Poseidon House
Castle Park
Cambridge CB3 0RD
United Kingdom

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

ANSA Phase III

DIMMA Stub Generator Design and Implementation

Youcef Laribi

Abstract

Distributed Multimedia programming requires the definition of new kinds of interactions between clients and servers other than operational interfaces. IDLs have been agreed to be the right level for expressing the interactions between clients and servers in a distributed environment.

Within the DIMMA project, we designed and are still implementing a Stub Gnerator Toolset, which supports several IDL languages giving more flexibility in expressing contracts between clients and servers and supporting several runtime environments, in order to enable bridging different environments. The toolset supports also a new IDL language specifically designed to experiment with the expression of multimedia and real-time interfaces.

This report describes the design choices of the DIMMA Stub Generator Toolset, its architecture and the supported IDL languages and runtime environments.

APM.1554.00.05 **Draft** 11th September 1995
Technical Report

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Superseded by: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.

Contents

1	**1**	**DIMMA Stub Generator Design and Implementation**
1	1.1	Introduction
2	1.2	Operational and stream Interfaces
2	1.3	Services and runtime environments
3	1.4	The DIMMA Stub generator Objectives
4	1.5	AST: a key architectural component
6	1.6	Overall Architecture
8	1.7	A DIMMA front-end and an ODP backend
8	1.7.1	The DIMMA IDL front-end
10	1.8	A backend for the ODP runtime environment
10	1.8.1	Marshalling/Unmarshalling parameters
11	1.9	Current Status of the Stub Generator Toolset
11	1.10	Conclusions

DIMMA Stub Generator Design and Implementation

DIMMA Stub Generator Design and Implementation

Youcef Laribi

APM.1554.00.05

11th September 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.

1 DIMMA Stub Generator Design and Implementation

1.1 Introduction

Distributed computing involves complex interactions between distributed entities. In order to manage and control this complexity, one widely used model is to structure the interactions in terms of services. Interacting entities are termed as clients and servers depending on their role in the interaction; whether they provide the service, or they use it. This model is known as the client/server model and each interaction in the system must be either a service request, or a reply to a service request (service provision).

In order to avoid erroneous interactions between communicating entities, service providers define the kind (type) of requests they expect from their clients. Requests that do not conform to the types expected by the service provider are erroneous interactions and must not be allowed by the system. In a similar way to progamming languages, detecting these erroneous interactions as early as possible (e.g at compile time) is very desirable.

To achieve this goal, the type of requests that are accepted by a service provider are expressed in languages called Interface Definition Languages (IDLs) that are usually different from the programming languages used to write the client and the server part but may be the same [Parrington 93]. From this definition, an IDL compiler generates modules in a programming language compatible with the client and the server programming languages. These modules are known as *stubs*. There are mainly two stubs:

- The client stub: This module emulates the server side interface that must be used by the client code. Thus, by linking this module with the client code, it is possible to check the correct usage of the service by the client, using existing type checking mechanisms of the used compiler/linker/runtime[1] at the client side. Moreover, the client stub serves to hide the distribution issues to the client code, by playing the role of a server proxy.
- The server stub: Similarly this module purpose is to hide the distribution issues to the service provider and to interact with the client stub to achieve the service provision.

Most modern distributed computing environments offer IDL languages and corresponding stub generators (e.g ANSA IDL, DCE IDL, CORBA IDL).

1. This depends on when the type checking is performed at the client side (compile-time, link-time or run-time)

1.2 Operational and stream Interfaces

In non-distributed computing, structuring the interactions between software modules has led to modular programming concepts and mainly to the abstraction of the procedure, as the main concept to structure software programs. This abstraction was so simple yet powerful, that there was an early search to extend it to distributed computing [Birrell 84]. This has led to the construction of Remote Procedure Call (RPC) systems, where modules resident on different sites in the distributed system interact by procedure calls, as in non-distributed computing. The RPC system hides the distribution effects, and unifies local and remote programming.

With the advent of the object era, the software structure has changed to become a set of objects, instead of a set of modules. Each object definition consists of a data part and a set of procedures called object operations that manage this data. Objects interact only through invocations. An invocation is a directed interaction that specifies a target object, a name of an operation supported by that object and a set of parameters for that operation. The result of the invocation is the execution of the operation with the given parameters on the server side, and the attention of return results on the client side. In the same way to procedures, new Remote Object Invocation technologies have been built to support remote object usage [Parrington 92]. In that case, objects are said to have *operational interfaces*.

However, interactions between software components are not always of the operational type. For example, if a service provision consists of feeding the client with a continuous flow of data (e.g audio or video streams), this kind of service cannot be specified using an operational interface. The interface is best described as a *flow* of data items of some type from one side to another instead of a request/reply interface. These kind of interfaces are called a *stream interfaces*.

Other kinds of interactions like signals[1] may also be of interest to the distributed computing community. However, most currently available distributed computing environments (e.g DCE, CORBA) restrict their support to operational interfaces.

1.3 Services and runtime environments

Clients and service providers rely on some distributed computing environment to express their interactions, check their validity, enable their occurrence and deal with issues like failures and heterogeneity, found in any realistic distributed environment.

Stubs insulate clients and service providers from most of the engineering details of the runtime environment such as data formats on the wire, used transport porticoes, and other issues such as trading, security and error recovery.

Commonly, it is the IDL compiler that embeds the knowledge about the runtime interface knowledge and generates the adequate stubs according to that knowledge.

1. Usually used in real-time computing

Hence stub generators are tightly related to the runtime environment for which the stubs are generated and any change to the runtime interface (e.g transport protocols), may require updating the stub generator.

1.4 The DIMMA Stub generator Objectives

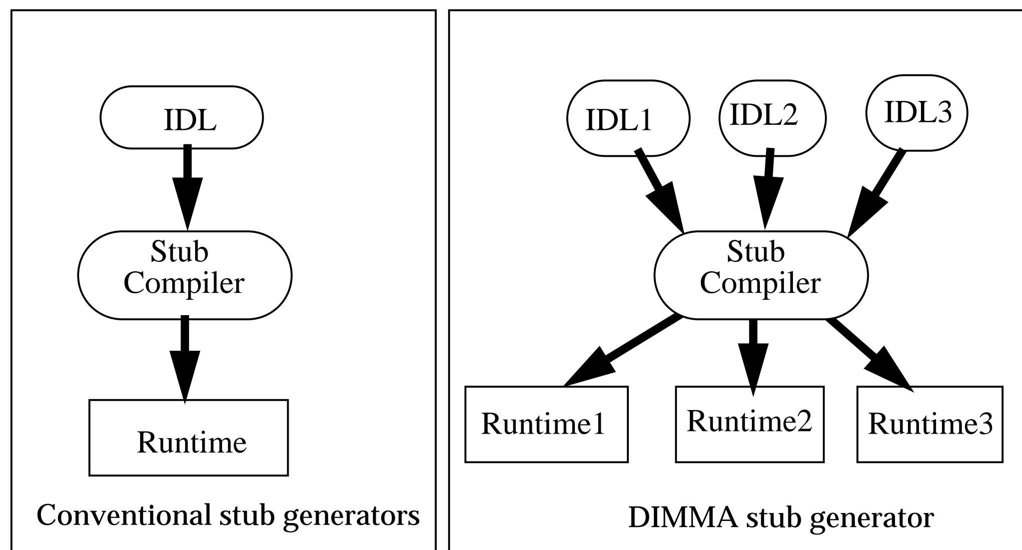
Within the DIMMA project [Li 94], we undertook the design and implementation of a new stub generator toolset for the following reasons:

- The need to support a new IDL language which allows for the definition of streams interfaces, in addition to operational ones. This language called the DIMMA IDL, will be used to specify multimedia service interfaces.
- The desire to support existing IDL languages such as the CORBA IDL [OMG 94]: We want to take advantage of existing services whose interfaces are expressed in other IDL languages.
- The desire to support several runtime environments. As some runtime environments are better suited for some applications, we want to decouple services from runtime environments.

This architectural choices of the DIMMA stub compiler were motivated by the following reasons:

- Services are semantically independent from the runtime environment that they will be running on. Hence, it is interesting to enable the usage of a service, described by an IDL language, in any available runtime environment offering the required features.

Figure 1.1: Different structuring of Stub compilers



- IDLs and runtime requirements are different components and they may evolve independently. It is important to reflect the evolution of one IDL in all the potential runtimes that might be used with it and vice-versa. A stub generator enables the propagation of these changes transparently.
- It is easier to experiment with new features added to an IDL (e.g QoS attributes for multimedia) by using any available runtime supporting the corresponding engineering mechanisms (e.g real-time transport).

Some of the benefits we seek from such a design are:

- Running existing services on several runtime environments (portability).
- Decoupling services interface description from the runtime environment on which the client and the server will run.
- Mapping between different computational models and bridging runtime environments (with the help of an engineering support).
- Enabling clients running on one runtime environment to use services available on other environments (interoperability).

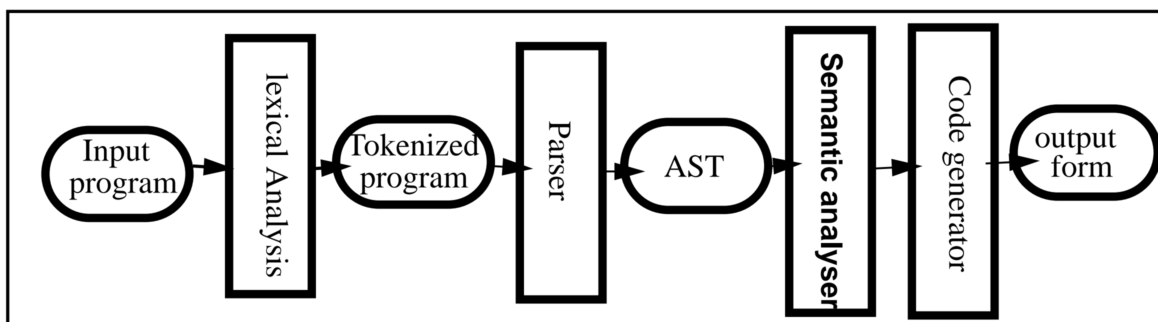
In the remaining sections, we will explain the architecture of the stub generator, how it can be extended to support new IDLs and new runtime environments.

1.5 AST: a key architectural component

A compilation process is usually divided into four phases (cf. Figure 1.2:):

- Lexical Analysis: This phase is dedicated to recognize the lexical entities as defined by the language, and to tokenize the input program. Hence from a program defined as a set of characters, the output of this phase is a set of tokens.
- Syntactic Analysis: This phase checks that the program respects the syntactic rules defined by the grammar of the language. One of the most used outputs of this phase is an intermediate form called the Abstract Syntax Tree (AST).
- Semantic Analysis: Commonly, this phase is dedicated to type and scope checking, ensuring that the operand types in an operation are as expected. It also performs the implicit type conversion where necessary (e.g integer conversion to real).
- Code generation: Starting from the AST, the code generation phase consists of generating the appropriate output required by the target environment.

Figure 1.2: Different phases of the compilation process

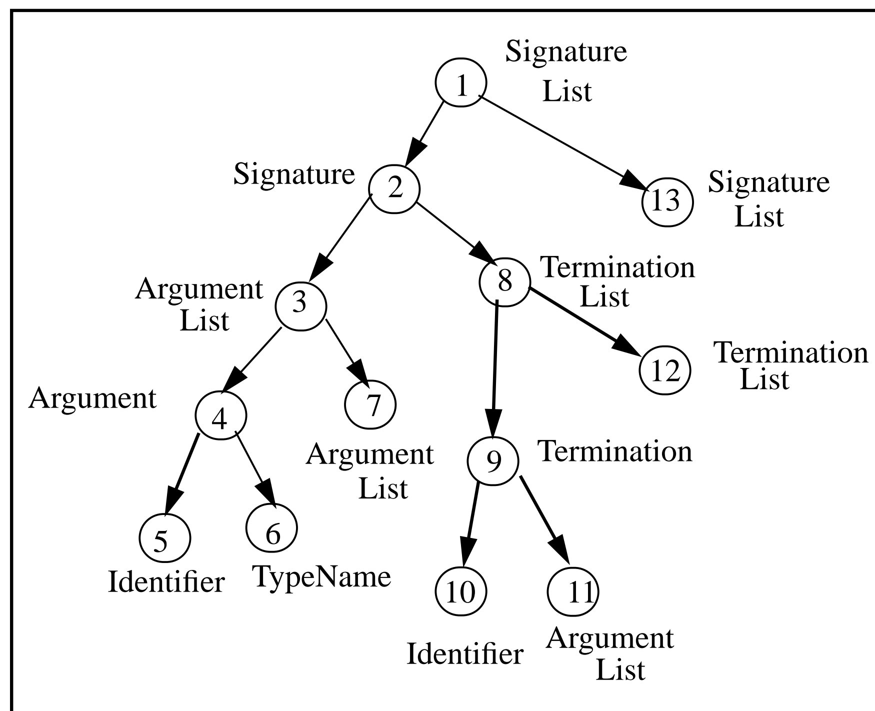


In the DIMMA stub generator toolset, the AST is a central component of the architecture, because it enables to capture the semantics of the service interface, independently from the IDL originally used to describe that service interface. Therefore, the AST module in the Stub compiler architecture, has been designed as a separate component, with well-defined interfaces, that might be used from the front by a parser to generate the AST, or from the back, by a code generator, to generate an output form.

The AST component defines the DIMMA supported computational model. This computational model is based on the ODP reference model [ODP 94]. Each supported IDL in the stub compiler toolset must have for all its features, corresponding representations in the AST component. By compiling an IDL service interface description into a DIMMA AST, the stub generator maps the service interface described by one computational model (expressed by the IDL) into an equivalent representation expressed using the DIMMA computational model.

The Figure 1.3: depicts the internal structure of an AST tree. Each node of the tree represents a syntactic item, and contains the necessary information required by the backend to generate the corresponding output form.

Figure 1.3: AST internal structure



The current AST module implementation supports the following computational constructs:

- Interfaces.
- Objects.
- Terminations
- Signatures.
- Blocks
- Attribute lists.
- Expression lists.
- Identifiers.
- Declaration lists.
- Bindings.
- Literals.

- Basic Types (integer, short, long, boolean, real, char, double, ...).

The front-end is responsible to translate the input service description into an equivalent description based on those computational constructs.

Note: The computational constructs and the attached semantics may evolve in the future, to support new features as new IDL languages and new runtimes are added to the toolset.

1.6 Overall Architecture

The stub compiler can be split up into a set of major components:

- Front-ends: They are responsible for performing the lexical and syntactic analysis. There are as many front-ends as supported IDLs. The front-end allows also to map between the computational model described by its IDL and the DIMMA computational model.
- Backends: generate an output form given an AST. They interact with the AST component to read the AST representation. Each targeted language/ runtime combination is implemented by a separate backend.
- The AST component: Enables the generation of the AST by the front-ends and its usage by the backends.
- The compilation driver: Processes command-line options, chooses the front-end and the backend involved in compiling an input service interface description and coordinates the actions of the different modules in the stub compiler.

The Figure 1.4: outlines the main components of the stub compiler toolset. The AST Generator component serves as the AST interface to the front-end. The internal structure of the AST is hidden from the front-end. Expressed in C++, this interface that enables the front-end to generate the AST looks like:

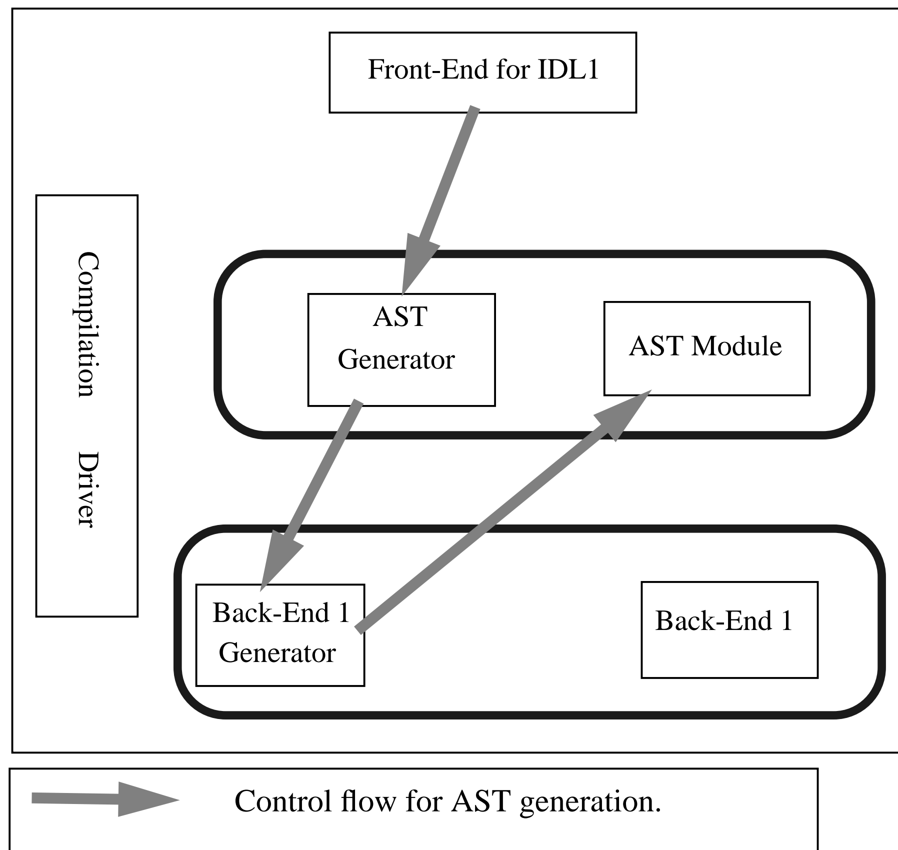
```
1 class AST_Generator {
2 virtual AST_SignatureNode *CreateSignature();
3 virtual AST_TerminationNode *CreateTermination();
4         ....
5 virtual AST_IdentifierNode *CreateIdentifier();
6 };
```

When the front-end creates an AST item (e.g a signature), the AST Generator returns a handle for that item that can be used to create other AST items (e.g a signature list). The handle is opaque to for the front-end and can only be interpreted by the AST Generator.

The AST layout is not necessarily well-suited for the backend to generate the output form. In order to optimize that layout so that code generation from the AST can be done without restructuring it, we allow the backend to change the AST layout during its generation, by providing a back-end generator as is depicted in Fig.1. If a backend involved in the compilation provides a back-end generator, the AST generator will delegate the AST generation to the backend generator.

However, backend generators are constrained to only add to the basic structure of the AST. They are not allowed to change it. This is important to preserve the AST structure independence from the backends used.

Figure 1.4: Architecture of the stub compiler



Using C++ object-oriented features, this can be achieved by letting the backend generator inherit and overload the AST Generator interface. This way, all calls to the AST Generator interface are intercepted by the backend generator, which first creates the AST structure corresponding to the syntactic item (e.g signature), and adds to it the information that facilitates the code generation phase for the backend.

Expressed in C++, the backend generator interface looks like:

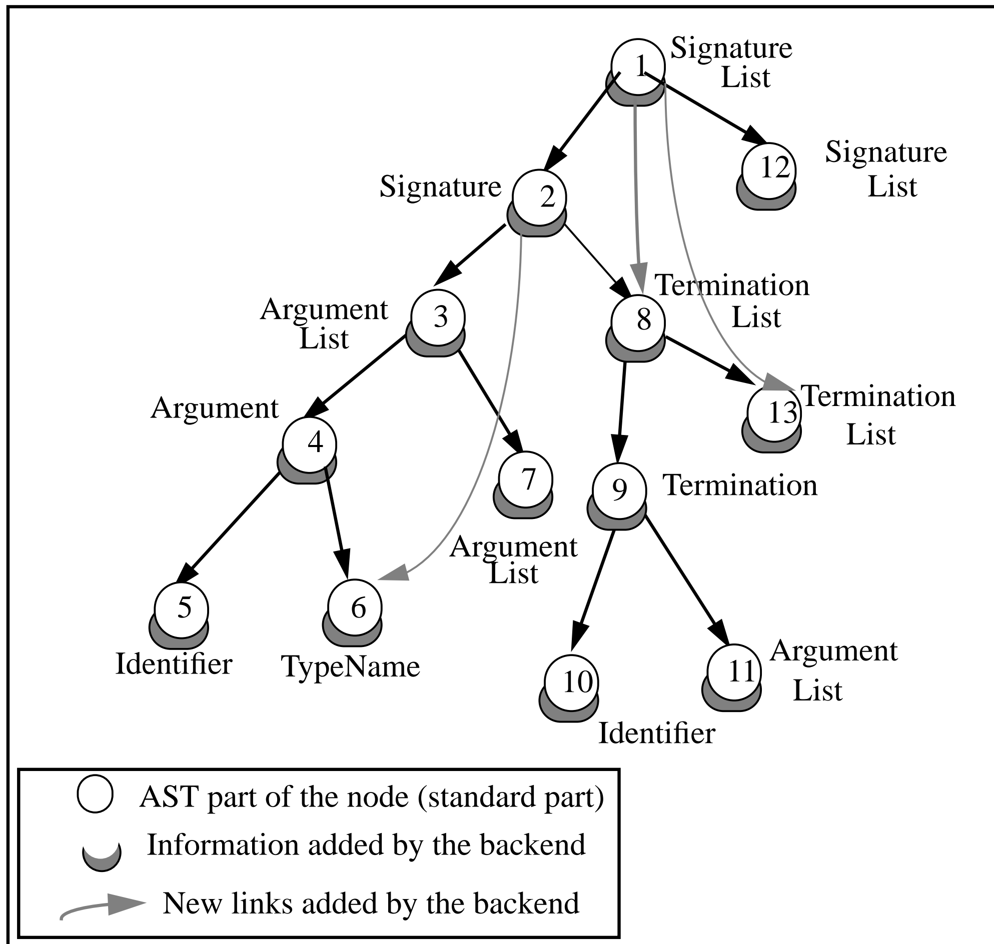
```
1 class BE1_Generator : public AST_Generator
2 {
3    AST_SignatureNode *CreateSignature();
4    AST_TerminationNode *CreateTermination();
5         ....
6    AST_IdentifierNode *CreateIdentifier();
7 };
```

In Figure 1.5:, the backend needs to generate the first argument's type (node 6) before the signature name (node 2). To facilitate the code generation, it adds a link between the signature node and the typeName node. When crossing the AST tree at the code generation phase, the backend can obtain the type name of the first argument without any further search, by using the link added by the backend generator when the AST was generated.

1.7 A DIMMA front-end and an ODP backend

To start with, we have developed a front-end for a DIMMA IDL language, our experimental IDL language for expressing multimedia requirements, and a backend targeted to the ODP runtime environment. The front-end parses a DIMMA IDL service description into an AST, while the ODP backend generates C++ client and server stubs for the ODP runtime environment developed within DIMMA [Otway 95].

Figure 1.5: AST structure augmented by the backend added information



1.7.1 The DIMMA IDL front-end

The DIMMA IDL front-end performs the lexical and syntactic analysis, and uses the AST Generator interface to produce the corresponding AST. The lexical and syntactic analysis is done using lex and yacc [Mason 90], standard UNIX compilation tools. The benefit of using these tools is the neat separation between the rules (lexical and syntactic) and the recognition process itself enabling hence, an easy maintenance and extension of the IDL grammar or its lexical definition rules.

Here is an example of a bank service interface described using DIMMA IDL:

```
1 Pence = Int32 ;
2 Bank : module
3 [
4    Pin = Int32 ; AccountNo = Int32 ; CustomerNo = Int32 ;
```

```
5    Account : interface
6    (
7          credit (amount:Pence)->(Pence)
8          debit (cpin:Pin amount:Pence)->(Pence)->invalidPin()
9                                        ->insufficientFunds(Pence)
10         balance (cpin:Pin)->(Pence)->invalidPin()
11   );
12 ]
```

This description contains the definition of user types (`Pence, Pin`, etc.) and the description of one operational interface `Account` containing three operations (`credit, debit, balance`). The operation `debit` has two parameters (`cpin` and `amount`) and can either terminate normally by returning a parameter of type `Pence` or raise an exception to signal an abnormal termination (`invalidPin` or `insufficientFunds`). The module block `bank` gives a namespace scope to all the types and interfaces described inside it.

The current status of the DIMMA IDL language defines only operational interfaces but is expected to be extended to support stream interfaces once the engineering mechanisms in the runtime are in place.

The C++ class defining the DIMMA IDL front-end looks like:

```
1 class DIMMA_IDL_FE : public FE
2 {
3 public:
4    // Constructors
5    DPL_IDL_FE() {};
6
7    // Member Functions
8    void               IO_Init(char *filename=0);
9    char *             Version(void) { return "1.0"; }
10   void               ProcessArg(char **argv, index_t *indp,
11                                 int argc);
12   parseStatusCode_t  Parse(AST_Node **);
13   void               End(void);
14
15   // The destructor
16   ~DPL_IDL_FE() {};
17 };
```

This class is instanciated once the compilation driver identifies that the DIMMA IDL front-end should be used to parse an input form recognized as expressed in the DIMMA IDL language. The created instance invoked to perform the parsing on the input.

It is possible for the user to specify specific options to a front-end module in the command line by preceding the option flag with the name of the front-end. For example in the command line:

```
> sgen -dimma_verbose 5
```

the option `dimma_verbose` is recognised by the compilation driver as an option meant for the DIMMA front-end and is hence handed to it for further analysis. This allows the toolset to support runtime configurable front-ends.

1.8 A backend for the ODP runtime environment

On the other hand, we have constructed a backend targeting the ODP runtime environment [Otway 95]. The aim is to generate stubs corresponding initially to services described using the DIMMA IDL.

The ODP backend participates in the AST generation by providing a backend generator class, which intercepts calls on the AST Generator to extends the AST by adding information that helps into an easy stub generation.

Given an input file (e.g bank.idl), the ODP backend generates four files:

- A header file (e.g bank.hh) included in the client and server code.
- A client stub file (e.g bank_C.cc)linked with the client code.
- A server stub file (e.g bank_S.cc) linked with the server code.
- A Name file (e.g bank_N.cc) which maps between operation and termination names and their position in their interface description.

The stubs generated by the ODP backend perform the following functions:

- Marshalling/Unmarshalling of parameters.
- Calling the underlying ODP runtime transport routines and buffer management for sending the request message and receiving the reply buffer.
- Dealing with abnormal terminations of an invocation by mainly raising and catching exceptions.

1.8.1 Marshalling/Unmarshalling parameters

The stubs generated by the ODP backend are designed to be as independent as possible from the engineering details such as the transport protocol used to convey messages or the buffer management policy and marshalling/unmarshalling formats. The ODP runtime hides these details from the stubs thus offering a greater stability to the backend and to the stub generator toolset as a whole.

As an example, here is the code generated by the backend, in the client stub for calling the remote operation `credit` on an object which offers an interface of type `Account`:

```
1 Pence odp_Account_Client::credit (Pence a1)
2        throw (odp_EngineeringTermination)
3 {
4    const odp_Releaser buffer = (odp_request(1) << a1).invoke();
6
7    switch (buffer->_response())
8    {
9        case 0: {
10            Pence r1;
11            *buffer >> r1;
```

```
12             return r1;
13         }
14   }
15 }
```

In the example above, the buffer is automatically allocated by the runtime (line 4) when the stub calls `odp_request(1)`[1] which returns the allocated buffer. The marshalling of the first parameter of the operation `credit` is done by using the overloaded operator `<<` in the `buffer` object returned previously. This has the effect of marshalling the parameter `a1` without requiring the stub to supply the type of `a1`. Once all the parameters have been marshalled into the buffer, the operation `invoke()` is called on it which has the effect of conveying the request message, and receiving the reply message. After the invocation, the buffer object contains the reply information and the unmarshalling of the results can start using the overloaded operator `>>`. The stub code is insulated from any references to which format the parameters have been marshalled in, which transport protocols have been used, or how the communication buffers management is done.

1.9 Current Status of the Stub Generator Toolset

We have finished the design and implementation of the toolset framework which enables hosting complying front-ends and backends. Currently this framework contains one front-end (DIMMA IDL) and one back-end (ODP IDL). We are in the process of adding the CORBA IDL as a second front-end to the toolset. This will enable us, to target the ODP runtime given service interfaces described in either DIMMA or CORBA IDL languages. It also allows to increase the independence between the front-end and the backend, given that the ODP backend should be able to use ASTs generated by both front-ends.

We are also extending the DIMMA IDL to support complex types such as structures and the sequences.

Note: The status of the DIMMA IDL language is expected to change thoroughly in the future.

1.10 Conclusions

This report described the DIMMA stub generator toolset, a component of the DIMMA architecture [Li 94] which offers an environment for describing service interfaces in different IDL languages, depending on user preferences and on application requirements. The toolset is also able to generate stubs for any supported runtime.

The toolset architecture was designed in such a manner that the addition of new IDL languages (front-ends) and the support of new runtime environments (backends) can be done easily, given the front-ends and backends respect certain conventions.

1. The parameter “1” corresponds to the position of the operation `credit` in the interface `Account`.

On the other hand, we are experimenting with a new IDL language (DIMMA IDL) to express multimedia interfaces in a similar manner to how operational interfaces are expressed in today's IDLs.

References

[Birrell 84]

A.D. Birell and B.J. Nelson, “Implementing Remote Procedure Calls”, *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, January 1984

[Gibbons 87]

P. B. Gibbons, “ A Stub Generator for Multilanguage RPC in Heterogeneous environments”, *IEEE Transactions on Software Engineering,* vol. SE-13, no. 1, pp. 77-87, January 1987

[Li 94]

G.Li, A. Herbert, D. Otway “An Overview of the Distributed Interactive Multimedia Architecture”, APM.1295, Septembre 1994

[Li 95]

G.Li “An Overview of DIMMA nucleus”, APM.1553, August 1995

[ODP 94]

Open Systems Interconnection, Data Management and Open Distributed Processing, ISO/IEC JTC1/SC21, “Basic Reference Model of Open Distributed Processing - Part 3: Prescriptive Model”, February 1994

[OMG 94]

DEC, HP, HyperDesk Co., NCR Co., Object Design Inc., SunSoft Inc. “The Common Object Request Broker: Architecture and Specification”, OMG Document, 1994

[Otway 95]

D. Otway “ODP C++ Design Overview”, APM.1555, September 1995

[Parrington 92]

G. D. Parrington, “A Stub Generation System For C++”, Technical Report, Department of Computing Science, The University of Newcastle upon Tyne, Newcastle upon Type, NE1 7RU, UK

[Mason 90]

T. Mason, D. Brown “lex & yacc”, Unix Programming Tools, O’Reilly & Associates, Inc, 1990