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

A Toolkit for Access Federation Experimentation

Gray Girling

Abstract

The business problem addressed is...

The technical problem created by that business problem is ...

The solution being offered is....

APM.1199.00.01 **Draft** 24 July 1995

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

Distribution:
Supersedes:
Superseded by:

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

A Toolkit for Access Federation Experimentation

A Toolkit for Access Federation Experimentation

Gray Girling

APM.1199.00.01

24 July 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

Copyright © 1995 Architecture Projects Management Limited
The copyright is held on behalf of the sponsors for the time being of the ANSA Workprogramme.

Architecture Projects Management Limited takes no responsibility for the consequences of errors or omissions in this Report, nor for any damages resulting from the application of the ideas expressed herein.

Contents

1	**1**	**Introduction**
1	1.1	Audience
1	1.2	Scope
2	1.3	Motivation
3	**2**	**Toolkit**
3	2.1	Generic Transfer Syntax “Notation”
4	2.2	Transfer Syntaxes
4	2.3	Transferred Data Types
5	2.4	Type Representation
5	2.5	Generic Remote Operation Abstract Syntax
8	2.6	Conformance
9	**3**	**Toolkit Uses**
9	3.1	Transfer Syntax Interceptor
10	**4**	**Related E-mail**

1 Introduction

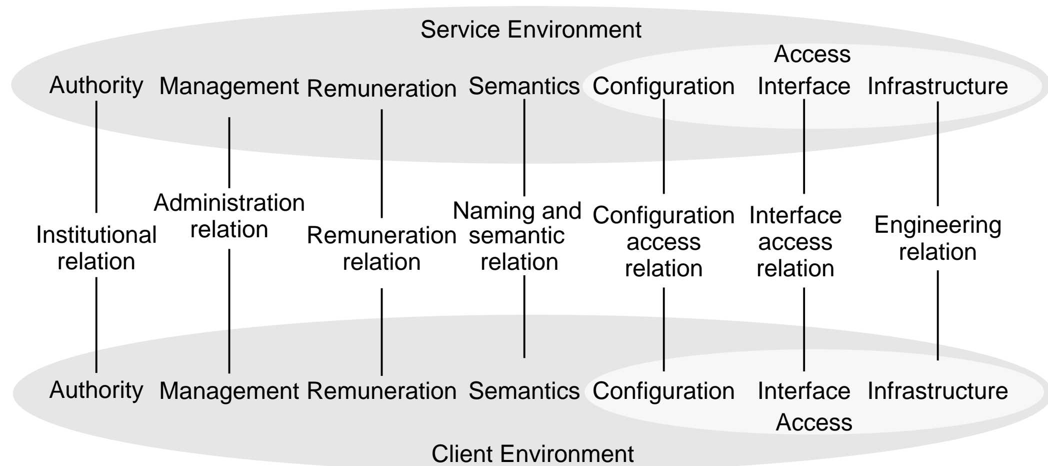
1.1 Audience

This is an internal document requesting comments from the ANSA team regarding the potential use of the toolkit described in C++ federation experimental code.

1.2 Scope

[APM 1139] decomposes the tasks of federation, separation and monitoring into a number of separate areas, each of which need to be addressed (figure 1.1).

Figure 1.1: The relations between properties of two distributed systems



One, important task involves enabling a client's access to a service, which [APM 1139] analyses in terms of infrastructure, interface, and configuration differences. Part of the mechanisms required to address access federation are those addressing differences in local, transfer and abstract syntax (figure 1.2).

Each of these syntaxes implicitly provide rules determining how information or interactions are to be represented.

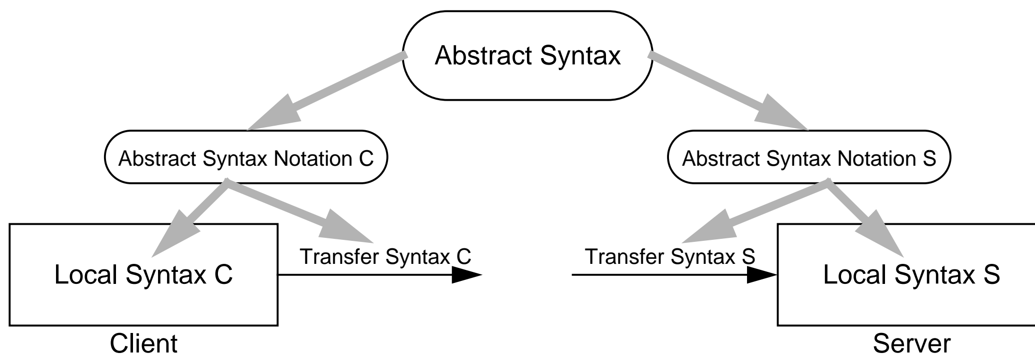
- Local syntax

  is the term referring to the rules used in a programmer's local environment (e.g. those enforced by the C++ compiler he is using).

- Transfer syntax

  is the term referring to the rules used in formatting an end-to-end communication used to support an interaction.

Figure 1.2: Local, transfer and abstract syntaxes



- Abstract syntax

  is the term conceptually embodying whatever is common between all the local or transfer syntaxes that might be associated with an interaction (for example it may define the type that any more concrete syntax should provide for each of the arguments in an invocation). An interface signature is an example of an abstract syntax specification.

An abstract syntax notation is a (normally formal) language in which abstract syntaxes can be expressed. An interface definition language is an example of an abstract syntax notation.

This document considers a programers view of these notions and proposes an object orientated outline of code that provides the relevant abstractions. This forms a toolkit that could be used as the basis of implementation experiments in access federation. The toolkit is described in C++, but could be implemented in other object orientated languages.

This document focuses on abstract syntaxes for operational interfaces (as defined in [ISO ODP-3 94]), which support the interactions used when providing potentially remote operations (e.g. remote procedure call protocols).

The document deals with

- the representation of a transfer syntax in C++
- the representation of a remote operation abstract syntax in C++
- the dynamic creation of the latter
- type checking and conformance between these abstract syntaxes
- dynamic creation of transfer syntax interceptors in C++

It does not deal with differences in abstract syntax notations.

1.3 Motivation

This document is largely motivated by a desire to support dynamic selection of transfer syntax and the type of remote invocations whose signature is unknown at run time but which must be correctly type checked (e.g. in order to operate the transfer syntax mechanism).

2 Toolkit

2.1 Generic Transfer Syntax “Notation”

A chosen generic transfer syntax notation is represented by a class consisting only of virtual functions which has methods supporting a set of base types and a selection type constructors. A very simple one might support only bit strings and have constructors for sequences and choices:

```
class TransferSyntax
{ public:

  /* base type: bit string */
      virtual void start_bitstring() = 0;
      virtual void put_bits(int bits, int len) = 0;
      virtual void end_bitstring() = 0;

      virtual len len_bitstring() = 0;
      virtual int get_bits(int len) = 0;

      /* type constructor: sequence */
      virtual void start_sequence() = 0;
      virtual void end_sequence() = 0;

      virutal len len_sequence() = 0;

      /* type constructor: choice */
      virtual void start_choice(choice c) = 0;
      virtual void end_choice() = 0;

      virtual choice which_choice() = 0;
}
```

“`choice`” is the name of a type capable of holding a value representing a selection from a set of alternatives (e.g. int, if MAXINT is large enough). Similarly “len” is the name of a type capable of holding a value representing the number of things in a bitstring or sequence (e.g. int, if MAXINT is large enough).

Note that no support is included to support recogniation of the types that the representations generated are intended to realize. This function is carried out by mutual knowledge of the same abstract synatx (which can be checked by other means).

In practice the generic transfer syntax notation will have to be rich enough for the range of TransferSyntaxes anticipated (e.g. at least to incorporate abstractions of all of the features of ANSAware IDL and CORBA IDL).

2.2 Transfer Syntaxes

Different transfer syntaxes are represented as examples of the syntax notation.

```
class TransferSyntax1: public TransferSyntax
{ ...
 /* definitions for each of the TransferSyntax functions */
}

class TransferSyntax2: public TransferSyntax
{ ...
 /* definitions for each of the TransferSyntax functions */
}
```

These classes might deal with "`fstream`" representation optimized for compactness, for portability, for human readability etc., or they may deal with representation on a transfer protocol.

2.3 Transferred Data Types

Programming data types that are to be represented in different transfer syntaxes have a class from which they inherit:

```
class Representable
{ public:
     virtual void represent(TransferSyntax &out) = 0;
     virtual void construct(TransferSyntax &in) = 0;
}
```

Representable 32-bit integers, for example might be provided in a class definition such as:

```
class Rep_Int32: public Representable
{ private:
     int value;

 public:
     ...
     /* various operations that integers support (or you could
make its
      value public */

     void represent(TransferSyntax &out)
     { out.start_bitstring();
      out.put_bits(value, 32);
      out.end_bitstring);
     }

     void construct(TransferSyntax &in)
     { len n = len_bitstring();
      value = get_bits(n);
     }
}
```

Derived representable types can use the "`represent`" and "`construct`" methods of the representable types from which they are composed for their own methods (e.g. a representable array of 32 bit integers could use

`Rep_Int32::represent` in its represent routine). In principal known optimal features of specific `TransferSyntax`s could also be taken into account.

2.4 Type Representation

If type checking from dynamically established types is to be feasible a representation of a type will need to be created in a program. Its most likely source is some external reprsentation (either generated by a stub generator, retrieved from local storage or obtained via a transfer syntax), and its most relevant property is that is can be checked for conformance to another type.

We will also require the identification of a function capable of creating an object that can hold a representation of that type.

```
class Type: virtual public Representable
{ private:
 /* data structure for representing a type in the adopted type
system */

 public:

     virtual bool conforms_to(Type &larger_type) = 0;

     virtual &Representable container(void) = 0;

}
```

“`bool`” is a type capable of holding a boolean value `TRUE` or `FALSE` (e.g. `int`).

Note that the relationship between the type system used and the generic transfer syntax notation used is that the types of the type-system should be representable using the generic transfer syntax notation -- they are not necessarily identical (e.g. the base types of the type system may be constructed from the base types of the generic transfer syntax notation).

2.5 Generic Remote Operation Abstract Syntax

Operation invocations consist of a set of arguments (all `Representable` objects), and a number of sets of termination arguments (one for each termination). If the “dynamic” establishment of abstract syntax is to be possible (e.g. the `DataBase` case) we need a data structure to represent this. It will have two purposes:

1. to use for explicit type checking (the compiler can’t do this for us)
2. to manage the marshalling and unmarshalling of transfer syntax in the absence of pre-processor support

Thus it needs to hold both the structure of Representable objects implied and the types associated with them. There are a number of degrees to which these two peices of information can be integrated (e.g. a type representation could be made an attribute in Representable and be constructed from any `Representable` types from which it is constructed). However it is most likely to be derived directly from some external representation of the type delivered with the abstract syntax specification and so its use only in the local representation of the abstract syntax is reasonable.

```
class RO_AbstractSyntax
{ private:

     int argument_n;
     Type *arg_types[];
     Representable *arguments[];

     int termination_n;
     struct
     { choice terminator; /* must be unique in all conforming ops
*/
      int term_result_n;
      Type *result_types[];
      Representable *term_results[];
     } results[];


     int termination_index(choice termination)
     { int index = -1;
      for (int i=0; i<termination_n; i++)
      if (results[i].terminator == termination)
           index = i;
     }


 public:
     ...
     /* methods for constructing the above data structure - or
make the
      data structure public, these can be based on the creation of
 suitable Representables as dictated by the Type objects */
     ...
     /* methods for accessing the values associated with the
Representable's
      in the above data structure - or make the data structure
public */
     ...


     void represent_invoke(TransferSyntax &out)
     { out.start_sequence();
      for (int i=0; i<argument_n; i++)
          arguments[i]->represent(out);
      out.end_sequence();
     }

     void represent_termination(TransferSyntax &out, choice
termination)
     { out.start_choice(termination);
      int i = termination_index(termination);
      if (i >= 0)
      { out.start_sequence();
      for (int j=0; i<results[i].term_result_n; i++)
           results[i].term_results[j]->represent(out);
          out.end_sequence();
      }
      out.end_choice();
     }
```

```
        void construct_invoke(TransferSyntax &in)
        { len args = in.len_sequence();
         /* should equal argument_n */

         for (int i=0; i<argument_n; i++)
             arguments[i]->construct(in);
        }

        void construct_termination(TransferSyntax &in)
        { termination = in.which_choice();
         int i = termination_index(termination);
         if (i >= 0)
         { len results = in.len_sequence();
             /* should equal results[i].term_result_n */

             for (int j=0; i<results[i].term_result_n; i++)
              results[i].term_results[j]->construct(in);
         }
        }

        /* type information methods: */

        bool conforms_to(RO_AbstractSyntax &server_op)
        { bool conforms = (server_op.argument_n == argument_n &&
         server_op.termination_n <= termination_n);
         if (conforms)
         { for (int i=0; i<argument_n; i++)
              if (!arg_types[i]-
>conforms_to(server_op.arg_types[i]))
                  conforms = FALSE;
             for (int t=0; t<server_op.termination_n; t++)
             { int term = termination_index(t);
              if (term<0)
                  conforms = FALSE;
              {   Type *these_types = results[term].arg_types;
                  Type *server_types =
server_op.results[t].arg_types;
                  for (int i=0; i<argument_n; i++)
                   if (!server_types[i]-
>conforms_to(these_types[i]))
                       conforms = FALSE;
              }
             }
         }
         return conforms;
        }

}
```

"`bool`" represents a type capable of holding a boolean value `TRUE` or `FALSE` (e.g. `int`).

Objects of the type `RO_AbstractSyntax` ("virtual invocation objects") can becreated either at run time or at compile time.

- A stub generator could produce the C++ code in-line that uses the `RO_AbstractSyntax` methods, or it could create the data structure it uses directly.

- A program could create the `RO_AbstractSyntax` in all the ways open to it
- If `RO_AbstractSyntax` is expressed in terms of the abstract syntax and made Representable:

```
class RO_AbstractSyntax: public Representable
 { ...
 /* as above */

     void represent(TransferSyntax &out)
     { ... }

     void construct(TransferSyntax &in)
     { ... }
 }
```

It may be delivered in any of the transfer syntaxes available (e.g. one might be provided for its interpretation from a compiler generated string, or one delivered using a transfer protocol). In particular it could be “constructed” from the result of a RDBMS “describe” function or from data retrieved from a Trader.

In either case the transfer syntax used can vary at run time.

Although the above is expressed in terms of single operations packaging operations into interfaces does not represent a major change.

Note that this “generic” abstract syntax suites a specific protocol (i.e. RPC) and that similar approaches could be used for other protocols (e.g. ones associated with streams).

2.6 Conformance

Both the `RO_AbstractSyntax` that a client expects and the one that a serveractually provides need to be known in order to execute `RO_AbstractSyntax::conforms_to`. This can be achieved either at stub generation time (e.g. through the stub generator contacting the trader at that time) or dynamically at run time (e.g. through some trading function, or RDBMS query).

The latter is preferable since it allows errors resulting from changes in a server’s interface specification to be spotted.

It should be noted that this function is the only formal type checking that is performed and that, if it registers non-conformance, it is NOT SAFE to invoke or respond to the remote operation.

The `Type::conforms_to` operation indicates the circumstances in which an object of one type can be delivered when another was expected. This judgement can only be made if the represention of the type (in terms of the generic transfer syntax notation) is known. For example, it may not be safe to assume that types that require only a “smaller number of bits” can be provided in place of a larger type unless they are represented with the larger number of bits expected.

The information in the two `RO_AbstractSyntax` objects can be used by either the client or the server to select transfer functions that safely map provided to accepted Types. When this is done Type::conforms_to can be modified to recognize this fact.

3 Toolkit Uses

3.1 Transfer Syntax Interceptor

By using one transfer syntax for `represent_invoke` and `construct_termination` and another for `construct_invoke` and `represent_termination` a transfer syntax translating interceptor is simply created.

```
class TransferSyntaxInterceptor
{ private:
     TransferSyntax client_syntax, server_syntax;
     RO_AbstractSyntax operation_spec;

 public:
     TransferSyntaxInterceptor(TransferSyntax in, TransferSyntax
out,
      RO_AbstractSyntax op)
     { client_syntax = in;
      server_syntax = out;
      operation_spec = op;
     }

     void forward_invocation()
     { operation_spec.construct_invoke(client_syntax);
      operation_spec.represent_invoke(server_syntax);
     }

     void return_response()
     { operation_spec.construct_invoke(server_syntax);
      operation_spec.represent_invoke(client_syntax);
     }
}
```

Note that the (dynamically created) instances of this class perform the main function of an interceptor for a given remote operation between a client and a server. Because it is dynamically created it may, for example, be derived after negotation with client and server as to the most suitable transfer syntaxes to use.

Again although the above is expressed in terms of single operations packaging operations into interfaces does not represent a major change.

4 Related E-mail

Date: Fri, 4 Mar 94 12:47:58 GMT

From: Gray Girling <cgg>

To: ajh

Cc: fedtg

In-Reply-To: <9403041039.AA10958@costello.ansa.co.uk> (ajh)

Subject: Re: comments on federation infrastructure - C++ notes

Andrew

> Some comments arising from a cursory reading of your note.

> A complaint about the DCE (and ANSAware) is that stubs are very large. Some experimentation suggests that a generic marshalling / unmarshalling engine driven by a simple format string is much more compact and not much slower (typically becuase the code fits into the machine's cache).

Clearly there should be some relationship between whatever format descriptor drives the marshalling code and the runtime representations of types (e.g. they could be the same).

In the example I provided the relationship boils down to the object that holds an abstract syntax: `RO_AbstractSyntax`. This object is the thing that controls the automatic marshalling, or unmarshalling of calls and, if it is a "`Representable`" object, it can be built from an arbitrary transfer syntax. It would not be difficult (as hinted) to define a TransferSyntax object that either created or read readable C strings. The `RO_AbstractSyntax` object would be used with this TransferSyntax in a "stub generator" to write the relevant C string and would be used again in the client or server to regenerate a `RO_AbstractSyntax`.

Where would the stub generator get its `RO_AbstractSyntax` object from? It would generate ("construct") it from a Trader specification (i.e. a signature fromthe trader) - this specification should be expressed in some "trader" `TransferSyntax`. The same source (the trader) and `TransferSyntax` object could alternatively be used in the client or server to generate the `RO_AbstractSyntax` directly at run time. The wonderful thing about all this is that it requiresno extra code to be provided in `RO_AbstractSyntax` – only new `TransferSyntax` objects to be written.

In a type checked world you hardly ever have to pass type descriptions around. About the only exceptions are when trading - becuase the result depends upon the type required.

Also, as Gomer was hinting in his talk, we are not in a type checked world if we try to support some kinds of interaction (in particular ones that are determined dynamically – e.g. by a user).

This has two virtues - less bits on the wire, and early checking. However it might be argued it makes interception harder,since an interceptor cannot unravel packets as they go by, whereras if the types were alongside the data (e.g. as in OSI TLV encodings) one could write generic interceptors at the protocol level.

I talked about this issue in my talk on Wednesday (or at least I attempted transmission). The whole of the information needed is the abstract syntax and this can be provided

- at compile time (but it might change before invocation)
- at bind time, or at any rate before invocation (e.g. creation of a `RO_AbstractSyntax` from trader information, as above)
- at invocation time (by embedding the abstract syntax specification in the transfer syntax - “dynamic typing”?

> I think the argument about interceptors is specious, but this needs checking. As well as converting representations, an interceptor has put up proxy objects so that the address mapping works properly.

Building interceptors as applications is the easiest way to do this! If one had a generic stub driven by type descriptors, then the amount of code required to provide an interceptor would be rather more bounded than in the case where stubs are explcitly generated.

There are clearly many interception functions that have to go on between an arbitrary client and server. The information I gave pertains only to a function that deals with the translation between transfer syntaxes. Obviously the approach (instantiating the mechanism from abstract syntax specifications, and using classes of that style) will be applicable no matter where this

interception function is located (e.g.

- in the client’s capsule
- in a separate computational object
- in the server’s capsule).

The key thing is that if you can represent the abstract syntax(es) in an arbitrary transfer syntax then you can get it whereever it needs to go (and for the transfer syntax interceptor it needs to go there) - either at compile time (e.g. stub generator to string) or at run time (e.g. read from trader).

I would recommend trying to unravel these issues into a little RC. Talk to Dave Otway and Owen Rees who have previous experience and views on these topics.

I would be quite keen to make an RC out of this.

Gray

Gray

> Where would the stub generator get its RO_AbstractSyntax object from? It would

> generate (“construct”) it from a Trader specification (i.e. a signature from

> the trader)

Don’t forget that this information can also come from a file specifying the signature of the operations, as currently occurs in STUBC. Its storage in the trader implies a view of a trader more akin to a specification repository, which is part of the “new trader” work. ... and, since a server knows its own specification it is more applicable to the client than the server.

References

[APM 1139]

Hoffner Y and Girling C G, *Boundaries and Domains*, APM Ltd, Cambridge UK, 1994.

[ISO ODP-3 94]

ISO/IEC JTC1/SC21/WG7 and ITU-T SG VII Q.ODP, *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing – Basic Reference Model of Open Distributed Processing – Part 3: Prescriptive model*, **ISO/IEC DIS 10746-3** and **Draft ITU-T X.903**, Feb 1994