OODSF: An Object-Oriented Data Specification Framework in a Heterogeneous Computing Environment

by

Jae Woong Hwang

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for the degree of

MASTER OF SCIENCE
in
Computer Science

©Jae Woong Hwang and VPI & SU 1995

APPROVED:

Dr. Dennis G. Kafura, Chairman

Dr. Edward A. Fox

Dr. Scott F. Midkiff

October, 1995
Blacksburg, Virginia

Key words: Object-Oriented Systems, Heterogeneous, Distributed Computing, Interoperability, IDL, ASN.1, Specification Languages, XDR, BER, CDR, Encoding/Decoding, Language Binding, C++,

c.2

LD
5655
V855
1995
H836
c.2

OODSF: An Object-Oriented Data Specification Framework in a Heterogeneous Computing Environment

by

Jae Woong Hwang

Committee Chairman: Dr. Dennis Kafura

Computer Science

(ABSTRACT)

The Object-Oriented Data Specification Framework (OODSF) is a C++ framework to facilitate programming in a heterogeneous distributed environment. Using the OODSF, C++ language bindings of commonly used specification languages, such as Abstract Syntax Notation 1 (ASN.1) and Interface Definition Language (IDL), can be defined. The OODSF defines C++ class libraries for ASN.1 and IDL to simplify the C++ language bindings. Arbitrary application-level IDL and ASN.1 specifications can be translated into C++ representations based on these class libraries. The OODSF contains facilities for encoding and decoding transferred data, allowing interoperability in a heterogeneous distributed system. A general interface is provided to encoding and decoding services so that a flexible choice of an encoding rule can be made. The current implementation of the OODSF contains eXternal Encoding Rule (XDR) and Basic Encoding Rule (BER).

ACKNOWLEDGEMENTS

At first, I would like to give thanks to my advisor and committee chairman, Dr. Dennis G. Kafura, who not only suggested this research but also patiently helped me in the whole process of conducting this research. His clear explanations as well as the methods of abstracting problems were of utmost value to me in both implementing the OODSF and preparing this thesis. I would like to extend my thanks to Dr. Edward A. Fox and Dr. Scott F. Midkiff for serving on my committee as well as for their advice. Their comments and critiques pertaining to this thesis further helped me to express the thoughts and ideas more clearly. I would also like to thank Dr. Verna Schuetz for her guidance, advice, and help in general throughout my graduate study.

I would also like to thank the members of the ACT++ project group. Working with earlier members, Arjun Khare and Venkateswara Vykunta, convinced me of the necessity of my work. I would also like to thank the current members for the ACT++ project group, who will enhance my work by integrating to a distributed working prototype. I would also like to thank Siva Challa, who provided numerous information relating to the OMG and CORBA.

I owe my gratitude to many friends. Especially, I would like to thank Chun Ho Bae for his friendship and support. His grateful advice always helped me out of problems in life. I would also like to thank Eun Jin Hwang for emotional support, who helped me keep a healthy perspective.

Finally, I would like to thank my family. Without their support and inspiration, I doubt if I would have been able to complete this work. I dedicate this thesis to my parents, Ik Nam Hwang and Shin Ja Hwang.

TABLES OF CONTENTS

1 Introduction 1

1.1 Basic Concepts 3

1.1.1 Data Conversions 3

1.1.2 Specification Language 5

1.1.3 Transfer Syntax 5

1.1.4 Language Binding 6

1.2 Goals 8

1.3 Outline 8

2 Approach and Comparison 9

2.1 General Approach using ASN.1 and IDL 9

2.2 OODSF Design Rationale 12

2.2.1 Class Library 12

2.2.2 Type Core 13

2.2.3 Layered Model 14

2.2.4 Benefits 15

2.3 Related Work and Comparison 17

3 OODSF Design and Architecture 21

3.1 Layered Architecture 21

3.2 Separation of Data Representation and Translation 23

3.3 Two-Phase Definitions: Predefinition and Postdefinition 24

3.3.1 Predefinition 24

3.3.2 Postdefinition 25

3.4 ITC, LTC, and ROD 26

4 Predefinition: General Type Definition Library 28
4.1 Overview of GTDL Class Hierarchy 28
4.2 Special Members 31
4.3 Basic Types 32
4.3.1 Definition and Specialization of Basic Types 34
4.3.2 Basic Types and Value Representations 35
4.3.3 Type Conversion 37
4.4 Constructed Types 38
4.4.1 Array Type 39
4.4.2 Sequence Type 42
4.4.3 Struct Type 48
4.4.4 Union Type 52
4.5 Any Type 57
4.5.1 Special Operators 58
4.5.2 _copy and _assign 59
4.6 ITC, LTC, and ROD 60
4.6.1 ITC 60
4.6.2 LTC 61
4.6.3 ROD 62
4.7 LANGDEF 62

5 Predefinition: General Transfer Syntax Interface and Implementations 64
5.1 Overview of GTSI class hierarchy 64
5.2 Related Work 65
5.3 Abstract GTSI Class 65
5.4 Template TSI<T> Class 67
5.5 DStream: Data Stream 68
5.6 TSI<XDR> and TSI<PPBER> 69

6 Postdefinition: IDL and ASN.1 Class Libraries 71
6.1 Issues 71
6.2 Type Aliases 73
6.3 IDL Class Library 74

6.4 ASN.1 Class Library 76
6.4.1 ASN.1 Tag and Tagged Types 76
6.4.2 Tagged Type Tuples 77
6.4.3 Goals and Strategies 78
6.4.4 ASN.1 Base Type Aliases 79
6.4.5 Template Class For Tagged Type Tuples 79
6.4.6 Discussion 81

7 Functional Testing 83
7.1 Testing Environment 83
7.2 Testing IDL Class Library with GTSI 86
7.3 Testing ASN.1 Class Library with GTSI 88
7.4 Mixture: GTDL, ASN.1, and IDL 88
7.5 Discussion 89

8 Conclusions and Future Work 90

REFERENCES 94

LIST OF FIGURES

Figure 1.1 Global View 3
Figure 1.2 Data Conversions 4
Figure 2.1 General Application Program Development 10
Figure 2.2 Client and Server Communication 11
Figure 2.3 Layouts of OODSF 14
Figure 4.1 GTDL Class Hierarchy 29
Figure 4.2 Abstract _Type class 30
Figure 4.3 _Integer template class 33
Figure 4.4 _Integer Type Class Specialization 33
Figure 4.5 _Boolean Type 37
Figure 4.6 _Array Type 40
Figure 4.7 _Sequence Type 44
Figure 4.8 _Sequence: List Data Structure 45
Figure 4.9 _Struct Type 49
Figure 4.10 An example: C++ struct 49
Figure 4.11 An example: GTDL _Struct 50
Figure 4.12 _Struct: List Data Structure 51
Figure 4.13 _Union Type 53
Figure 4.14 _CASE Class 54
Figure 4.15 An example: GTDL _Union 55
Figure 4.16 _Union: List Data Structure 56
Figure 4.17 _Any Type 58
Figure 4.18 LANGDEF Class 62
Figure 5.1 GTSI Class Hierarchy 64
Figure 5.2 The GTSI class 66
Figure 5.3 The TSI<T> class 67

Figure 5.4 The DStream 69
Figure 6.1 IDL Specification: class_info 75
Figure 6.2 C++ Representation: class_info 75
Figure 6.3 UNIV.h: ASN.1 Data Type Definitions 78
Figure 6.4 BUILTIN Template Class 80
Figure 7.1 IDL test description 85
Figure 7.2 C++ Translation for datatypes 86
Figure 7.3 ASN.1 test description 87

LIST OF TABLES

Table 1.1 IDL/ASN.1 Data Types 7
Table 2.1 Language Binding Summary 17
Table 4.1 GTDL Type Class Definitions 32
Table 4.2 GTDL Basic Types and C++ Value Representation 34
Table 4.3 Constructed Types and Definitions 38
Table 4.4 ITC kinds 61
Table 4.5 ITC to ASN.1 Tag Conversion 63
Table 6.1 IDL/ASN.1 Type Aliases 72
Table 7.1 Testing Results 89

Chapter 1

Introduction

The existence of multiple data representations in a heterogeneous distributed computing environment is a barrier to achieving interoperability among different computer systems. Since each computer system may have its own way of representing data internally, agreements and conversions are necessary to allow different computer systems to share data with one another [COM93, TAN89].

To cope with multiple data representations in a distributed environment, a *specification language* and a *transfer syntax* are used. The specification language defines the structures of commonly understood data among communicating machines, and it incorporates a transfer syntax to define the physical representations of this data in the network.

To use a specification language in a programming environment, a *language binding* must be defined. In a language binding, a specification described by a specification language is translated into a representation in a programming language. However, defining a language binding for a specification language is complex. Application developers

demand convenient data manipulations as well as abstractions that hide the details of data sharing, allowing an application-centered environment for building distributed applications. Structuring and incorporating a language binding along with a transfer syntax into a distributed prototype is even more complex.

Distributed systems often rely on intensive use of a translator to cope with the complexity of defining a language binding. However, designing and implementing such a translator is expensive, and it may be restricted to only one specification and one transfer syntax among many options. To incorporate either a different specification language or an alternative transfer syntax, an almost complete redesign and reimplementation may be necessary even though language bindings are aimed at a common programming language.

A carefully designed class hierarchy in an object-oriented language such as C++ provides an alternative solution to multiple specification languages and transfer syntaxes. An object-oriented language allows a highly sophisticated structuring mechanism for synthesizing software systems [LAV93]. Object-based descriptions using classes, templates, inheritance, and polymorphism allow abstractions as well as flexible and convenient manipulations of complex data structures, permitting relatively easy extensions of predefined definitions.

This thesis presents a C++ framework for structuring a general data specification system which can accommodate different specification languages as well as various transfer syntaxes. Using this framework, C++ language binding of a specification language is simplified, allowing a flexible choice among different transfer syntaxes.

1.1 Basic Concepts

This section describes the basic concepts for various issues including data conversion problems, specification languages, transfer syntaxes, and language bindings. Figure 1.1. shows a global view of how these concepts are related to one another in a C++ programming environment.

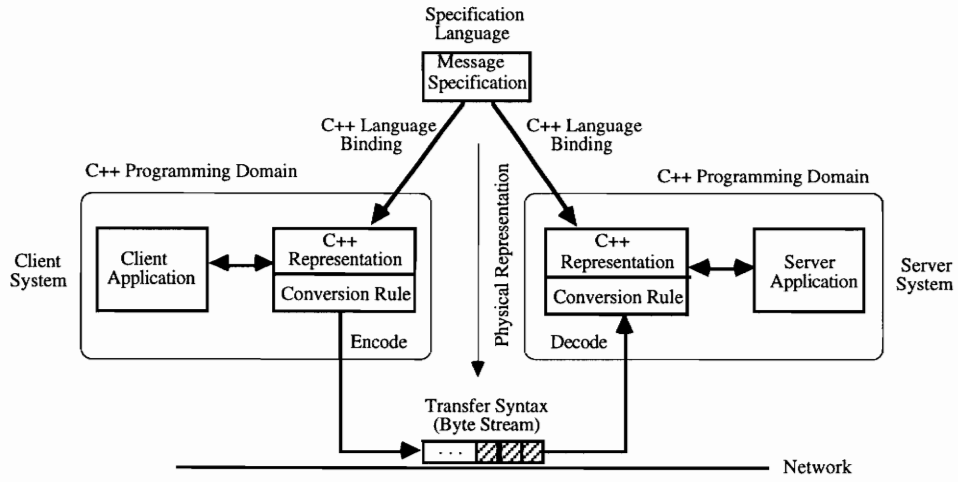

Figure 1.1 Global View

1.1.1 Data Conversions

Fundamentally, there are two approaches for exchanging data among different computer systems to achieve interoperability: *asymmetric data conversion* and *standard representation*. Figure 1.2 illustrates these two approaches.

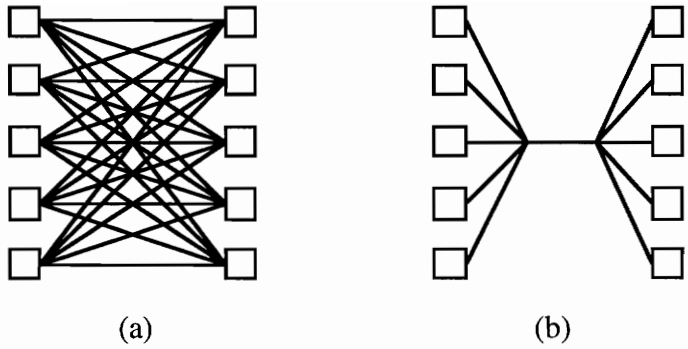

Figure 1.2 Data Conversions: (a) Asymmetric Data Conversion Approach, (b) A Standard Representation Approach

At one extreme, every machine can embed knowledge of the internal format used by every other computer's architecture. This approach of direct conversions between a client's representation and a server's representation is called asymmetric data conversion because one side or the other performs conversion [COM93]. However, this method is inherently complex due to the multiple conversion routines for each possible client-server pair. For example, with N different types of machines in the network, N(N-1) different conversion routines are necessary to allow interoperability [TAN89].

In a standard representation approach, each machine only knows how to convert the internal data to and from a standard representation. When a machine A needs to transfer data to a machine B, machine A converts the data to the standard representation before sending, and machine B converts the received data in the standard representation to a local representation. With this standard representation approach, when N different representations are possible, only 2N conversion routines are necessary for converting to and from a standard representation. This significantly reduces the complexity and it provides a viable environment for interoperable heterogeneous computing systems.

1.1.2 Specification Language

An approach for coping with the standard representation to achieve interoperability is by a formal data specification language. The task of a specification language is to describe data types in an architecture-independent fashion. Such a specification language defines the structure of the transferred data and its architecture-independent external representation [HAS92]. Communicating machines agree to a message structure using this specification language.

Abstract Syntax Notation One (ASN.1) [ISO87a] and Interface Definition Language (IDL) [OMG91] are commonly used specification languages for describing data. ASN.1 has been standardized by the International Standard Organization (ISO) and the International Telephone and Telegraph Consultative Committee (CCITT). It is currently used in various communication protocols [ANS92, HAS92, FED90, DAV90]. The Object Management Group (OMG) defines the Common Object Request Broker Architecture (CORBA) along with an interface definition language, called OMG IDL [OMG91]. IDL is a language that provides mechanisms for defining types, objects, and operations in heterogeneous distributed systems. Table 1.1 shows available data types of IDL and ASN.1.

1.1.3 Transfer Syntax

The data described by the specification language are converted to and from the network-level byte stream at the transmitting and receiving ends, respectively. The canonical representation of this data is called the transfer syntax. Encoding (marshaling) and decoding (unmarshaling) are used in converting the data described by the specification

languages to and from the transfer syntax, respectively [STE90, TAN89]. Some commonly used encoding rules are Basic Encoding Rule (BER) [ISO87b] and Packed Encoding Rule (PER) by ISO/CCITT [ISO92], eXternal Data Representation (XDR) by Sun Microsystems [SUN87], Network Data Representation (NDR) by HP/Apollo [KON90], and Courier by Xerox [XER81]. In addition, the OMG defines Common Data Representation (CDR) in the CORBA II specification to allow interoperability among different CORBA implementations [OMG95]

1.1.4 Language Binding

To incorporate a specification language into a programming environment, a language binding must be defined. In the language binding, a type defined by the specification language is mapped to a type representation in a selected programming language (target language), using one or more constructs available in the target language. For example, IDL can be used in a C programming environment by mapping an IDL specification to a C representation [OMG91]. Similarly, to use IDL in an object-oriented programming environment, such as C++, a mapping from IDL to C++ is defined [OMG92, OMG94, HDC93, INS93, HPV93, PMC94a, PMC94b].

The complexity of defining a language binding for a specification language varies depending upon the target language. The experience of this and related research is that a language binding for C++ is less complicated compared to that for C since classes and templates in C++ provide richer type definition mechanisms [STR91, MUL93, LAV93, KHE94, KLM94, LON94].

Table 1.1 IDL/ASN.1 Data Types

IDL	ASN.1
boolean	BOOLEAN
short long unsigned short unsigned long	INTEGER
float double	REAL
char	
octet	
	BIT STRING
	OCTET STRING
string	Character Strings NumericString PrintableString TeletexString(T61String) VisibleString(ISO646String) IA5String GraphicString GeneralString
object reference	OBJECT IDENTIFIER
any	ANY
array	
sequence	SEQUENCE OF
struct	SEQUENCE
	SET (OF)
union	CHOICE

1.2 Goals

The main objective of this research is to build an Object-Oriented Data Specification Framework (OODSF), that will:

- accommodate different specification languages including IDL and ASN.1,
- simplify the translation from the specification language to the C++ language binding,
- provide a natural-to-use binding, and
- cope with multiple encoding rules including XDR and BER.

These goals are achieved by structuring the OODSF to include an extensible set of data types and a general interface for different transfer syntaxes. The extensible set of data types accomplishes the first three goals. The last goal is accomplished by the general interface.

1.3 Outline

This thesis is organized as follows. Chapter 2 presents a general approach for using ASN.1 and IDL, and discusses the OODSF design rationale as well as a comparison with related work. Chapter 3 describes the details of the OODSF design and architecture. Chapters 4 and 5 discuss implementations for building various components of the OODSF. Chapter 6 presents ASN.1 and IDL class libraries for extending the OODSF. Chapter 7 presents the test cases for the OODSF and IDL/ASN.1 class libraries. Chapter 8 presents conclusions and outlines further work.

Chapter 2

Approach and Comparison

2.1 General Approach using ASN.1 and IDL

A distributed application consists of separate entities which reside on different nodes of the network and cooperate to achieve a common goal [BEV93]. The model embedded internally in a particular distributed system is generally described as a client/server model, where a client requests a service and a server provides the requested service to the client. A supporting infrastructure of a particular distributed system is the realization of a client/server model with its own communication paradigm and environment. The main parts of the infrastructure are the underlying communication protocols between the client and server. The message structures in the underlying communication protocols and those used by the applications are usually described in terms of a specification language such as ASN.1 and IDL.

A conventional approach for using an ASN.1 or IDL specification is by preprocessing them to produce corresponding components in a high-level language such as C++ so that the specifications can be understood and embedded in a programming

environment [OMG91, SHI92, PMC94a, PMC94b, KHK94]. The resulting C++ code, via *stub* and the application specific code can be compiled and linked with the system-dependent runtime library to implement the client and server, as depicted in Figure 2.1.

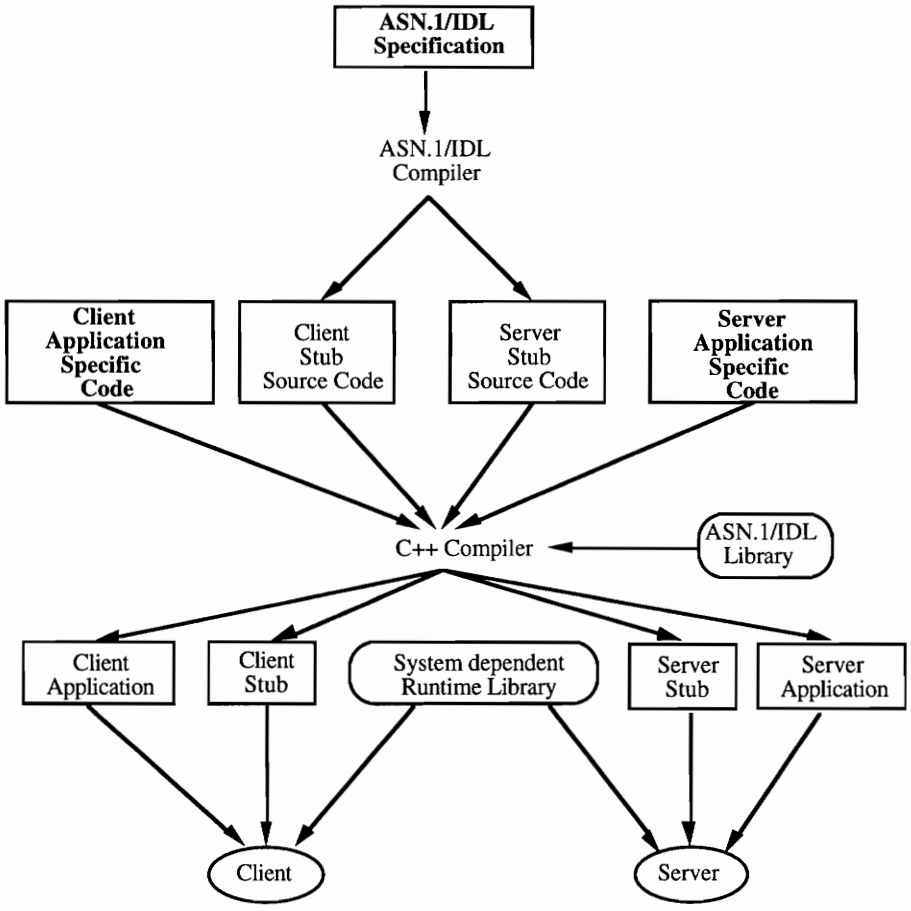

Figure 2.1 General Application Program Development

An application programmer specifies the application's message structures using ASN.1 or IDL when the application is developed. In the case of IDL, since it allows the specifications of operation invocation interfaces in addition to the specification of message structures, the whole process of stub source code generation can be automated. Since

ASN.1 does not allow the specification of interface declarations, this can be done manually or by employing some interface description language for complete automation. The generated stub source code and the application specific code can be compiled using a translator and linked with the system-dependent runtime library to create an executable client or server.

Client and server messages can be sent by means of the stubs using the functionality of the runtime library specific to the selected model. The runtime library provides the services for contacting a broker, if one exists, locating the server, establishing the connections, controlling synchronous or asynchronous communications, etc. Each message is marshaled and inserted into the network using a selected encoding rule, such as XDR, BER, or NDR. Similarly the server decodes and unmarshals the messages. Figure 2.2 illustrates this process.

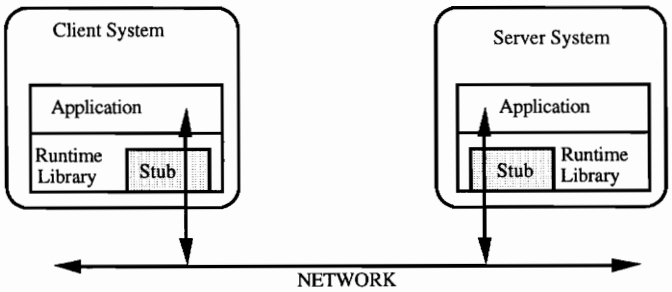

Figure 2.2 Client and Server Communication

2.2 OODSF Design Rationale

This section discusses the design rationale behind the OODSF model and discusses the benefits of using the OODSF as well as a comparison of the OODSF with other related systems.

2.2.1 Class Library

While it is essential to define a language binding for a specification language, the language binding is inherently complex due to a translation of one language form to another. This translation must preserve the expressiveness of the specification language while the translated representation should be natural and easy to use in the target language environment. To satisfy these properties, it is required that the translated representation must provide appropriate mechanisms for data manipulations and conversions to and from a transfer syntax.

A class library in a target language can reduce the complexity of the translation from a specification language to a target language. This class library provides abstractions which hide most of the details for both data manipulations and conversions, freeing the translation from dealing with these details. Therefore, the translation from a specification language to a target language is simplified because the translation is a high-level systematic definition using the methods provided by the class library. For example, a C++ class library for IDL can be used for simplifying the translation from a IDL specification to a C++ representation. The class library can implement the details of manipulating data structures

as well as the encoding rule for converting to and from the transfer syntax. As the implementation details are hidden within the definition of the class library, the translation is simply how to combine the methods provided by this class library. Similarly, a C++ class library for ASN.1 can help the translation of a ASN.1 specification to a C++ representation.

2.2.2 Type Core

While it is possible to provide a completely separate class library for either ASN.1 or IDL data types, a better approach is to build a general framework containing a common set of data types, via a *type core*, and map either IDL or ASN.1 data types to this type core. This approach reduces the complexity of translating multiple specification languages to a common target language because the type core simplifies the definition of class libraries for ASN.1 and IDL. The type core eliminates the complete reimplementations of class libraries when the translation is aimed at a common target language.

To define a type core, the semantic difference of similar data types must be resolved. For example, IDL and ASN.1 differ in that IDL provides a set of representable data types while ASN.1 provides representation-independent data types. IDL has a set of integer types including **short**, **long**, **unsigned short**, and **unsigned long**, depending upon the sizes that they can represent. ASN.1 provides an integer type, **INTEGER**, which can represent an integer value of any size. However, sharing of the common type core can be still achieved by restricting some semantics of ASN.1. This is reasonable since the programming environment for the target language usually restricts the representations to a fixed set of representable data types. Consequently, translating the ASN.1 **INTEGER** type to the C++ **long** type, which can represent the largest integer value in C++, is natural.

Similarly, the various integer types of IDL are the subset of available C++ integer types. This makes it possible for both integer types in ASN.1 and IDL to use the common definitions available in C++.

The type core must accommodate the non-compatible features of ASN.1 and IDL as well, allowing the extensions of the type core for defining both ASN.1 and IDL class libraries. For example, a fixed-length array is provided by IDL but does not exit in ASN.1. To extend the type core to the IDL class library, this fixed-length array must be included in the definitions of the type core.

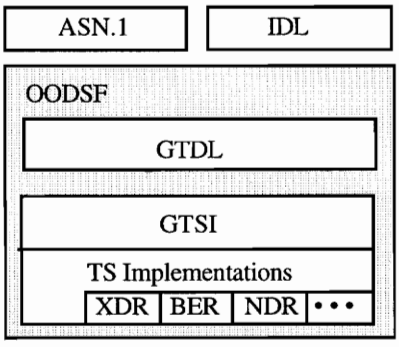

Figure 2.3 Layouts of OODSF

2.2.3 Layered Model

The OODSF is designed as a layered model to achieve levels of abstractions. It consists of General Type Definition Library (GTDL), General Transfer Syntax Interface (GTSI), and transfer syntax implementations as shown in Figure 2.3. The GTDL defines the type core which includes common as well as non-compatible features for both ASN.1

and IDL. The GTSI is an interface for the encoding and decoding services supported by an arbitrarily selected transfer syntax (currently, XDR or BER). The transfer syntax implementations contains the code for the selected encoding rules. On top of this layered model, ASN.1 and IDL class libraries are defined using the type core (GTDL). The details are discussed further in Chapter 3.

2.2.4 Benefits

By predefining the type core, the definition of C++ language binding for a specification language is simplified because:

- the type core allows the definition of a C++ class library for a specification language which is a simple extension using either aliasing or inheritance, and
- this class library simplifies the translation of a user-defined specification to a C++ representation since the translation is based on the definitions in this class library, providing abstractions which hide the details of both data manipulations and conversions.

Use of the type core maximizes the degree of reusability because it can define different specification languages including ASN.1, IDL, and possibly other specification languages. It also enhances maintainability because the translations based on the class library allow changes of implementation details within the type core, without affecting the translation schemes for a language binding. The type core allows new base classes for representing different data types to be added flexibly.

The OODSF copes with multiple transfer syntaxes by separating representations from encoding/decoding schemes. It provides a general interface to different encoding rules so that the identification of an encoding rule can be deferred, allowing a flexible choice among different transfer syntaxes. As a result, it allows the encoding/decoding methods to be changed dynamically without a complete reimplementation of the translator.

The OODSF extensively utilizes classes and templates as well as list-oriented data structures in its definitions. The corresponding result is that the translated representations are easy and natural to use, much as in a normal C/C++ programming environment. This allows both system developers and application programmers to freely manipulate data with less complexity.

To remain independent of a specific runtime environment, the OODSF does not include interfaces or code dependent on a specific runtime system or communication paradigm. While this enhances the portability of the OODSF, it does not limit its functionality. In contrast, IDL describes not only data type declarations, but also interface declarations including operations, which can be invoked by clients [OMG91]. This is a distinguishing feature, not present in ASN.1. The interface description is generally translated into stubs for a system-dependent runtime environment. The implementation details for the generated stub are subject to the control structure inherent in a particular communication paradigm. For example, orbeline [PMC94a, PMC94b], DCE (Distributed Computing Environment) [SHI92, BEV93, WEI93], NCS (Network Computing System) [KON90], and ACT++ [KHA94, KHK94] use IDL for describing the data types and interfaces from which the corresponding stub are generated using translators, according to their own runtime environment and communication paradigm.

2.3 Related Work and Comparison

Previous research has defined language bindings for ASN.1 and IDL. Table 2.1 summaries the characteristics of these systems [SAM94, SAM93, ROD90, BBN95, LON94, KON90, SHI92, OMG91, OMG94, OMG95, PMC94a, PMC94b, INS92, HDC93].

Table 2.1 Language Binding Summary

Name	Prog. Language	Spec. Language	Encoding Rule	Category
snacc	C/C++	ASN.1	BER	stand-alone
ISODE	C	ASN.1	BER	integrated
BBN	C++	ASN.1	BER	stand-alone
caty	C++	ASN.1	flexible	stand-alone
NCS	C	IDL (NIDL)	NDR	integrated
DCE	C	IDL	NDR	integrated
CORBA	C/C++	IDL	*	integrated
orbeline (CORBA)	C++	IDL	*	integrated
INS mapping (CORBA)	C++	IDL	*	integrated
HyperDesk mapping (CORBA)	C++	IDL	*	integrated
CORBA II	C/C++	IDL	CDR	integrated
OODSF	C++	IDL/ASN.1	flexible	stand-alone

* implementation-specific

Common encoding rules used in these systems include BER, NDR, XDR, and CDR. Previous systems can be categorized as either stand-alone or integrated depending upon whether they make assumptions about the underlying communication environment.

Stand-alone systems make no assumption about the communication environment. Integrated systems define the language bindings as a part of the entire distributed prototype, allowing the language bindings to incorporate the knowledge of communications. For instance, snacc is an example of the first case. The orbeline is an example of the second case in which the C++ language binding for IDL is defined as a part of its CORBA implementation.

The rest of this section compares OODSF to the above systems. The first three paragraphs describe how the OODSF is different from all or some of the above systems. The relationship of the OODSF to its predecessor, caty, is also discussed.

First, while the OODSF is designed to cope with both ASN.1 and IDL as well as different encoding rules, all systems in Table 2.1 are specific to one specification language and one encoding rule. For example, snacc and BBN define C++ language bindings for ASN.1 while they adopt BER for encoding/decoding. Similarly, DCE uses IDL for a specification and NDR for encoding/decoding. Among those listed in Table 2.1, caty is the only one which allows several choices of encoding rules even though it is still restricted to ASN.1 as well as the encoding/decoding of data types only available in ASN.1

Secondly, the OODSF attempts to simplify the translation by more focusing on structuring mechanisms, resulting in less use of a translator. In contrast, several systems listed in Table 2.1 rely on intensive use of translators for defining language bindings such as in ISODE and orbeline. This translation method results in a long listing of a target language code, which makes it complex to understand as well as almost impossible to translate manually. As an example, the orbeline produced about 1000 lines of C++ code for about 60 lines of an IDL test description, which only contained data parts to exclude the

translation of the definitions related to a runtime library. For the identical IDL description, the OODSF required approximately 100 lines of C++ code.

Third, while the OODSF emphasizes natural appearance and easy use, many translations such as orbeline and INS mapping require somewhat unnatural C++ representations to manipulate data. For example, for a given IDL specification,

```
struct typeA {
   short i;
   long  j;
};
```

their translated C++ representations require **typeAvar.i()** to get a value and **typeAvar.i(3)** to set a value (3). As opposed to this, the OODSF allows the manipulations of data, much as in a normal C/C++ **struct** type, such as **typeAvar.j = typeAvar.i + 3**.

From an architectural point of view, the OODSF is an evolution of the ASN.1/C++ class library developed for the caty in the following aspects:

- objectification,
- list-oriented data structure, and
- separation of data representations and translation.

Each data type is implemented as a C++ class and each instance is expressed as an object so that the overall type structures and specifications are expressed as the interactions and behaviors of typed objects. The elements of a constructed type are managed through the manipulations of a list so that the constructed type itself serves as a list data structure. The

data representation in C++ and the transfer syntax are separated so that an arbitrary transfer syntax can be applied to allow maximum flexibility. As a descendent, the OODSF also utilizes this concepts in its architecture. However, compared to the ASN.1/C++ class library, the OODSF defines a different class structure as well as different class definitions since the OODSF is designed to incorporate both IDL and ASN.1.

To summarize, the OODSF is different from other systems in that it can cope with multiple specification languages and multiple transfer syntaxes. The translation is relatively simple, and the translated C++ representations are natural and easy to use compared to several other systems. The OODSF is similar to the caty in that it can accommodate different transfer syntaxes; however, the caty is only restricted to encode/decode data types for ASN.1 while the OODSF is expanded to encode/decode more various data types including both IDL and ASN.1.

Chapter 3

OODSF Design and Architecture

The main objective of this work is to provide a flexible framework for representing C++ data types specified in either IDL or ASN.1. This chapter discusses the overall design issues of the framework defined by the OODSF.

3.1 Layered Architecture

Figure 2.3 in Chapter 2 shows the overall OODSF architecture. The main components of the OODSF consist of General Type Definition Library (GTDL), General Transfer Syntax Interface (GTSI) and the implementation of transfer syntaxes. ASN.1 and IDL class libraries are layered on top of the OODSF.

GTDL

The GTDL is a class library which contains various data type definitions. The GTDL is generic in that no particular specification language feature is included. For example, the class names in the GTDL are not specific to ASN.1 or IDL. The types defined in the GTDL are C++ classes, which permit the definition of abstract data types so that a GTDL type can be treated like a built-in C++ type in an application level

programming. For example, a GTDL **_Short** type can be used similarly to the C++ **short** type in an arithmetic operation. Furthermore, since it is defined as a class, it allows a flexible type extension by means of specifying a type alias with typedef or by inheritance.

GTSI

The GTSI component of the OODSF provides an interface for various encoding rules. The transferred data, stored in an object of the GTDL, are encoded into a network-level data stream by invoking a routine in the GTSI. Similarly, the receiving side decodes the network-level data stream into a local representation and stores it into an appropriate type object of the GTDL using an interface provided in the GTSI.

The GTSI is defined as an abstract C++ polymorphic base class. Derived classes conforming to the GTSI interface implement a selected encoding rule. Consequently, the GTSI provides an interface to access the selected encoding or decoding services. For example, a TSI<XDR> object provides XDR encoding/decoding services. In this case, the GTSI interface could invoke the XDR encoding/decoding services. Similarly, if a TSI<BER> object is used, the GTSI interface would invoke the BER encoding/decoding services.

Transfer Syntax Implementations

The specific transfer syntax, such as those of BER, XDR, and NDR, are layered at the bottom of the architecture. Each transfer syntax can be implemented separately with its own rules as long as it provides the interface required by the GTSI. Encoding/decoding services conforming to the GTSI interface are accessed by type objects of the GTDL in an application.

ASN.1/IDL

Class libraries for ASN.1 and IDL are layered at the top of the OODSF. The C++ representations of data types in ASN.1 and IDL are defined in terms of the available types of the GTDL. Since all type definitions of the GTDL are described as C++ classes, the redefinition of ASN.1 and IDL data types can be done using type aliases with typedef or using inheritance. The data types defined in these class libraries know how to convert to and from a transfer syntax using the GTSI.

3.2 Separation of Data Representation and Translation

A common approach to implementing a data specification system with a transfer syntax is to integrate a selected encoding rule within a given specification system. This approach encodes and decodes for only one transfer syntax. Common examples are cohesive implementations of ASN.1 and BER or IDL and NDR. However, each encoding rule exhibits different characteristics with respect to performance and expressiveness. For example, the BER's inherently flexible scheme for the variable length encoding/decoding of data results in high expressiveness regardless of the machine-dependent representation. However, more runtime overhead is incurred due to the extra encoded information such as the tag and length. XDR, while supporting more limited encoding/decoding compared to BER, shows superior performance over BER in benchmark comparisons [SAM94].

The OODSF adopts an architectural design in which data representation and translation (encoding/decoding) are separated to allow a flexible choice among different encoding rules. As opposed to a fixed pair of a data specification system and an encoding rule, this approach allows the OODSF to take advantages of various encoding rules

including BER and XDR [MUL93]. As the representation is loosely coupled with the translation, the binding between them can be deferred and, hence, the separate implementation and adoption of various encoding rules are possible. This also enhances the notion of abstraction and flexibility, which makes the OODSF more manageable and reusable.

3.3 Two-Phase Definitions: Predefinition and Postdefinition

The OODSF adopts the strategy of *predefinition* as a *generalization* step and *postdefinition* as a *specialization* step [LAV93, LIS86]. In predefinition, the GTDL, the GTSI, and transfer syntax implementations define a data specification system in C++. In postdefinition, the OODSF is specialized to either ASN.1 or IDL using the predefinition result.

3.3.1 Predefinition

To generalize the functionality defined in the predefinition phase, the OODSF defines type constructs with the following philosophy.

- No specific knowledge of a particular data specification language, such as encoded type information or type names, is included. For example, all type names defined in the type system of the OODSF are described in terms of generic names, e.g. **_Short**. The ASN.1 tag information is not embedded in the OODSF in the predefinition phase, but rather it is deferred to the postdefinition phase when the OODSF is specialized for ASN.1.

- A sufficient number of type constructs are provided so that the type system in the OODSF serves as a universal set, from which a smaller subset for the specialized type system can be redefined for a particular data specification language.

The predefined definitions provide the mechanisms such that:

- each instance of a data type can hold value(s), and
- each data type can be manipulated like a built-in type because the associated accessing methods such as assignment operators and arithmetic operations are provided appropriately.

In addition, each data type embeds the capability of serving as a data specification system for a distributed application because each data type in the GTDL is provided with specific routines, **encode** and **decode**, which know how to invoke the interface routines defined in the GTSI system.

3.3.2 Postdefinition

Definitions of the types in a particular data specification language is achieved by either creating type aliases with typedefs or using inheritance. Creating a type alias is the simplest type extension which allows a different type name with the same functionality. Inheritance in object-oriented programming allows extra information to be added to the existing definition. For example, tag information can be added to the predefined definition by inheritance (The details are discussed further in Chapter 6). Specialization using type

aliasing and inheritance from the type system of the OODSF is sufficiently powerful to describe the data types of a particular specification language.

Data types of both IDL and ASN.1 in C++ are specialized through this postdefinition phase using the OODSF. These languages include the functionality of OODSF through type aliasing. ASN.1 adds tag information through inheritance. The implementation details are discussed further in Chapter 6.

3.4 ITC, LTC, and ROD

The GTDL includes several special classes, Internal Type Code (ITC), Language-defined Type Code (LTC), and Required/Optional/Default (ROD) in addition to the classes for data types. The ITC is a type code used internally in the OODSF to identify the type of an object. Every data type defined in the GTDL is assigned a unique number. Hence, each type object embeds self-knowledge of the ITC kind so that it knows what kind of data type it is in the GTDL. The LTC is a type code which can be defined by a language built on top of the OODSF. For example, the ASN.1 tag information can be stored in the LTC. This is particularly useful when the GTSI is bound with the BER encoding rule since BER requires the ASN.1 tag information to encode and decode data. When the language defined with OODSF is ASN.1, the OODSF provides a mechanism for converting the LTC to an ASN.1 tag so that the GTSI system encodes correctly with BER. If BER is used to encode a data type not defined by ASN.1, the conversion is taken from the ITC so that the corresponding ASN.1 tag is produced for BER encoding. To summarize:

When the BER encoding rule is used,

Conversion(LTC) --> AsnTag, if ASN.1 language defined

Conversion(ITC) --> AsnTag, otherwise.

Specific details for the conversions are presented in Chapter 5. The ROD denotes an option among required, optional, and default, for each element of a constructed type. This class is specifically used for ASN.1 language definition because a constructed type in ASN.1, such as **SEQUENCE**, allows an element to be required, optional, or default when it is encoded and decoded.

Chapter 4

Predefinition: General Type Definition Library

The General Type Definition Library (GTDL) system is a collection of C++ definitions of data types in the OODSF. It presents a set of basic types as primitive building blocks and a set of type constructors for structuring more complex data types. The GTDL system provides a sufficient set of type constructs so that the definition of a data specification language in C++ is possible using the GTDL system.

4.1 Overview of GTDL Class Hierarchy

A type in the GTDL system is described in terms of a C++ class. An instance of a GTDL basic type associates a value represented by either a built-in type or a predefined C++ class. A GTDL constructed type such as **_Struct** type associates a data structure and its accessing mechanisms for manipulating its elements.

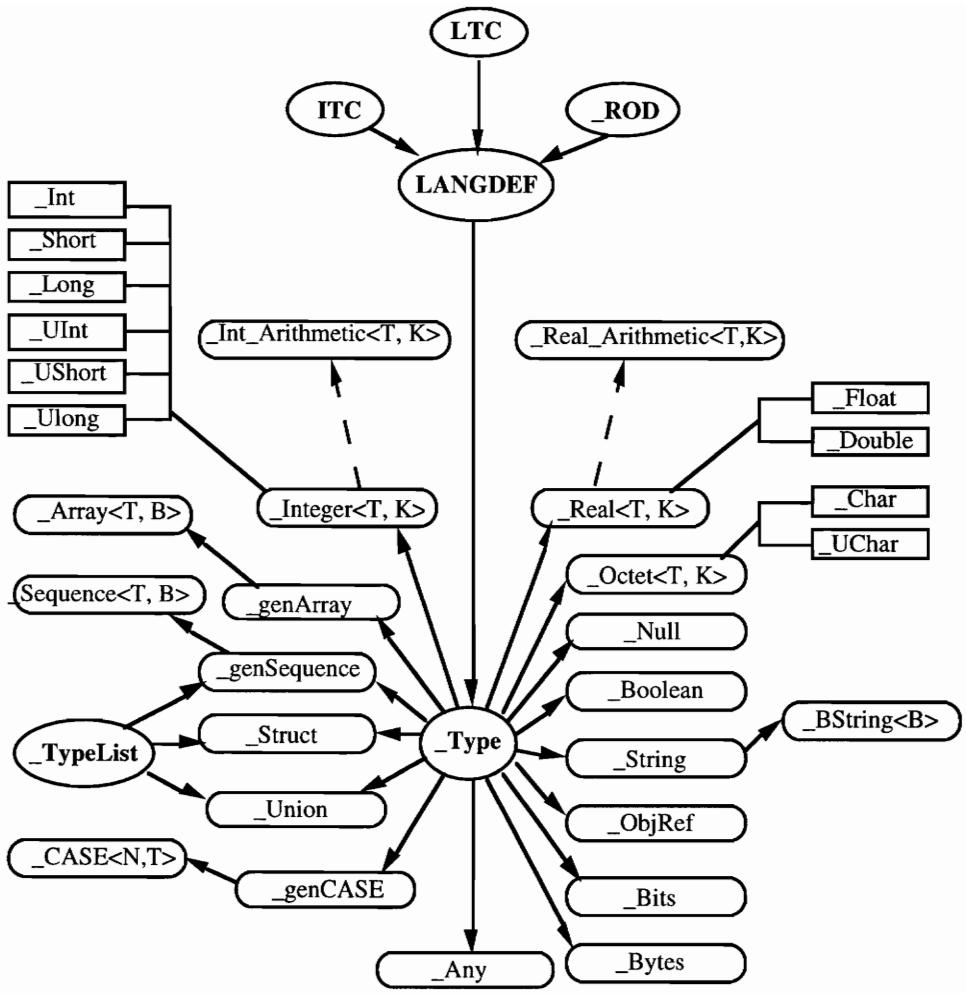

Figure 4.1 GTDL Class Hierarchy

Figure 4.1 depicts the overview of the GTDL class hierarchy. A plain arrow indicates an inheritance relation. A dashed arrow depicts a friend class relation. An undirected line indicates a typedef, or an instantiation. For example, **_Type** is an abstract base class, from which all data types inherit. The **_Int_Arithmetic** is the friend class of **_Integer**, which provides overloaded arithmetic operators for the **_Integer** class. The **_TypeList** provides list manipulation services. The **_Struct**, **_Union**, and **_Sequence** inherit from this class to associate list-oriented data structures for manipulating their

elements. The **_Short** is an instantiation of the **_Integer** with the C++ **short** built-in type for the value holding member. This allows the representation-independent **_Integer** class to be instantiated with a **short** type, resulting in the **_Short** type. The details are discussed further in the rest of this chapter.

The abstract type class **_Type** inherits from three different classes, **ITC** (Internal Type Code), **LTC** (Language-defined Type Code), and **ROD** (Required/Optional/Default). The **ITC** is the type code information managed internally in the GTDL system. The **LTC** is the type code (tag), which can be defined by a particular data specification language built on the GTDL system so that it can manage its own tagging information. The **ROD** (Required/Optional/Default) is an ASN.1 specific class, denoting an option for an element in a constructed type such as **SEQUENCE**. Each element of the **SEQUENCE** is specified with one of the above options. The details are discussed further in the rest of this chapter.

```
class _Type : public LANGDEF {
public:
        ...

        virtual int        encode(GTSI&) = 0;
        virtual int        decode(GTSI&) = 0;

        virtual void       _assign(_Type& v) = 0;
        virtual _Type*     _copy() const  = 0;
};
```

Figure 4.2 Abstract **_Type** class

4.2 Special Members

Figure 4.2. illustrates the definition of the abstract **_Type** class. It contains pure virtual methods to allow the binding of proper implementations in the inherited classes so that the correct reference will be made during runtime. All child classes must define the implementations. All GTDL types include the following special methods in their class definitions.

- The GTSI related routines: **encode** and **decode**.

  The **encode** marshals value(s) occupied in the current GTDL type object to the data stream of the standard format specified by the GTSI system. Similarly, the **decode** unmarshals the data stream of the standard format to the current GTDL type object. The details are discussed further in Chapter 5.

- Special routines: **_assign** and **_copy**.

  The **_copy** produces an exact replication of the current GTDL type object. The **_assign** is a virtual assignment function, which is used when an **_Any** object is assigned to another GTDL type object. Both routines are specifically designed to work correctly with the **_Any** type. More details are discussed in Section 4.5.

In addition to the above members, each GTDL type contains an additional special member, called **_link**.

- The **_link** is a special method which inserts the current object into an enclosing constructed type such as **_Struct** or **_Union**. It is invoked when the current type object is an element of a constructed type.

4.3 Basic Types

Basic types are the primitive building blocks in the GTDL system, and all types are ultimately constructed out of the basic types. They are generally used to describe a single, simple entity such as a number or a character. Table 4.1 lists all GTDL basic types and definitions.

Table 4.1 GTDL Type Class Definitions

GTDL type class	Definitions
_Null	No-intrinsic-value, which serves as both type and value. The interpretation of the value is system-dependent.
_Boolean	Logical information with a value, either true (1) or false (0).
_Integer	Numerical information, where all the numbers involved are whole numbers. Different kinds of integers are defined in the GTDL: **_Short**, **_Long**, **_Int**, **_UShort**, **_ULong**, and **_UInt**, which are the corresponding C/C++ built-in type representations of **short**, **long**, **int**, **unsigned short**, **unsigned long**, **unsigned int**, respectively.
_Real	Floating point type, which represents real numbers. Two different kinds of reals are defined in the GTDL: **_Float** and **_Long**, which are the corresponding C/C++ built-in type representations of **float** and **long**, respectively.
_Octet	8-bit type. Two different kinds are defined in the GTDL: **_Char** and **_UChar**, which are the corresponding C/C++ built-in type representations of **char** and **_octet** (**unsigned char**), respectively.
_String	Ordered sequence of characters, which can be used to build textual information.
_BString	**_String** with the restriction of a maximum length.
_Bits	Ordered sequence of bits, which can be used to model binary digits.
_Bytes	Ordered sequence of transparent 8-bit quantities.
_ObjRef	Identification of information objects.

In the GTDL system, the basic types are described in terms of the C++ classes, and used along with one or more type constructors in order to provide more complex constructed types such as arrays and unions. There are definitions for the following basic types: a logical type (**_Boolean**), an integer type (**_Integer**), a real type (**_Real**), an 8-bit type (**_Octet**), character string types (**_String**, **_BString**), transparent data types (**_Bits**, **_Bytes**), an object reference type (**_ObjRef**), and a no-intrinsic-value type (**_Null**).

```
template<class T, ITCKIND K>
class _Integer : public _Type {
protected:
        T _val;
public:
      typedef _Integer<T, K>  base_c;
      typedef T               base_t;

      _Integer();
      _Integer(const base_t v);
      _Integer(const base_c& v);
      ...
};
```

Figure 4.3 _Integer template class

```
typedef      _Integer<int,     ITC::itc_int>          _Int;
typedef      _Integer<short,   ITC::itc_short>        _Short;
typedef      _Integer<long,    ITC::itc_long>         _Long;
typedef      _Integer<_uint,   ITC::itc_uint>         _UInt;
typedef      _Integer<_ushort, ITC::itc_ushort>       _UShort;
typedef      _Integer<_ulong,  ITC::itc_ulong>        _ULong;
```

Figure 4.4 _Integer Type Class Specialization

Table 4.2 GTDL Basic Types and C++ Value Representation

Base Class : GTDL Basic Types	Base Type : C++ built-in Types
_Null	application specific
_Boolean	_boolean (unsigned char)
_Int	int
_Short	short
_Long	long
_UInt	unsigned int
_UShort	unsigned short
_ULong	unsigned long
_Float	float
_Double	double
_Char	char
_UChar	_octet (unsigned char)
_String	Gnu String class
_BString	Gnu String class (_String)
_Bits	Gnu BitString class
_Bytes	Gnu String class
_ObjRef	Gnu String class

4.3.1 Definition and Specialization of Basic Types

In the GTDL system, **_Integer**, **_Real**, and **_Octet** are implemented as template classes to separate the C++ built-in type representation for the value from the GTDL class definition. For example, Figure 4.3 shows the definition of **_Integer** template class. The **_Integer** parameters can be used to specialize **_Integer** with any of the different integer types available in C++. The template parameter **T** denotes the C++ built-in data type, and the template parameter **K** is the internal type code used for identifying the specialized **_Integer** class within the OODSF. For instance, Figure 4.4 shows the definitions of the

specialized **_Integer** classes with various kinds of C++ integer types. Similarly, **_Float** and **_Double** are obtained from the **_Real** template, and **_Char** and **_UChar** from the **_Octet** template. Table 4.2 lists all basic types of the GTDL (Base Class) and the corresponding C++ built-in type (Base Type). The value that a GTDL type can represent is of either a C++ built-in type such as **short** or other C++ class such as **Gnu String** class as shown in Table 4.2

4.3.2 Basic Types and Value Representations

To distinguish the type reference to a GTDL data type from its internal type representation of the value, the GTDL data type is referred to as the base class (**base_c**), and the type of the value, which the GTDL data type can hold, is referred to as the base type (**base_t**). Figure 4.3 and 4.4 show the definition of the **_Integer** type. The **_Short** type in the GTDL is referred to **base_c** and the value that **_Short** can hold during runtime, via the **short** type in C++, is referred to as **base_t.** The scope of these type references, **base_c** and **base_t**, is restricted to within the class definition. For example, the references to **base_c** and **base_t** within the **_Short** class definition refer to **_Short** and **short**. Similarly, **base_c** and **base_t** within the **_Char** class definition refer to **_Char** and **char**.

In providing the storage and access methods for the value that a basic type can hold, two approaches are used in the GTDL system.

- A GTDL basic type explicitly declares the value holding member of **base_t** and provides several operators and functions to access it.

- A GTDL basic type inherits from the predefined C++ class and defines it as **base_t**. Consequently, the value holding member is embedded within **base_t**, and all access methods are inherited also.

The first approach is used when C++ explicitly provides **base_t** as a built-in type, such as **short, long**, or **char**, etc. In this case, the GTDL basic type explicitly provides the methods to access and manipulate the value holding member. Those methods are defined within the class definition when the operations are for the primitive purposes such as setting and getting the value. When the operations are for additional purposes such as the arithmetic operations with respect to the value holding member, the GTDL basic type declares a friend class, which actually provides all operations to do the arithmetic.

Not all types defined in the GTDL basic types have the corresponding C++ built-in types. For example, a string type which is actually an array of characters is not defined as a C++ built-in type, but rather provided as a class library such as the **Gnu String** class with various accessing methods. The GTDL basic type in this case inherits from the predefined C++ classes instead of defining the value holding members and all accessing methods in the type definition. This allows reuse of value representations and accessing mechanisms embedded in the existing classes. The GTDL basic data types which take this approach include **_String**, **_Bits**, and **Bytes**. They inherit from **Gnu String**, **Gnu Bit String**, and **Gnu String** class, respectively. **_BString** is a bounded string, which inherits from the **String** class but adds a maximum length restriction. It is implemented as a template with a template parameter for the length restriction. Essentially, it also inherits from the **Gnu String** class since it inherits from **_String**.

```
class _Boolean : public _Type {
private:
        _boolean   _val;
public:
        typedef _Boolean   base_c;
        typedef _boolean   base_t;

        _Boolean();
        _Boolean(const base_t& v);
        _Boolean(const base_c& v);

        inline void _link(_TypeList* list);

        void operator = (const base_t& v);
        void operator = (const base_c& v);

        virtual void _assign(_Type& v);

        virtual int encode(GTSI& pe);
        virtual int decode(GTSI& pe);

        operator base_t() const;
        _boolean value() const;

        virtual _Type* _copy() const;

       ...
};
```

Figure 4.5 _Boolean Type

4.3.3 Type Conversion

The implicit and explicit type conversion routines from **base_c** to **base_t** are provided in each GTDL basic data type definition. The explicit type conversion method is called **value**. The implicit type conversion provides a seamless environment for using a GTDL basic data type like a normal C++ built-in data type. For example, the variables declared as **_Short** can be used like a C++ **short** variable:

```
int          i = 3;
_Short       short_1 = 0, short_2;

short_2 = i + short_1;
```

Figure 4.5 is the complete definition of **_Boolean** type which illustrates all the details including the special members and type conversion members.

4.4 Constructed Types

The GTDL provides various kinds of type constructors in order to build more complex types. Table 4.3 shows the list of available constructed types and their definitions.

Table 4.3 Constructed Types and Definitions

Constructed Types	Definition
_Array	A fixed-length array of a homogeneous type
_Sequence	A variable-length array of a homogeneous type. The length is available at runtime.
_Struct	A record type, consisting of an ordered set of heterogeneous types.
_Union	A discriminated union type, consisting of an ordered set of heterogeneous types. Only an instance of an element, indicated by the discriminator value, is valid.

A constructed type is defined in terms of other types each of which may be simple or constructed. This nesting can proceed to an arbitrary depth as necessary for the

application. All constructed types are ultimately defined in terms of basic types. The available constructed types in the GTDL system include the type constructors for homogeneous entities such as **_Array** and that for heterogeneous entities such as **_Struct**.

4.4.1 Array Type

The GTDL Array type is a fixed-length vector or multidimensional matrix of homogeneous entities. Those homogenous entities may be either basic or constructed. Generally, a concrete array type is defined by two attributes. First, the type of the homogeneous entities should be specified. Secondly, the length of the array, which is either static or dynamic, should be specified. In the case of static length, the length of the array is known before runtime so that the required size of memory can be allocated accordingly when an array object is declared. When the length of an array is dynamic, some dynamic memory management should be incorporated since the length varies during runtime. The GTDL **_Array** is an array type with a fixed length and is discussed in this section. The GTDL **_Sequence**, which is an array type with dynamically varying lengths, is discussed in the next section.

4.4.1.1 Implementation

The **_Array** type is implemented as a C++ template class, where the type of elements and the maximum length are specified as template parameters. Figure 4.6 illustrates the **_Array** type definition.

```
template<class T, int B>
class _Array : public _genArray {
private:
        int     _size;
        T       buffer[B];

public:
        typedef _Array<T,B>     base_c;
        typedef _Array<T,B>     base_t;

        _Array();
        _Array(const base_c& v);

        inline void _link(_TypeList* list);

        void operator=(const base_c &v);

        virtual void _assign(_Type& v);

        const T& operator[] (int index) const;
        T& operator[] (int index);

        virtual int length() const;

        // encode/decode methods are inherited.

        virtual _Type* _copy() const;
};
```

Figure 4.6 _Array Type

The definition of **_Array** provides two template parameters. **T** is for the type of elements, and **B** is for the length of elements. A fixed-length array variable can be declared by specifying the element type and the size along with **_Array**. For example, the following segments illustrate the declaration of two array variables: one for the array of the **_Char** type with a length of 3 and one for the array of the **_Short** type with a length of 2.

```
_Array<_Char, 3>  array_1;
_Array<_Short, 2> array_2;
```

Multidimensional arrays can be defined in terms of nested arrays in the GTDL system. A variable declaration for a 2-dimensional array of characters with a length of 3 and a length of 2 can be expressed in C/C++ as

```
char  char_2D[3][2];                    // C/C++ 2-D array
```

The same variable can be expressed in the GTDL system by

```
_Array< _Array<_Char, 2>, 3>  char_2D;    // GTDL 2-D array
```

4.4.1.2 Interface

In a typical array type, an indexing mechanism among array elements, via the selection operation, is provided by using special symbols such as brackets ([,]) that are associated with the array name and indexing value. Symbolically, this notation can be shown as

```
array_name[index_value] --> element
```

This normal interface for accessing an array element is also employed in the **_Array** type to provide a familiar programming environment. Two overloaded operators (**operator[]**) in Figure 4.6 are the functions designed to select appropriate array elements and to return references to the individual elements. These operators allow an **_Array** variable to be used much like a C/C++ array variable as shown below.

```
_Array<_Short, 3> short_array;
short_array[1] = 11;
short_array[2] =  4 + short_array[1];
```

In addition to the indexing members, another interfacing member called **length** is provided to return the length of the current **_Array** type object.

4.4.1.3 Memory Management

The **_Array** type class has a private member called **buffer** which is a normal C++ **array** type as shown in Figure 4.6. When an **_Array** type is declared, the **_Array** constructor is invoked and this declares **buffer**. The size of required memory, **B***sizeof (**T**), is allocated when this **buffer** is declared. When the lifetime of an **_Array** variable is over, a destructor is called that frees the allocated memory to the **buffer**. Note that an **_Array** type does not provide an explicit destructor, but will use the default destructor implicitly.

4.4.2 Sequence Type

_Sequence type is a variable length vector of homogeneous entities. Internally, a **_Sequence** uses a dynamic array in which the binding of subscript ranges and storage allocation can change any number of times during the array's lifetime [SEB93]. The **_Sequence** type has the maximal flexibility of growing and shrinking arbitrarily as the need for space changes during runtime so that the length is the number of elements occupied at a particular moment during the program execution. This dynamic characteristic of the **_Sequence** imposes a possible performance penalty due to the runtime storage

allocations and deallocations, compared to the **_Array** type. However, the overhead performance enhances the flexibility of the **_Sequence**.

4.4.2.1 Implementation

In comparison with an **_Array**, the **_Sequence** type also specifies the type of elements and the maximum length even though it is a little different from the **_Array** type in that the maximum length can be either bounded or unbounded. In addition, the current number of elements in a **_Sequence** object is traced during runtime because it changes dynamically.

Figure 4.7 illustrates the definition of the **_Sequence** type. Similar to **_Array**, the type of elements and the maximum length are incorporated into the **_Sequence** type with the template parameters since the concept of binding between the two attributes and an array type during the variable declaration still can be used. The runtime length information is embedded within the definition of **_Sequence** since it varies during runtime.

The **_Sequence** type can be either bounded or unbounded. In the case of an unbounded **_Sequence** type, a predefined constant (**UNBOUNDED**) can be used as the specification of a maximum length so that the behavior of the **_Sequence** type can be incorporated accordingly without a maximum length restriction as shown below.

```
_Sequence<_Char, 3>            seq_1; // maximum length bounded
_Sequence<_Char, UNBOUNDED>    seq_2; // maximum length unbounded
```

```
template<class T, int B>
class _Sequence : public _genSequence {
...
public:
        typedef _Sequence<T,B>        base_c;
        typedef _Sequence<T,B>        base_t;

        _Sequence();
        _Sequence(const base_t& v);
        ~_Sequence();

        inline void _link(_TypeList* list);

        void operator=(const base_t& v);
        virtual void _assign(_Type& v);

        int insert(int index, const T::base_t& v)
        int insert(int index, T& v);

        int append(const T::base_t& v);
        int append(const T& v);

        inline void flush();
        void remove(int index);

      // int length() is inherited.

        const T& operator[] (int i) const;
        T& operator[] (int i);
```

Figure 4.7 _Sequence Type

The fundamental representation of the **_Sequence** type to describe the dynamic length array is a linked list data structure as shown in Figure 4.8. By inheriting from the linked list management class, **_TypeList**, the **_Sequence** includes the services necessary for the linked list manipulations (**_genSequence** inherits from **_TypeList**; the arrow from **_TypeList** to **_Sequence** reflects the inheritance relations in Figure 4.8.) The square objects linked by the arrows from **_TypeList** are the **_TypeListNode** objects,

which actually contain the pointer to the element objects shown as circles. The linkages are established and managed by the **_TypeListNode** objects.

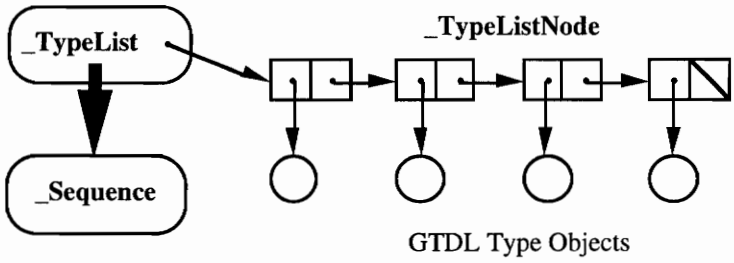

Figure 4.8 _Sequence: List Data Structure

4.4.2.2 Interface

The appropriate interfaces for accessing and manipulating elements should be defined properly. Using the list-accessing routines defined in **_TypeList**, the **_Sequence** redefines several accessing and managing methods for the linked list for the dynamic length array. The defined list management methods are:

- **insert**

  Inserts a given element at the specified index in the linked list. It is implemented by **_Typelist::insert**.

- **append**

  Appends a given element at the end of the linked list. It is implemented by **_TypeList::append.**

- **remove**

  Deletes the element at the specified index from the linked list. It is implemented by **_TypeList::remove**.

- **length**

  Returns the current number of elements in the linked list. It is inherited from **_TypeList**.

- Indexing operator, **[ ]**

  Returns the reference to the element at the specified index so that a variable declared as the **_Sequence** type can use a normal array-like indexing interface. It is implemented by **_TypeList::operator[ ]**.

The following code illustrates the use of some method invocations described previously.

```
_Sequence<_Char, 3>     seq_1; // maximum length bounded

seq_1.append('a');
seq_1.append('c');
seq_1.insert(2, 'b');          // seq_1 : a - b - c
```

4.4.2.3 Memory Management

Explicit dynamic management of memory due to allocations and deallocations occurs in two levels: a whole list level and an element level. The element-level allocations/deallocations occur when the interface routines that add and remove elements are called. For example, if **append** is called, the memory necessary for a new element and a

new **_TypeListNode** object for holding the pointer to the created element are allocated as shown below.

```
int append(const T::base_t& v) {
      if (_max_leng != UNBOUNDED && length() >= _max_leng)
            return 0;

      return _TypeList::append(new _TypeListNode<T>(new T(v)));
}
```

The **append** first checks the boundary. Then, it does the memory allocations for the element of type, **T**, and the **_TypeListNode** object, which is then appended to the type list.

The list-level allocations/deallocations occur in the following instances.

- When a **_Sequence** type object with a length bigger than 0 is passed as a parameter for the constructor invocation, the memory allocation occurs at a whole list level to store and link all values occupied by the elements of the passed **_Sequence** type object.
- When a **_Sequence** type object with the length bigger than 0 is assigned, the memory allocation occurs at a whole list level in a similar way to the first case.
- If the destructor is invoked when the lifetime of the current **_Sequence** type object is over, the memory occupied by elements and **_TypeListNode** objects is deallocated.

The list-level allocations/deallocations are actually not more than the repetitions of the element-level allocations/deallocations in the **_Sequence** type. For example, if a **_Sequence** type object which contains N elements is assigned, the list-level allocation is

actually N element-level allocations since it will be the repetition of N element and **_TypeListNode** object allocations as shown below.

```
void operator=(const base_t& v) {
        ...
        int len = v.length();
        for (int i=0; i < len; i++) {    // append
            this->append(v.operator[](i));
        }
}
```

4.4.3 Struct Type

A **_Struct** type is a record type which consists of an ordered set of heterogeneous entities. Compare to an array type, it is fundamentally different in that heterogeneity of elements is preserved so that a **_Struct** type can have elements of different GTDL types. This also introduce different accessing mechanisms. The primitive method of accessing the elements in an array type is through referencing by indices. However, in the case of a record type, the basic referencing mechanism for elements is through the name of each element. The **_Struct** type manages the heterogeneity of its members and provides the referencing mechanisms by implementing a list-oriented data structure.

4.4.3.1 Implementation

The **_Struct** type is implemented as a C++ class which inherits from the **_TypeList** as shown in Figure 4.9. This implies that it inherits all services necessary for

manipulating a linked list so that the **_Struct** will behave as a list-oriented data structure in terms of managing its elements.

```
class _Struct : public _Type, public _TypeList {
public:
        typedef _Struct  base_c;
        typedef _Struct  base_t;

        _Struct();
        _Struct(const base_c &v);

        inline void _link(_TypeList* list);

        void operator=(const base_c &v);
        virtual void _assign(_Type& v);

        virtual int encode(GTSI& pe);
        virtual int decode(GTSI& pe);

        virtual _Type* _copy()  const;

};
```

Figure 4.9 **_Struct** Type

Figure 4.10 is an example of declaring a user-defined record type in C++. Figure 4.11 is an example of declaring the equivalent record type using the GTDL **_Struct** type.

```
struct class_info {                          // C++ struct type
        long            index;
        long            coursenum;
        String          dept;
        String          coursename;
};
```

Figure 4.10 An example: C++ **struct**

```
class class_info : public _Struct {  // GTDL _Struct type
public:
        _Long     index;
        _Long     coursenum;
        _String   dept;
        _String   coursename;

        class_info() {  _LINK(index); _LINK(coursenum);
                        _LINK(dept);  _LINK(coursename);  }

        void operator=(class_info& v) { _Struct::operator=(v);
}
};
```

Figure 4.11 An example: GTDL **_Struct**

Each member of C++ built-in types is described in terms of the corresponding GTDL types. This implies that each member behaves as a GTDL type object while it is a class member and belongs to the container class, **class_info**. As the constructor is invoked, **_LINK**'s are called for all elements as shown in the definition of the constructor. Note that **_LINK(index)** is the macro defined as **index._link(this)**, where the object, **index**, is appended into the list data structure pointed by **this**. Since **this** points to the current type object, **class_info**, the invocations of all **_LINK**'s in the constructor result in the creation of the linked list data structure where all members are linked to the current **class_info** type object. Figure 4.12 exhibits the conceptual description of this linked list data structure.

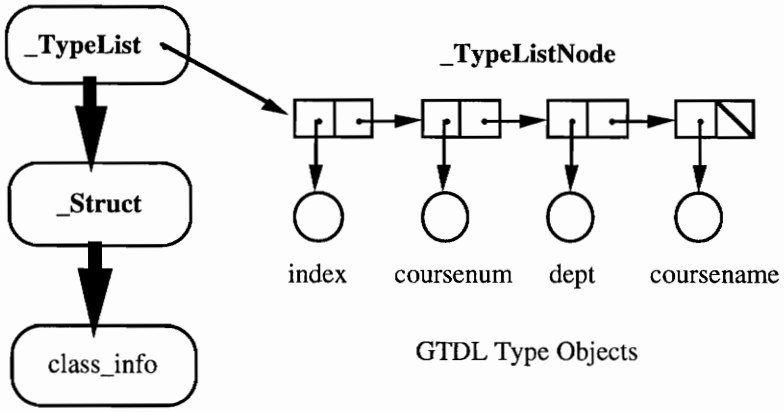

Figure 4.12 _Struct: List Data Structure

4.4.3.2 Interface

In the **_Struct** type, there are two methods provided for accessing a member. First, a member can be referenced by the name, using a normal C/C++ class member accessing syntax. For example, the **dept** member can be accessed by the **class_info.dept** notation which is the same as a C/C++ **struct** member accessing mechanism. This preserves knowledge of a typical member accessing scheme for a record type. Secondly, the implementation of the list-oriented data structure for **_Struct** also provides an alternative member accessing mechanism, via an array-like indexing. For example, the accessing notation for **dept** can be expressed as **class_info[3]**. However, it is different from the first case in that it returns the pointer to the generic types class **_Type**, to which the actual member object is bound. Accordingly, the proper method invocations for the elements are by means of virtual methods as in the polymorphic types. The array-like indexing is preserved for the GTSI system for encoding and decoding purpose only.

4.4.3.3 Memory Management

The memory allocations and deallocations for a **_Struct** object and its members are achieved by normal C++ constructor and destructor invocations since each member is statically declared as a member in the **_Struct** class definition. The **_TypeListNode**'s in Figure 4.12 are dynamically created for establishing a linkage when the **_LINK**'s are called as shown in Figure 4.11. Consequently, the **_Struct** destructor explicitly deallocates all created **_TypeListNode**'s.

4.4.4 Union Type

A union is a type that stores a single value of different type at different times during program execution. Particularly, a discriminated union is one that is associated with an additional value, a tag or discriminant, which identifies the current type of the value stored in the union [SEB93]. The **_Union** type is an implementation of the discriminated union. It consists of a discriminant and an ordered set of heterogeneous GTDL types among which only an instance of them, as described by the discriminating value, is bound to the **_Union** type object.

4.4.4.1 Implementation

The **_Union** type is described in terms of a C++ class similar to the **_Struct** type as shown in Figure 4.13.

```
class _Union : public _Type, public _TypeList {
private:
        int     _discriminator;
public:
        typedef _Union      base_c;
        typedef _Union      base_t;

        _Union();
        _Union(const base_c& v);

        inline void _link(_TypeList* list);

        void operator=(const base_c& v);
        virtual void _assign(_Type& v);

        virtual int encode(GTSI& pe);
        virtual int decode(GTSI& pe);

        _Type* set_type(int t);

        inline int get_type() const;

        virtual _Type* _copy() const;
};
```

Figure 4.13 _Union Type

The **_Union** type inherits from **_TypeList** so that it includes a list-oriented data structure similar to the **_Struct** type. However, the **_Union** manages its members differently. Compared to the lifetime of the **_Struct** type objects which is the same as its members, **_Union** members have different lifetimes. For example, if a member of a **_Union** is referenced, only the referenced member is bound to the **_Union** type object and the previously bound member is discarded since only one member can be bound at a time.

Generally, a **union** type such as in C/C++ is implemented by allocating an enough memory to hold any single instance of its members while permitting only one member to be

bound at a particular moment of time. However, in the case of the **_Union** type, this behavior is reflected in terms of the dynamic creation and deletion of member objects.

```
template<int N, class T> class _CASE : public _genCASE {
private:
        _Union      *choices;
        T*          tbound;
public:
        typedef _CASE<N,T>      base_c;
        typedef T               base_t;

        _CASE();
        _CASE(_Union* list);

        inline void _link(_Union* list);

        ~_CASE();

        void operator = (base_c& v);
        void operator = (const base_t& v);

        virtual void _assign(_Type& v);

        virtual int encode(GTSI& pe);
        virtual int decode(GTSI& pe);

        _Type* create();
        void destroy();

        operator T&() const;
        T& _getobj() const;

       virtual _Type* _copy() const;
};
```

Figure 4.14 _CASE Class

Figure 4.14 shows the definition of the **_CASE** class. An instance of the **_CASE** serves as the dedicated memory managing object of a particular member while cooperating with the **_Union** type object. The **_CASE** is defined as a C++ template class with two template parameters: a discriminant id (**N**) and the type of element (**T**). The **_CASE** class

provides two methods: **create**, which creates an instance of a **_Union** member, and **destroy**, which frees a **_Union** member. These two routines are called by the **_Union** type object when a different **_Union** member is referenced. For example, when a member is referenced, the **_Union** type object requests the current element managing **_CASE** object to deallocate the element by calling **destroy** and requests the **_CASE** object for the referenced element to create the element by calling **create**.

A user defined **_Union** type can be expressed similarly to the **_Struct** type except that **_CASE**s are specified along with the members as shown in Figure 4.15.

```
class mixed : public _Union {
public:
      _CASE<1, _Short>  sval;
      _CASE<2, _Float>  fval;
      _CASE<3, _Char>   cval;

      mixed() { _LINK(sval); _LINK(fval); _LINK(cval); }

      void operator=(mixed& v)       { _Union::operator=(v); }
};
```

Figure 4.15 An example: GTDL **_Union**

As the constructor is called, all **_CASE** objects are linked to the **mixed** object so that the **mixed** object builds a list-oriented data structure as shown in Figure 4.16. Note that currently **fval** is bound to the **mixed** object.

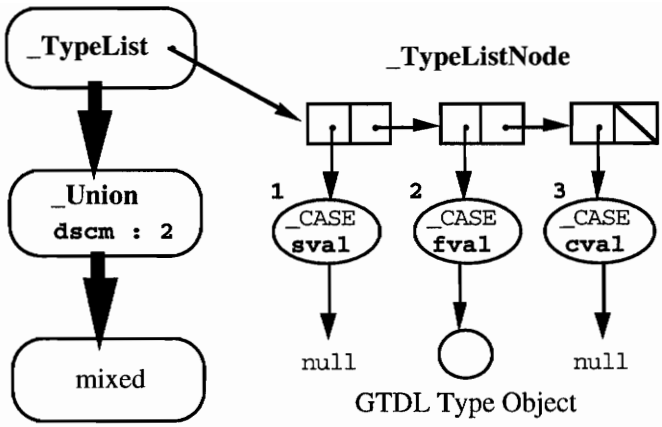

Figure 4.16 _Union: List Data Structure

4.4.4.2 Interface

To access a member such as **fval**, the combination of a normal C/C++ struct member accessing syntax and a special notation, **CASE**, is used as illustrated in the following line.

```
mixed.CASE(fval) = 3.5;
```

The **CASE** is the macro definition of a **_CASE** class member, **_getobj()**, described in Figure 4.14. The **_CASE** object for the **fval** requests the **_Union** to free the element currently bound and to set the appropriate discriminant value for the **fval**. Then, it creates an instance of **fval** and assigns 3.5 to it.

As an alternative member referencing syntax similar to the **_Struct** type, the **_Union** type also provides an array-like member accessing notation, utilizing its list-

oriented data structure. However, its intended use is restricted to the GTSI system for encoding and decoding.

4.4.4.3 Memory Management

The memory allocations and deallocations for the **_Union** object and **_CASE** objects are done by normal C++ constructor and destructor invocations since each **_CASE** object is statically declared as a member in the **_Union** class definition. The **_TypeListNode**s in Figure 4.16 are dynamically created to link all **_CASE** objects when the **_LINK**s are called in the constructor as shown in Figure 4.15. The **_Union** destructor explicitly deallocates all created **_TypeListNode**s. Actual member objects are managed dynamically by the **_CASE** objects as described in the previous sections.

4.5 Any Type

An **any** type can store a value of any type during runtime. From a programming language perspective, the type is not specified by a declaration statement but it is bound to the type of a value assigned to it [SEB93]. The data specifications in ASN.1 [ISO87a] and IDL [OMG 91] provide the **any** type to describe a "hole" in the specification, which can not be determined at specification time. The GTDL type system also employs an **any** type, called **_Any**, to describe the data with an arbitrary GTDL type, which is determined dynamically.

4.5.1 Special Operators

Implementing the **any** type, which includes a dynamic type binding feature, in a statically typed environment is somewhat unnatural. However, the polymorphism associated with dynamic binding and inheritance and the object based descriptions of data types in the GTDL system alleviate the difficulty of the design and implementation of an **any** type. The **_Any** type is defined as a C++ class as illustrated in Figure 4.17.

```
class _Any : public _Type {
private:
        _Type   *_any;
public:
        typedef _Any    base_c;
        typedef _Type   base_t;

        _Any();
        _Any(const base_t& v);
        _Any(const base_c& v);
        ~_Any();

        inline void _link(_TypeList* list);

        _Type*  any();

        void operator=(const base_t& v);
        void operator=(const base_c& v);

        void operator<<=(const base_t& v);
        void operator<<=(const base_c& v);

        virtual void _assign(_Type& v);

        void operator>>=(base_t &v);
        void operator>>=(base_c &v);

        virtual int encode(GTSI& pe);
        virtual int decode(GTSI& pe);

};
```

Figure 4.17 _Any Type

The **_Any** type has a private member, called **_any**, which is declared as the pointer to the generic **_Type** class so that any GTDL type object can be bound during runtime while the polymorphism is utilized properly. It also provides two special assignment operators, **<<=** and **>>=** [HPM94]. The **<<=** operator assigns an arbitrary GTDL type object to the **_Any** (widening). The **>>=** operator assigns an **_Any** type object to the specified type object (narrowing).

```
_Any  any_val;
_Char char_1 = 'c', char_2;

any_val <<= char_1;            // widen
any_val >>= char_2;            // narrow
```

In this example, **char_1**, which is declared as **_Char** type, is assigned to the **any_val** by using the **<<=** operator. Then, the **any_val**, which actually holds a **_Char** object is copied back to the **char_2** by using the **>>=** operator.

4.5.2 _copy and _assign

Two special assignment operators described in the section are highly integrated with the **_copy** and **_assign** methods defined in all GTDL data types. For example, the following segments of codes show the definition of the **<<=** and **>>=** operators for the **_Any**.

```
void operator<<=(const base_t& v) {
      ...
      if (_any) { delete _any; _any = 0; }
      _any = v._copy();
}

void operator>>=(base_t &v)     { v._assign(*_any); }
```

When the <<= operator in the example (Section 4.5.1) is invoked, the previously bound object to **any_val._any** is deallocated. Then, **char_1._copy()** which produces an exact replication of the **char_1** is called so that it can be saved under **any_val._any**. The >>= operator calls a virtual assignment operator, **char_2._assign(*any_val._any)** that assigns the object bound to **any_val._any** to the **char_2**.

4.6 ITC, LTC, and ROD

In addition to the data type definitions, the GTDL provides three special classes: **ITC** (Internal Type Code), **LTC** (Language-Dependent Type Code), and **ROD** (Required/Optional/Default).

4.6.1 ITC

The **ITC** is type code information embedded within the GTDL system. Each type defined in the GTDL includes a unique non-negative number (a value in an enumerated type) as illustrated in Table 4.4. For a given type object, its type in the GTDL system can be identified by its **ITC** type code.

Whenever a constructor for a GTDL type is invoked, its unique id is assigned to the **ITC** embedded within the object. For example, the definitions of the **_Array** and **_Struct** type constructors are illustrated below.

```
_Array::_Array()          { ITC::_itckind = ITC::itc_array;
                            ... }

_Struct::_Struct()        { ITC::_itckind = ITC::itc_struct;
                            ... }
```

Table 4.4 ITC kinds

GTDL types	ITCKIND
_Null	itc_null
_Boolean	itc_boolean
_Int	itc_int
_Short	itc_short
_Long	itc_long
_UInt	itc_uint
_UShort	itc_ushort
_ULong	itc_ulong
_Float	itc_float
_Double	itc_double
_Char	itc_char
_UChar	itc_uchar
_String	itc_string
_BString	itc_bstring
_Bits	itc_bits
_Bytes	itc_bytes
_ObjRef	itc_objref
_Array	itc_array
_Sequence	itc_sequence
_Struct	itc_struct
_Union	itc_union
_Any	itc_any

4.6.2 LTC

The **LTC** is a language-defined type code. No information for the **LTC** is defined during the predefinition phase but it is deferred to the postdefinition phase when the mapping to a given specification language is defined. For example, ASN.1 tag information can be stored in the **LTC** so that it can be used to identify a unique id relevant to an ASN.1 data type. This is discussed further in Chapter 6.

4.6.3 ROD

The **ROD** denotes an option among required, optional, or default for an element in a constructed type which is defined in the postdefinition phase. It is mainly designed for the ASN.1 language definition and will be discussed further in Chapter 6.

4.7 LANGDEF

The **LANGDEF** is a C++ class from which all GTDL type classes inherit. The primary objective of this class is to let the GTDL system embed knowledge for the type of specification language defined on the top of the OODSF in the postdefinition phase. Figure 4.18 shows the definition of the **LANGDEF** class.

```
class LANGDEF : public ITC, public _ROD, public LTC {
protected:
        LANGMODE _lang;
        AsnTag   _tag;
public:
        LANGDEF() : _lang(__GTDL) {}

        void     lang(LANGMODE L)     { _lang = L; }
        LANGMODE lang()               { return _lang; }

        virtual AsnTag& get_AsnTag();
};
```

Figure 4.18 LANGDEF Class

It provides a constructor which specifies the current language mode; GTDL is the default. This language mode can be redefined in the postdefinition by calling

```
LANGDEF::lang(__ASN1);
```

The language mode information is primarily used for creating an ASN.1 tag because the GTSI system requests the ASN.1 tag when it encodes and decodes with BER. The method **get_AsnTag** returns an ASN.1 tag. It simply creates an ASN.1 tag using the **LTC** if the language mode is __ASN1 because the **LTC** actually contains the ASN.1 tagging information. Other than that, it does a default conversion to create an ASN.1 tag from the **ITC** as illustrated in Table 4.5.

Table 4.5 ITC to ASN.1 Tag Conversion

ITC	ASN.1 Tag Class	ASN.1 Tag Id
itc_null	UNIVERSAL	asn_null (5)
itc_short itc_long itc_int itc_ushort itc_ulong itc_uint	UNIVERSAL	asn_int (2)
itc_float itc_double	UNIVERSAL	asn_real (9)
itc_boolean	UNIVERSAL	asn_bool (1)
itc_string itc_bstring	UNIVERSAL	asn_ia5str (22)
itc_octet itc_bytes	UNIVERSAL	asn_octetstr (4)
itc_bits	UNIVERSAL	asn_bitstr (3)
itc_sequence itc_array itc_struct	UNIVERSAL	asn_seq (16)
itc_objef	UNIVERSAL	asn_objid (6)
itc_union itc_any	N/A	N/A

Chapter 5

Predefinition: General Transfer Syntax Interface and Implementations

5.1 Overview of GTSI class hierarchy

The General Transfer Syntax Interface (GTSI) provides an interface for encoding services to the General Type Definition Library (GTDL) so that a type object of GTDL can invoke an appropriate encoding or decoding routine. Figure 5.1 shows the overall class hierarchy for the GTSI.

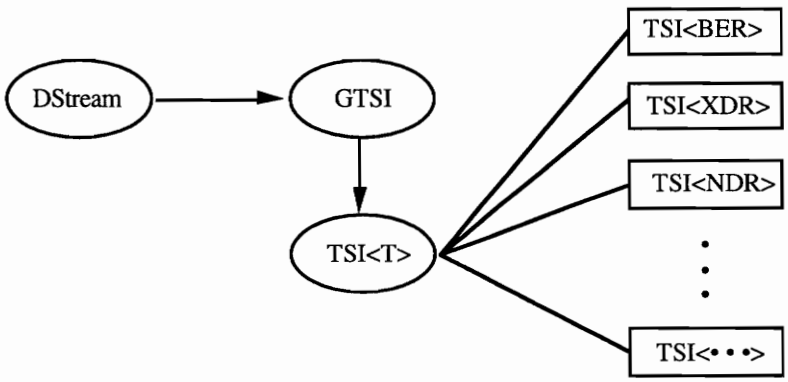

Figure 5.1 GTSI Class Hierarchy

The arrows indicate the inheritance relations in that the GTSI inherits from **DStream** and the template class **TSI<T>** inherits from the GTSI. An undirected line denotes the instantiation of **TSI<T>** with the specification of a selected encoding rule. For example, **TSI<XDR>** is a transfer syntax interface instantiated with the XDR encoding rules.

5.2 Related Work

The ASN.1 class library [MUL93] included this separated translation service as a PE (Presentation Elements) library, which is specifically designed for ASN.1 types. The OODSF, as a descendent of the architectural design with respect to the separation of representation and translation, also adopts this strategy with the design of the GTSI. However, since the OODSF is a framework to incorporate general data types defined in the GTDL, which will be further used for the definition of IDL and ASN.1, the GTSI provides different and extended interfaces, compared to the PE class defined in the previous ASN.1 class library. The details of the GTSI system are discussed in the following sections.

5.3 Abstract GTSI Class

Figure 5.2. illustrates the part of definition for the GTSI class. The GTSI is an abstract class where all interface routines are declared as pure virtual methods, allowing the binding of implementation to be deferred. For example, if a **TSI<XDR>** object is bound to the GTSI, the GTSI embeds the behavior of the **TSI<XDR>** object so that the

invocation of a GTSI routine results in an XDR encoding service. Similarly, the GTSI may include **TSI<NDR>** object behavior if NDR encoding services are necessary.

```
class GTSI : public DStream {
public:
      ...

      virtual int prim2pe(_Null&) = 0;
      virtual int pe2prim(_Null&) = 0;

      virtual int prim2pe(_Int&) = 0;
      virtual int pe2prim(_Int&) = 0;
      ...

      virtual int array2pe(_genArray&) = 0;
      virtual int pe2array(_genArray&) = 0;

      virtual int struct2pe(_Struct&) = 0;
      virtual int pe2struct(_Struct&) = 0;
      ...
};
```

Figure 5.2 The GTSI class

Figure 5.2 shows only some of the interface routines. In the actual definition of the GTSI class, one-to-one mapping from a set of all type definitions in the GTDL to a set of interface routines in the GTSI occurs. In other words, for each type definition in the GTDL, there is a corresponding interface routine for invoking the proper encoding service. For example, **_Short::encode** calls an interface routine **prim2pe(_Short&)** and **_Long::encode** calls **prim2pe(_Long&)**.

```
int _Short::encode(GTSI& pe)   { pe.prim2pe(*this); }
int _Long::encode(GTSI& pe)    { pe.prim2pe(*this); }
```

5.4 Template TSI<T> Class

Figure 5.3 shows the definition of the TSI class. It is declared as a template class with a template parameter **T**, which denotes an arbitrary encoding rule. For example, the declaration of **TSI<XDR>** is the binding of the TSI class with an XDR encoding rule.

```
template<class T> struct TSI : public GTSI {
      ...

      virtual int prim2pe(_Null&);
      virtual int pe2prim(_Null&);

      virtual int prim2pe(_Int&);
      virtual int pe2prim(_Int&);
      ...

      virtual int array2pe(_genArray&);
      virtual int pe2array(_genArray&);

      virtual int struct2pe(_Struct&);
      virtual int pe2struct(_Struct&);
      ...

};
```

Figure 5.3 The **TSI<T>** class

Similarly, **TSI<NDR>** implies the binding of the TSI class with a NDR encoding rule. Consequently, the declaration of a variable such as

```
TSI<NDR>     pe;
```

creates an object, **pe**, which provides an invocation interface associated with a NDR encoding rule. Furthermore, the GTDL obtains the access to the NDR service through the

virtual methods defined in the GTSI. For example, the following code segments demonstrate an invocation of a NDR service through the GTSI.

```
TSI<NDR>      pe;
_Short        short_1 = 3;

short_1.encode(pe);            // calls GTSI::prim2pe
```

The binding of a **TSI<NDR>** object, **pe**, to the GTSI occurs when the actual parameter, **pe**, is passed to **short_1.encode**, which is declared as the GTSI type. Consequently, the invocation of **GTSI::prim2pe** in the implementation of **short_1.encode** causes the actual method invocation of **TSI<NDR>::prim2pe** so that the NDR encoding rule is used for encoding the value stored in **short_1**.

5.5 DStream: Data Stream

DStream (Data Stream) is a C++ class, which provides a data structure for storing marshaled (encoded) data in a byte stream and also the methods for manipulating them. As all data are marshaled into the **DStream**, the runtime library can actually send the byte stream through the network using, for example, TCP/IP sockets, according to its own communication control structure. The runtime library of the receiving side may store the incoming byte stream in the **DStream**. It can be decoded to the local representation so that it can be used in the local environment, for instance, to construct the proper signature for invoking necessary operations. Figure 5.4 illustrates this.

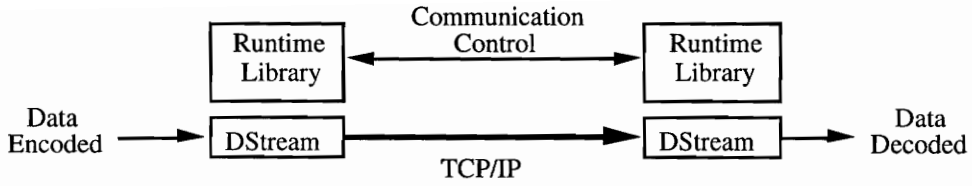

Figure 5.4 The DStream

As shown in Figure 5.1, since the GTSI class inherits from the **DStream**, the GTSI and **TSI<T>** inherit the functionality of a **DStream**. For instance, the **TSI<XDR>** object encodes data values according to the XDR encoding rule and stores them into **DStream**. In the receiving side, the **TSI<XDR>** object decodes the byte stream stored in the **DStream** with the XDR decoding rule.

5.6 TSI<XDR> and TSI<PPBER>

Among many existing encoding rules, the current version the GTSI system includes implementations for **TSI<XDR>** and **TSI<PPBER>**, where the **TSI<XDR>** provides XDR encoding and decoding services and **TSI<PPBER>** emulates BER encoding and produces a human readable form.

XDR and BER show different characteristics. The BER allows a variable length encoding scheme where tags are encoded in addition to values. However, the XDR uses a fixed-length encoding to avoid complexity and inefficiency [ROS90]. The implementation of these two different rules demonstrates the functionality and operational feasibility of the GTSI system.

TSI<XDR>

The **TSI<XDR>** is implemented on top of the Sun XDR encoding/decoding library package. The methods defined in the **TSI<XDR>** conform to the interfaces defined in the GTSI so that appropriate encoding/decoding services are provided to the GTDL system.

TSI<PPBER>

TSI<PPBER> is a pretty-print rule set which does not perform actual encoding but displays a human readable form of the BER encoding. Since BER requires the ASN.1 tag information, **TSI<PPBER>** requests the GTDL to return an ASN.1 tag (**AsnTag**) by calling **get_AsnTag()**. This allows the tag information to be incorporated accordingly when the value is encoded.

The implementation details of each encoding rule is subject to the specification of each rule, so they are not discussed further in this thesis. The functional testing is, however, discussed in Chapter 7.

Chapter 6

Postdefinition: IDL and ASN.1 Class Libraries

This chapter discusses the extensions of the OODSF to define the data types of OMG IDL and ASN.1 class libraries. Section 6.1 addresses the issues related to the definition of IDL and ASN.1, and Section 6.2 explains type aliasing. Sections 6.3 and 6.4 focus on the definitions of the IDL and ASN.1 class libraries using the OODSF.

6.1 Issues

The data types in the GTDL system can describe the semantics of both IDL and ASN.1 data types in C++ as shown in Table 6.1. Semantically, both IDL and ASN.1 data types are roughly subsets of the GTDL. However, there are still several extensions which should be incorporated for defining the data types of both IDL and ASN.1. These are:

- The syntactic differences, specifically, the type names appropriate to IDL and ASN.1, should be incorporated.

- Some mechanisms for describing additional information specific to encoding and decoding in BER, such as the tag information in ASN.1, should be incorporated.

Table 6.1 IDL/ASN.1 Type Aliases

GTDL types	IDL (OMG)	ASN.1 (without tags)
_Null		AsnNull
_Boolean	idl_boolean	AsnBool
_Int		
_Short	idl_short	
_Long	idl_long	AsnInt
_UInt		
_UShort	idl_ushort	
_ULong	idl_ulong	
_Float	idl_float	
_Double	idl_double	AsnReal
_Char	idl_char	
_UChar	idl_octet	
_String		AsnString
_BString	idl_string	
_Bits		AsnBitString
_Bytes		AsnOctetString
_ObjRef	idl_objref	AsnObjId
_Array	idl_array	
_Sequence	idl_sequence	
_Struct	idl_struct	AsnSeq, AsnSet
_Union	idl_union	AsnChoice
_Any	idl_any	AsnAny

6.2 Type Aliases

To redefine the data types in the OODSF, as either IDL or ASN.1, type aliases specific to IDL or ASN.1 type names are provided. The behavior of a type object is sufficiently defined in the predefinition phase (Chapters 4 and 5). Consequently, the simplest type extension using the GTDL is to create a type alias, which is specialized with its own type naming convention. For example, the IDL **short** type is defined as

```
typedef     _Short      idl_short;
```

Similarly, an ASN.1 boolean type can be defined as

```
typedef     _Boolean    AsnBool;
```

In the case that a data type is defined using a template class in the GTDL, typedef is not allowed unless the parameters are specified. For example, an array type requires two template parameters, the type and length of elements, to be specified along with the aliased type name to use typedef, as illustrated below

```
typedef     _Array<_Short, 3> ArrayType;
```

In general, however, these two pieces of information can only be obtained in an application program as a user-defined type rather than one specified in the postdefinition phase. To resolve this problem, a *macro* is used for creating a name alias. For example, to define **idl_array**,

```
#define      idl_array    _Array
```

is used instead so that the IDL specific type name is preserved while binding with template parameter specifications is deferred to an application program such as in the following line:

```
idl_array<idl_short, 3> array_var;
```

Table 6.1 summarizes the aliases of OMG IDL and ASN.1 base types. Note that the aliases for ASN.1 are the base types which do not contain ASN.1 tagging information. The inclusion of the tagging information is discussed in Section 6.4.

6.3 IDL Class Library

C++ representations of the IDL data types can be defined using the OODSF by allowing type aliases. In this regard, the IDL class library is a simple collection of the aliases, which denote the IDL specific names as in Table 6.1. The type definition using the IDL class library is the same as the GTDL. For example, an IDL specification using **struct** and its mapping to C++ using the IDL class library are shown in Figure 6.1 and Figure 6.2, respectively.

Since the IDL class library is defined using the GTDL system, it can access the encoding/decoding services using the GTSI system, allowing a flexible choice of an encoding rule.

```
TSI<XDR>     pe;
idl_boolean flag = idl_TRUE;
idl_short    num = 1;
```

```
flag.encode(pe);          // calls TSI<XDR>::prim2pe(*this)
num.encode(pe);           // calls TSI<XDR>::prim2pe(*this)
```

In the above code segments, **flag.encode** calls the **prim2pe** interface routine for encoding with XDR. Similarly, **num** also is encoded with the XDR encoding rule.

```
struct class_info {
        long                    index;
        string                  coursename;
        string<20>              department;
        long                    student_list[40];
        char                    student_grade[40];
        sequence<long, 20>      student_incomplete;
};
```

Figure 6.1 IDL Specification : **class_info**

```
class class_info : public idl_struct {
public:
        idl_long                        index;
        idl_string<UNBOUNDED>           coursename;
        idl_string<20>                  dept;
        idl_array<long, 40>             student_list;
        idl_array<char, 40>             student_grade;
        idl_sequence<long, 20>          student_incomplete;

        class_info() {      _LINK(index);
                            _LINK(coursename);
                            _LINK(dept);
                            _LINK(student_list);
                            _LINK(student_grade);
                            _LINK(student_incomplete);
        }

        void operator=(class_info& v) { idl_struct::operator=(v);
}
};
```

Figure 6.2 C++ Representation : **class_info**

6.4 ASN.1 Class Library

The fundamental idea of defining the ASN.1 class library using the OODSF is by means of implementing **tagged types** utilizing the template classes and inheritance. The role of the GTDL is to define the base types, from which the tagged types inherit information-bearing capacity and functionality.

This section discusses the basic concepts of the tagged types and the implementation issues for interfacing the OODSF with the previously defined tagged type tuples [KLM94]. Hence, the new ASN.1 class library can be defined on top of the OODSF while it still utilizes the existing tools and definitions such as caty and UNIV.h.

6.4.1 ASN.1 Tag and Tagged Types

In ASN.1, tagging is used to ensure that two types are distinct if and only if their tags are distinct [STE90]. In this regard, every ASN.1 type, including either built-in or defined, has a tag except for **CHOICE** type and **ANY** type where only one possible tag among all possible alternatives is allowed.

Tag information consists of a tag class and a non-negative integer. The tag class is one of: universal, context-specific, application-wide, or private-use. The non-negative integer denotes a unique tag identifier. All of the built-in types have tags in the universal class. The context-specific tags are used for distinguishing among components of sets and sequences. The application-wide tags are used when they need to be common across the

entire application protocol. The private-use tags are for extensions. Douglas Steedman [STE90] presents the details of these 4 different tag classes.

A **tagged type** is a new type definition which is defined in term of other (base) types so that it includes all information-bearing capacity but adds a new tag. For example, a tagged integer type may hold an integer value as similar to a normal integer type but it can have a tag additionally [STE90].

6.4.2 Tagged Type Tuples

The previous work has defined a translator, caty [LON94], which maps a given ASN.1 specification to a corresponding C++ representation. In defining the caty for translation, a form of tagged type tuple is used, utilizing the parameters for a template class [KLM94] such as:

<tag-class, tag-id, base-type>

The base-type denotes a primitive type or constructed type in ASN.1 which does not include the tag information. Both the tag-class and tag-id constitute the tag information. The tag-class is one of **UNIVERSAL**, **APPLICATION**, **PRIVATE**, or **CONTEXT**, which denotes universal, context-specific, application-wide, and private-use tag, respectively. The tag-id is a unique number specified by the ASN.1 specification. For example,

```
<UNIVERSAL, 2, AsnInt>
```

is a tuple for describing the **INTEGER** type in ASN.1 which has a **UNIVERSAL** tag class and a tag id as 2. Consequently, by creating an alias, ASN.1 types are defined as follows.

```
typedef BUILTIN<UNIVERSAL, 2, AsnInt>          INTEGER;
```

Similarly, all available types for ASN.1 are defined in *UNIV.h.* as shown in Figure 6.3.

```
typedef BUILTIN<UNIVERSAL,  1, AsnBool>         BOOLEAN;
typedef BUILTIN<UNIVERSAL,  2, AsnInt>          INTEGER;
typedef BUILTIN<UNIVERSAL,  3, AsnBitString>    BIT_STRING;
typedef BUILTIN<UNIVERSAL,  4, AsnOctetString>  OCTET_STRING;
typedef BUILTIN<UNIVERSAL,  5, AsnNull>         ASN_NULL;
typedef BUILTIN<UNIVERSAL,  6, AsnObjId>        OBJECT_IDENTIFIER;

typedef BUILTIN<UNIVERSAL,  9, AsnReal>         REAL;
typedef BUILTIN<UNIVERSAL, 10, AsnInt>          ENUMERATED;
typedef BUILTIN<UNIVERSAL, 16, AsnSeq>          SEQUENCE;
typedef BUILTIN<UNIVERSAL, 17, AsnSet>          SET;
...
```

Figure 6.3 UNIV.h - ASN.1 Data Type Definitions

6.4.3 Goals and Strategies

As discussed in the previous section, there exists a translator, caty, which was developed for mapping a given ASN.1 specification to C++ using the definitions of tagged type tuples. Since it is powerful enough to express the C++ representations of ASN.1 specifications, it is a reasonable approach to adopt the definitions of the tagged type tuples, utilizing the existing translator. To achieve this, appropriate interfaces should be provided

in order to incorporate the definitions of the tagged type tuples using the OODSF. This includes:

- Definition of appropriate ASN.1 base types, which are used as the template parameters of tagged type tuples. This is discussed in Section 6.4.4.
- The proper definition of template classes, which supports the fundamental idea of tagged type tuples. This is discussed in Section 6.4.5.

6.4.4 ASN.1 Base Type Aliases

ASN.1 base types used in the template parameters of tagged type tuples are either primitive or constructed types, which do not include tag information. Since the functionality and information-bearing capacity of constructs without tagging information are already defined in the GTDL system, ASN.1 base types can be defined as a subset of the GTDL using aliases as Table 6.1.

6.4.5 Template Class For Tagged Type Tuples

The BUILTIN template class is the one which makes the general types of the OODSF embed the specific properties of ASN.1 by the addition of

- tag class
- tag id
- required, optional, and default characteristics.

```
template<TagCl C, TagId I, class T>
class BUILTIN : public T {
public:

        BUILTIN() {
                ...
                set_LTC(I, T::itc_form(), C);
                LANGDEF::lang(__ASN1);
        }
        BUILTIN(const base_t& v) : T(v)      {
                ...
                set_LTC(I, T::itc_form(), C);
                LANGDEF::lang(__ASN1);
        }
        BUILTIN(AsnList* list, ROD rod = REQUIRED) {
                ...
                set_rod(rod);
                _link(list);
                set_LTC(I, T::itc_form(), C);
                LANGDEF::lang(__ASN1);
        }
        BUILTIN(AsnList* list, const base_t& v) {
                ...
                set_rod(DEFAULTS);
                _link(list);
                set_LTC(I, T::itc_form(), C);
                LANGDEF::lang(__ASN1);
        }

        typedef BUILTIN<C, I, T>  ThisType;
        _COPY(ThisType)

        /* encode/decode methods are inherited */
};
```

Figure 6.4 BUILTIN Template Class

Figure 6.4 illustrates the definition of the BUILTIN template class. The functionality of the base-type is gained through the inheritance from the base-type, **T**. Each constructor contains two common method calls. First, the **set_LTC** is used to record the language specific tagging information obtained through the template parameters, such as ASN.1 tag, to the **LTC**. Another common method, **LANGDEF::lang(__ASN)**, is called to let the LANGDEF know that the OODSF is used for the ASN.1 class library. By allowing the OODSF to embed the knowledge of the type of a specification language as

ASN.1, the GTDL system does not use the default ASN.1 tag conversion which is used when no ASN.1 tag information is provided, but it preserves the tag information recorded in the **LTC**.

The last two constructors in Figure 6.4 are the caty specific interfaces defined for compatibility with the caty generated C++ representations of ASN.1. They are invoked if the current type object is an element of a constructed type such as SEQUENCE. If so, it is linked to the linked list data structure of the enclosing constructed type, while specifying an option among required, optional, or default in the constructed type. The option is recorded in the ROD of the OODSF so that it can be used in the GTSI system.

_COPY is a macro defined for producing the exact replica of the current type object, which includes the tag information and option.

6.4.6 Discussion

The ASN.1 class library defined using the tagged type tuples can access the encoding/decoding services by the GTSI. Since the tagged type tuples inherit from the base types which are actually GTDL data types, the tagged type tuples know how to call the routines defined in the GTSI. The tag information is saved in the **LTC** of the GTDL so that it can be converted to the ASN.1 tag when the BER encoding rule is used as a transfer syntax. For example, the following segments of the application program are valid statements for encoding **BOOLEAN** and **INTEGER** types with the BER.

```
TSI<PPBER>  pe;
BOOLEAN     flag = TRUE;
```

```
INTEGER      num = 1;

flag.encode(pe);          // calls TSI<PPBER>::prim2pe(*this)
num.encode(pe);           // calls TSI<PPBER>::prim2pe(*this)
```

The details of the caty generated C++ representations of ASN.1 specifications are not within the scope of this thesis. However, they are used in Chapter 7 as a means of testing operational feasibility of the new ASN.1 class hierarchy built upon the OODSF.

Chapter 7

Functional Testing

This chapter describes the tests performed to demonstrate the operational condition of the OODSF. Each test case incorporates a fairly complex data structure and tests both the representational structures in the OODSF as well as the associated encoding and decoding routines. These test cases represent a reasonable, though not an exhaustive, test suite by constructing several levels of nested data type descriptions. XDR and BER are the two encoding rules used for test cases. XDR was tested for both encoding and decoding using a **TSI<XDR>** object. Only encoding was tested for BER using a **TSI<PPBER>** object, which displays a human readable form for an encoded value.

7.1 Testing Environment

To simulate a client, which encodes data to produce a transfer syntax (stream) for sending, and a server, which decodes the transfer syntax after receiving it, the test drivers are implemented as two separate subroutines called **sender** and **receiver**. The **sender** (client) generates a stream, **stream1**, by encoding data. Sending of this stream to the

server is simulated by copying this stream to the server's stream, **stream2**. The **receiver** (server) decodes this stream to local data.

```
void main()
{
        DStream stream1, stream2;

                sender(stream1);

                stream2 << stream1;      // copying

                receiver(stream2);
}
```

In the above code, the **sender** encodes all values into the **DStream** (**stream1**), and the **main** routine simulates the delivery of the message stream with copying from **stream1** to **stream2**. Then, the **receiver** decodes the **DStream** (**stream2**).

To test encoding and decoding for a complex data structure, the several levels of nested data type descriptions are used. For example in IDL, the outermost data structure, **union_t**, is described as an **union** type which includes a more complex **struct** type.

```
union union_t switch(enum enum1 {first, second, third}) {
        case first  :     struct_t    struct_1;
        case second :     long        long_1;
        case third  :     char        char_1;
};
```

In addition to testing of the complex data types, all predefined data types of IDL are specified within **struct_1** so that the test covers all possible individual IDL data types.

```
struct basic_t {
        short           short_1;
        unsigned short  ushort_1;
        long            long_1;
        unsigned long   ulong_1;
        float           float_1;
        double          double_1;
        char            char_1;
        octet           octet_1;
        boolean         boolean_1;
        string          string_1;
        any             any_basic_t;
};

struct struct1 {
        long            long_1, long_2;
        string<256>     string_1;
};


union union1  switch(enum enum1 {first, second, third}) {
        case first:
                struct1 struct1_1;
        case second:
                long long_1;
        case third:
                char char_1;
        default:
                any  any_union1;
};

struct struct_t {
        char                array_1[6];
        sequence<char,10>   sequence_1;
        basic_t             struct_1;
        union1              union_1;
        any                 any_struct_t;
};

enum idl_type {basic_type, structured_type};

union datatypes switch(idl_type) {
        case basic_type:
                basic_t   basic_data;
        case structured_type:
                struct_t  struct_data;
        default:
                any      any_datatypes;
};
```

Figure 7.1 IDL test description

7.2 Testing IDL Class Library with GTSI

The test IDL specification covers all basic IDL data types in the fairly complex data structures by defining several levels of nested descriptions as shown in Figure 7.1.

In addition to the testing environment described in Section 7.1, testing IDL data types involves one more additional step: the manual translation of the IDL specification to the C++ representation. Note that the IDL translator is not built yet so the manual translation is necessary. The corresponding C++ type binding using IDL class library is done manually according to the mechanism described in Chapters 4 and 6. For example, **datatypes** described in the beginning of this section is manually translated as shown Figure 7.2.

```
idl_enum idl_type {basic_type, structured_type, DEFAULT};

class datatypes : public idl_union {
public:

    _CASE<basic_type,        basic_t>        basic_data;
    _CASE<structured_type,   struct_t>       struct_data;
    _CASE<DEFAULT,           idl_any>        any_datatypes;

    datatypes() {   _LINK(basic_data);    _LINK(struct_data);
                    _LINK(any_datatypes);                     }

    void operator=(const datatypes& v) { idl_union::operator=(v); }
};
```

Figure 7.2 C++ Translation for **datatypes**

XDR and BER encoding rules were applied using **TSI<XDR>** and **TSI<PPBER>** of the GTSI system to test the functionality of conversions with respect to the data type descriptions listed above. XDR using **TSI<XDR>** is tested using the above data types. BER is tested with **TSI<PPBER>** without decoding since the **TSI<PPBER>** is a pretty-printing class, which emulates the actual encoding with BER.

```
DEFINITIONS ::=
BEGIN
BasicT ::= [APPLICATION 1] SEQUENCE {
        integer1        [0]  INTEGER,
        real1           [1]  REAL,
        bool1           [2]  BOOLEAN,
        bits1           [3]  BIT STRING,
        str1            [4]  IA5String,
        octetstr1       [5]  OCTET STRING,
        any1            [6]  ANY
}

ChoiceT ::= CHOICE  {
        integer1        [0]  INTEGER,
        real1           [1]  REAL,
        any1            [2]  ANY
}

StructT ::= [APPLICATION 2] SET {
        seq1            [0] BasicT,
        choice1         [1] Choice,
        any1            [2] ANY
}

DataTypes ::= CHOICE {
        basicdata       [0]     BasicT,
        structdata      [1]     StructT,
        anydatatypes    [2]     ANY
}


END
```

Figure 7.3 ASN.1 test description

7.3 Testing ASN.1 Class Library with GTSI

To test the type binding of ASN.1 in C++, a complex ASN.1 specification similar to the IDL specification in Section 7.2 was designed . Then, caty was used to produce an equivalent C++ representation. Figure 7.3 shows a ASN.1 specification tested.

The above ASN.1 specification was processed by caty to produce a C++ representation. Then, the C++ code produced is compiled with the ASN.1 class library to test encoding/decoding using **TSI<XDR>** and **TSI<PPBER>**.

7.4 Mixture: GTDL, ASN.1, and IDL

One additional group of tests were performed to demonstrate the functionality of the OODSF as a common underlying architecture for both IDL and ASN.1 class libraries. A class, called **Super**, is designed as a collection of the mixed data types from GTDL, IDL, and ASN.1 libraries as shown below.

```
class Super : public _Struct {
public:
        idl_short                   short_1;        // IDL
        REAL                        real_1;         // ASN.1
        idl_char                    char_1;         // IDL
        idl_array<INTEGER, 3>       array_1;        // IDL
        _Bits                       bits_1;         // GTDL
        ANY                         any_1;          // ASN.1

        Super() { _LINK(short_1);       _LINK(real_1);
                  _LINK(char_1);        _LINK(array_1);
                  _LINK(bits_1);        _LINK(any_1);
        }

       void operator=(const Super &v) { _Struct::operator=(v); }
};
```

Similar to the previous sections, the functionality of conversions with respect to transfer syntaxes is tested.

7.5 Discussion

To demonstrate applicability in heterogeneous systems, all test cases described earlier are performed on different platforms: Solaris 2.3 running on a SUN workstation (model Sparc 10) and OSF/1 running on a DEC Alpha machine. Using the pretty-printing methods embedded in each GTDL data type, functional correctness is tested and evaluated before and after a transfer syntax is applied. Hence, Table 7.1 illustrates the result of testing with respect to various cases.

Table 7.1 Testing Results

	GTDL	IDL	ASN.1	Mixture
XDR	ok	ok	ok	ok
BER	ok	ok	ok	ok

Performance issues with respect to the OODSF are not addressed in this thesis because it is more dependent on the choice of an encoding rule among various existing rules, and the performance study with respect to different encoding rules is beyond the scope of this thesis. However, a performance study with respect to different kinds of encoding rules is presented in [SAM94].

Chapter 8

Conclusions and Future Work

Developing a distributed application incorporates such varied issues as communication protocols, synchronous/asynchronous communications, data conversions, control of multiple threads, and so on. However among these issues, the basic mechanism permitting heterogeneity in a distributed system is a specification language for describing transferred data which allows each local machine to understand the data transmitted from other machines. This specification language is incorporated in a programming environment by binding it to a selected programming language.

In this thesis, the OODSF has been presented to facilitate C++ programming in a heterogeneous distributed environment, allowing a C++ language binding for a specification language. Specifically, the OODSF has been designed with the objectives of accommodating: multiple specification languages (IDL and ASN.1), multiple transfer syntaxes (XDR and BER), simple translations, and natural-to-use bindings.

To achieve the objectives, the OODSF has been structured as a layered model. The major components of this layering are GTDL and GTSI. The GTDL defines an extensible

set of data types, or a type core. This type core is extended to both IDL and ASN.1 class libraries, allowing the OODSF to deal with multiple specification languages. Also, the class libraries, defined using the type core, allow an easy translation of an application-level specification to C++ by freeing the translation from defining the details of data manipulations and conversions. This translation is a high-level systematic definition using the methods provided by the class libraries. The extensive use of classes and templates, along with list-oriented data structures, allows the translated representations to be easy and natural to use, much as in a normal C/C++ programming environment. The GTSI is a general interface to different encoding/decoding services, allowing the OODSF to cope with multiple transfer syntaxes. Currently, two encoding rules, XDR and BER, have been used in the OODSF to convert to and from a transfer syntax.

Compared to other systems, the OODSF is distinct in that it can cope with multiple specification languages as well as multiple transfer syntaxes. While all other systems only use one specification language and one encoding rule for a transfer syntax, the OODSF is much more flexible. The immediate result of this flexibility is that a distributed prototype built using the OODSF has a potential to communicate with systems that use various combinations of specification languages (IDL or ASN.1) and encoding rules (XDR,BER, etc.). An OODSF-based translation to C++ is also relatively simple. For example, the translated code is relatively small compared to several other systems, even allowing a manual translation. In one case, the OODSF-based translation produced about 10 times less in code size, compared to another system. The OODSF-based translation also provides a similar syntax as in a normal C/C++, allowing an easy-to-use programming environment.

In addition to the current implementation of the OODSF prototype, there is much work yet to do to improve the OODSF and to demonstrate the utility of the OODSF in a distributed system:

- Adopt additional encoding rules such as Network Data Representation (NDR) and Common Data Representation (CDR).
- Integrate the IDL class library with the distributed ACT++ model [LEE90, KML93, KHA94, VEN94, KHK94], which is currently under development.
- Develop a translator which maps a IDL specification into a C++ representation specific to the IDL class library and the operation invocation interface specific to the distributed ACT++ model.
- Define a specification language other than IDL and ASN.1 using the OODSF.

The ACT++ system is a C++ implementation of the actor model [AGH86] in which an application is structured as a collection of interacting concurrent objects (actors). Previously, the ACT++ model has been implemented as a concurrent object-oriented programming tool in a shared memory environment and IDL has been used as a specification language for the data passed in messages between actors [KHA94, KHK94]. As ACT++ was ported to a distributed environment, both the previous ASN.1 class library and the previous IDL mapping were inadequate. The previous mapping of IDL was restricted to a shared memory environment and ASN.1 does not allow the specification of operations which ACT++ requires. Consequently, the principal result of the OODSF is the definition of IDL class library which allows a complete integration with the distributed ACT++ prototype. As the IDL class libraries are integrated with the distributed ACT++ system, it will be necessary to investigate and test the OODSF further with respect to:

- more realistic testing as an integral part of a working system
- performance
- interoperability with other non ACT++ systems

The first steps toward the integration of the OODSF and ACT++ are currently underway, and it has demonstrated successful preliminary results. The new ASN.1 class library has replaced the old ASN.1 class library and XDR encoding/decoding performed correctly for messages among distributed actors. As this integration is completed, the next step will be the integration of the IDL class library with the ACT++.

The OODSF is not restricted to ACT++, but also it can be adopted into a different distributed system for building a data specification component. The flexible nature of the OODSF also can allow researchers to experiment with different platforms of specification languages as well as different encoding rules, providing a basis for their new ideas.

REFERENCES

[AGH86] G. A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA, 1986.

[ANS92] American National Standards Institute (ANSI). *Z39.50 Information Retrieval Service Definitions and Protocol Specification for Library Applications*, July 1992.

[BBN95] BBN. BBN ASN.1 to C++ compiler, Jan 1995. Available at http://ests.bbn.com/ASNSRC.html.

[BEV93] M. Bever, K. Geihs, L. Heuser, M. Muhlhauser, and A. Schill. "Distributed Systems, OSF DCE, and Beyond," *DCE-The OSF Distributed Computing Environment*, Lecture Notes in Computer Science #731, pp. 1-20, Springer-Verlag, New York, 1993.

[COM93] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP: Client-Server Programming and Applications*, vol. 3, Prentice Hall, Englewood Cliffs, NJ, 1993.

[DAV90] F. Davis, B. Kahle, H. Morris, J. Salem, and T. Shen. "WAIS interface protocol prototype functional specification," Technical Report version 1.5, Thinking Machines Corporation, April 1990.

[FED90] J. Case, M. Fedor, M. Schoffstall, and C. Davin. "Simple Network Management Protocol (SNMP)," Internet Protocol Specification, RFC 1157, Network Information Center, SRI International, May 1990.

[HAS92] T. Hasegawa, S. Nomura, and T. Kato. "Implementation and Evaluation of ASN.1 Compiler," *Journal of Information Processing*, vol. 15, Iss. 2, pp. 157-167, 1992.

[HDC93] HyperDesk Corporation. "Second Revised Submission in Response to the OMG RFP for a C++ Language Mapping," OMG Document 93-11-5, November 1993.

[HPM94] T. Moreau and S. Vinoski. "C++ Mapping for the IDL any Type," Hewlett-Packard Company, OMG Document 93-12-14, February 1994.

[HPV93] S. Vinoski. "A Review of the IDL C++ Mapping Submissions," Hewlett-Packard Company, OMG Document 93-12-19, December 1993.

[INS93] IONA Technologies Ltd., NEC Corporation, and SunSoft, Inc. "IDL C++ Language Mapping Specification," OMG Document 93-9-3, November 1993.

[ISO87a] International Organization for Standardization (ISO). *Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)*, 1987. ISO 8824 /CCITT X.208.

[ISO87b] International Organization for Standardization (ISO). *Information Processing - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One*, 1987. ISO 8825/CCITT X.209.

[ISO92] International Organization for Standardization (ISO). *Specification of ASN.1 Encoding Rules - Part 2: Packed Encoding Rules*, 1992. ISO 8825-2.

[KHA94] A. Khare. "ACT++ 3.0 - Implementation of the Actor Model Using POSIX Threads, M.S. report," Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.

[KHE94] R. Khera. "The design of C++ class hierarchy to model ASN.1 data types," M.S. report, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.

[KHK94] D. G. Kafura, J. W. Hwang, and A. Khare. Implementation of ACT++ 3.0, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.
Available at ftp://actor.cs.vt.edu/pub/Act++3.0.tar.Z.

[KLM94] D. G. Kafura, R. G. Lavender, and R. W. Mullins. "Programming with ASN.1 Using Polymorphic Types and Type Specialization," Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.
Available at ftp://actor.cs.vt.edu/pub/oosi/papers/ulpaa-asn1.ps.Z.

[KML93] D. G. Kafura, M. Mukherji, and G. Lavender. "ACT++: A class library for concurrent programming in C++ using actors," *Journal of Object-Oriented Programming*, vol. 6, no 6, October 1993.

[KON90] M. Kong. *Network Computing System Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1990.

[LAV93] R. G. Lavender. "Polymorphic Types for Constructing Concurrent Objects and Layered Communication Protocols," Ph.D. dissertation, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1993.

[LEE90] K. H. Lee. "Designing A Statically Typed Actor-Based Concurrent Object-Oriented Programming Language," Ph.D. dissertation, Virginia Polytechnic

Institute and State University, Department of Computer Science, Blacksburg, VA, 1990.

[LIS86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*, MIT Press, Cambridge, MA, 1986.

[LON94] W. Long. "CATY: an ASN.1-C++ translator in support of distributed object-oriented applications," M.S. report, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.

[MUL93] R. W. Mullins. "Separating Representations and Translation of Shared Data," M.S. report, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1993.

[OMG91] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, Draft 10, OMG Document 91-12-1, December 1991.

[OMG92] Object Management Group (OMG). *C++ Language Mapping Request For Proposals*, OMG Document 92-12-11, August 1992.

[OMG94] Object Management Group (OMG). *IDL C++ Language Mapping Specification*, OMG Document 94-9-14, September, 1994.
Available at http://www.omg.org/docs/94-9-14.ps

[OMG95] Object Management Group (OMG). *CORBA 2.0/Interoperability*, OMG Document, Draft, March 1995.
Available at http://www.omg.org/docs/95-3-10.ps

[PMC94a] PostModern Computing. *ORBeline User's Guide*, PostModern Computing Technologies, Inc., 1994.
Available at ftp://labrea.stanford.edu/pub/pomoco/orbeline.docs.tar.Z.

[PMC94b] PostModern Computing. *ORBeline Reference Manual*, PostModern Computing Technologies, Inc., 1994.
Available at ftp://labrea.stanford.edu/pub/pomoco/orbeline.docs.tar.Z.

[ROS90] M. T. Rose. *The Open Book: A Practical Perspective On OSI,* Prentice Hall, Englewood Cliffs, NJ, 1990.

[SAM93] M. Sample. SNACC 1.1: A High Performance ASN.1 to C/C++ Compiler. University of British Columbia, July 1993.
Available at ftp://ftp.cs.ubc.ca/pub/local/src/snacc/snacc1.1.tar.Z.

[SAM94] M. Sample and G. Neufeld. "High-performance ASN.1 compiler," *Computer Communications*, vol. 17, Iss. 3, pp. 156-171, 1994.

[SEB93] R.W. Sebesta. *Concepts of Programming Languages*, second edition, Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1993.

[SHI92] J. Shirley, W. Hu, and D. Magid. *Guide to Writing DCE Applications*, O'Reilly and Associates, Inc., Sebastopol, CA, 1992.

[STE90] D. Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference*, Technology Appraisals Ltd., England, 1990.

[STR91] B. Stroustrup. *The C++ Programming Language*, second edition. Addison-Wesley Publishing Company, Reading, MA, 1991.

[SUN87] Sun Microsystems Inc. *XDR: External Data Representation Standard*, RFC 1014, Network Information Center, SRI International, June 1987.

[TAN89] A. S. Tanenbaum. *Computer Networks*, second edition, Prentice Hall, Englewood Cliffs, NJ, 1989.

[VEN94] V. R. Vykunta. "Class Management in a Distributed Actor System," M.S. report, Virginia Polytechnic Institute and State University, Department of Computer Science, Blacksburg, VA, 1994.

[WEI93] Y. Wei and C. Wu. "Integrating RPC and Message Passing for Distributed Programming," *DCE-The OSF Distributed Computing Environment*, Lecture Notes in Computer Science #731, pp. 214-228, Springer-Verlag, New York, 1993.

[XER81] XEROX Corporation. *Courier: The Remote Procedure Call Protocol*, xsis-038112, December 1981.

VITA

Jae Woong Hwang was born on May 19, 1968 in Seoul, Korea. He received a Bachelor of Science degree in Computer Science from Virginia Polytechnic Institute and State University, Blacksburg, Virginia, in 1992. He started graduate studies at the same school in August 1992. The work reported in this thesis is the result of the IDL-related research between January 1994 and December 1994, and an intensive implementation for the OODSF between January 1995 and May 1995. He was involved in various developments of software systems. During his research period, he was involved in the development of the ACT++ 3.0 system, writing an IDL compiler, and the OODSF presented in this thesis. He served as a teaching assistant for classes in Multimedia, Unix, and Information Storage and Retrieval, and as a lab instructor for MIPS assembly language. Currently, he is involved in the integration of the OODSF with the distributed ACT++ system. His future research interests include object-oriented programming, distributed systems, network, and computer graphics.