transition should be hidden from the application programmer.

Interworking depends on:

- type checking conformance
- OTWF agreement

The choice of CIDL and OTWF are two points where mismatches with other implementations can occur as far as interworking is concerned.

4.1.1 Levels of type checking

There are three levels at which differences can emerge in heterogeneous systems: AIDL, CIDL and OTWF.

Type checking can be done at:

- CIDL level
- ADT + transformation level: this requires the matching of the AIDL definition and the transformations to the CIDL definition of the interfaces.

In order to test type conformance it is sufficient to test conformance at the CIDL and OTWF levels. However, in some case the comparison of two different CIDL definitions of an interface may be a complex process or one which can not be fully automated. In such cases it may be easier to do a comparison at the AIDL level together with the transformation decisions taken on the client and the server side. Alternatively, the CIDL definition may be reverse engineered into ADT description and then compared; however, problems may arise because some CIDLs only allow you to say that an

interface reference is passed, and there is no way of specifying what interface type it must refer to.

There are different ways of defining integers for example, but they are all likely to be translated to the same CIDL definition if supported in the programming language used.

When reverse engineering a CIDL description to an ADT, it is unlikely that a single person will translate two compatible CDTs to different ADTs. However, a compatibility problem arises if the translation is carried by two different people. The resulting ADT descriptions may differ unnecessarily.

4.2 Two extreme worlds: passing references versus passing values

The difference between ADTs and CDTs points to a spectrum of ways in which objects can be realized in a distributed system.

4.2.1 Passing references only

A “pure” ADT world (infinite resources and all arguments and results exist a priori). In this world, no value passing is allowed; only references can be passed. Clearly, an implementation of such a model will be inefficient in terms of space, communications and time.

It is possible to get interworking by passing interface references alone since no presentation problems arise, as no data is passed between objects. However, the problem of different interface references in different environments still has to be sorted out.

For distribution, the pure ADT world is more appropriate from the point of view of copying problems, divergence and data consistency.

4.2.2 Passing values only

No reference passing is allowed, everything is copied and passed as values.

From performance point of view the value passing world is better in some cases where certain object are referred to often, for example. Computationally, implementing types such as integers in pure ADT will be very inefficient.

4.2.3 Trade-off between the extremes

There is an entire spectrum of implementation choices which lie between the two extremes described above. Each implementation can choose the appropriate compromise between passing values and passing references. For objects referred to often and of small granularity, passing by value will be more efficient. For objects referred to infrequently or of large granularity, passing references may be more appropriate. The problem of consistency of data when copying values has to be taken into account and may be the overriding factor in this decision.

The advantage of describing a program in terms of ADT is that it can be implemented in either environment and that the fine tuning of the balance between the two can be changed as the need arises. Thus evolution and scaling.

Note: Abstract specification + automated tools => dealing with evolution and scaling

4.3 Unresolved issues

- Ifrefs to carry presentation level info?
- ?

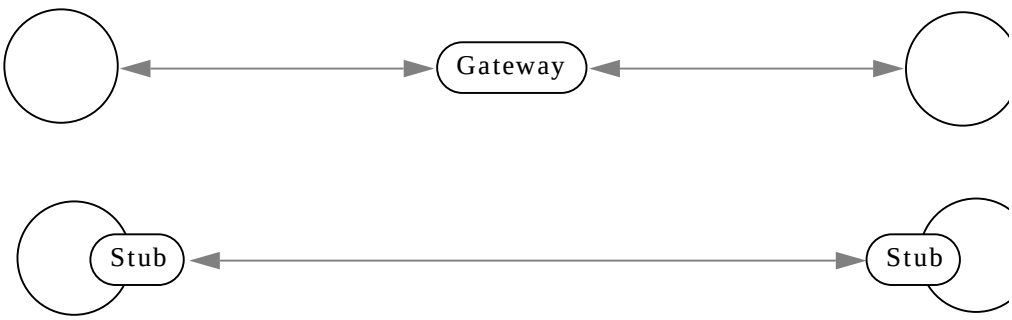
4.4 Crossing domain boundaries

Note: rough notes - revise! ajh suggests reading the interception AR

There are two ways in which interworking incompatibilities between objects can be overcome:

- domain gates:
  - (i) translation of ifrefs carried out by gate
  - (ii) requires type independent i.e. polymorphic gate to avoid having to rewrite the gate with every new interface type
  - (iii) changes in either side of gate are transparent to object programmers
  - (iv) O(n) translators required (?)
  - (v) does not require to change objects with new environments; only gate requires changes
- translation at object stubs:
  - (i) an additional environment means changes to all objects
  - (ii) not type specific i.e. polymorphic gate - not have to rewrite gate with every new if type
  - (iii) interface versioning is more difficult to accommodate
  - (iv) $O(n^2)$ translators required between objects.

Figure 4.1: Gateways and stubs



Note: ajh suggests that we look at jpd's trader AR, especially the crossing boundaries bit.

Note: ANSA type manager = CORBA interface repository.

5 Appendix

5.1 Detailed examples

5.1.1 Implementation of an enumeration type

This conforms to 'Colour' defined above.

```
Red = object (
     interface % the Red interface (
          equal(c:Colour) ->true() ->false()
          [
               after (c.select()) handle
               (
                    Red() [ ->true() ]
                    Green() [ ->false() ]
                    Blue() [ ->false() ]
               )
          ]
          select() ->Red() [ ->Red() ]
     )
)
```

and similar definitions for Green and Blue.

5.1.2 Implementation of a record type

This conforms to 'host_entry' as defined above.

```
my_host_entry = object [
     h_name:String := "";
     h_type:Integer := 0;
     h_addr:String := "";
     interface (
          get_h_name() ->(String) [ h_name ]
          get_h_type() ->(Integer) [ h_type ]
          get_h_addr() ->(String) [ h_addr ]
          set_h_name(name:String) ->() [ h_name := name | () ]
          set_h_type(type:Integer) ->() [ h_type := type | () ]
          set_h_addr(addr:String) ->() [ h_addr := addr | () ]
     )
]
```

5.1.3 Alternative definition of 'choice'

This version only gets the field that the discriminator indicates is present, and uses named terminations to indicate which field this is. The 'set' operations also set the discriminator field.

```
result = type (
     get_active_field() ->Red(Integer) ->Blue(String)
          ->Green(Integer)
     get_red(red:Integer) ->()
     set_blue(blue:String) ->()
     set_green(green:Integer) ->()
);
```

5.1.4 Implementation of alternative ‘choice’

This is conformant to ‘result’ as above. The ‘get_active_field’ operation uses the ‘select’ operation of the ‘Colour’ type. The ‘set’ operations also set the discriminator.

```
my_result = object [
     res_discrim:Colour := Red;
     res_red:Integer := 0;
     res_blue:String := "";
     res_green:Integer := 0;
     interface (
          get_active_field() ->Red(Integer) ->Blue(String)
               ->Green(Integer)
          [
               after(res_discrim.select()) handle (
                    Red() [ ->Red(res_red) ]
                    Blue() [ ->Blue(res_blue) ]
                    Green() [ ->Green(res_green) ]
               )
          ]
          set_red(red:Integer)
               [ res_discrim res_red := (Red | red) ]
          set_blue(blue:String)
               [ res_discrim res_blue := (Blue | blue) ]
          set_green(green:Integer)
               [ res_discrim res_green := (Green | green) ]
     )
]
```

5.1.5 Avoidance of ‘choice’

Definition of a type which uses a ‘choice’:

```
example_1 = type (
     op() ->(result)
);
```

More natural DPL style definition without ‘choice’. Gives you the same as invoking ‘op’ in example_1 above and then invoking the ‘get_active_field’ operation on the result:

```
example_2 = type (
     op() ->Red(Integer) ->Blue(String) ->Green(Integer)
);
```

References

[APM.1000.1 93]

van der Linden, R.J., “An Overview of ANSA”, Architecture Projects Management, Cambridge, 1993.

[APM.1001.1.93]

Rees, R.T.O., “The ANSA Computational Model”, Architecture Projects Management, Cambridge, 1993.

[APM.1014.1 93]

Howarth, N.J., Otway, D.J., Rees, R.T.O. and Watson, A.J., “DPL Programmers' Manual”, Architecture Projects Management, Cambridge, 1993.

[APM.1017.1 93]

Iggulden, D., “Architecture and Design Frameworks”, Architecture Projects Management, Cambridge, 1993.

[APM.1020.1 93]

Otway, D.J., “Abstract and Automate”, Architecture Projects Management, Cambridge, 1993.

[APM.1021.1 93]

Herbert, A.J. et al, “ORB Interoperability”, Architecture Projects Management, Cambridge, 1993.

[ARM]

ANSAware 4.0 Reference Manual, Architecture Projects Management, Cambridge, 1992.

[OMG 92]

“The Common Object Request Broker: Architecture and Specification”, Document Number 91.12.1, Object Management Group and X/Open, 1992.

[OSF 92}

“OSF DCE Application Development Guide”, Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, USA, 1992.

[RC.258.00 92]

Watson, A.J., “Types and projections”, Architecture Projects Management, Cambridge, 1992.

[RC.268.01 91]

Nicolaou, C.A., “ANSAware 4.0 Interface References”, Architecture Projects Management, Cambridge, 1991.

[RC.339.01 92]

Watson, A.J., “Revising the DPL type system”, Architecture Projects Management, Cambridge, 1992.