

CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO PARANÁ
Curso de Pós-Graduação em Engenharia Elétrica e Informática Industrial

DISSERTAÇÃO
apresentada ao CEFET-PR
para obtenção do título de

MESTRE EM CIÊNCIAS

por

DIERONE CÉSAR FOLTRAN JÚNIOR

**DESENVOLVIMENTO DE UMA FERRAMENTA
PARA TRADUÇÃO DAS ESPECIFICAÇÕES
GDMO/OSI PARA IDL/CORBA**

Banca Examinadora :

Presidente e Orientador:

Prof. Dr. MANOEL CAMILLO DE O. PENNA NETO CEFET-PR

Examinadores:

Prof. Dr. ANTONIO MAURO BARBOSA DE OLIVEIRA ETF/CE

Prof. Dr. LUIZ NACAMURA JÚNIOR CEFET-PR

Prof^ª. Dr^ª. KEIKO VERÔNICA ONO FONSECA (*suplente*) CEFET-PR

Curitiba, 9 de Novembro de 1998.

Ato 138
Deferido 09/11/98

DIERONE CÉSAR FOLTRAN JÚNIOR

**Desenvolvimento de uma Ferramenta para Tradução das
Especificações GDMO/OSI para IDL/CORBA**

Dissertação apresentada como requisito parcial à
obtenção do título de “Mestre em Ciências”. Área
de Concentração: Telemática.

Curso de Pós-Graduação em Engenharia Elétrica e
Informática Industrial - CPGEI.

Centro Federal de Educação Tecnológica do
Paraná - CEFET-PR.

Orientador: Prof. Dr. Manoel Camillo Penna.

CURITIBA

1998

Aos meus pais, especialmente meu pai (in memoriam), por todos os esforços e sacrifícios realizados para me proporcionar uma educação de qualidade.

A Elenice, minha fiel companheira, pela sua presença constante e efetiva, que com seu estímulo e sua ajuda me faz enfrentar e ultrapassar desafios.

AGRADECIMENTOS

Para que este trabalho fosse realizado, muitas pessoas dele participaram direta e indiretamente. Neste momento, apresento os meus agradecimentos a todas elas, especialmente

a Deus, pelo dom da vida, da inteligência e pela sua energia divina sempre presente em minha existência;

ao professor Dr. Manoel Camillo Penna, por acreditar no meu potencial e por estar sempre presente, conduzindo de forma coerente todo o processo de construção deste trabalho;

ao Marcelo Ferrasa, por sua amizade e apoio constante nestes anos de mestrado, sempre me incentivando a continuar;

aos colegas do laboratório de redes do CEFET-PR, que tornaram a experiência do mestrado ainda mais interessante;

à todos aqueles que, direta e indiretamente, apoiaram e incentivaram para tornar possível a realização do mestrado;

ao Departamento de Informática da Universidade Estadual de Ponta Grossa-UEPG, em especial ao Furquim, pelo incentivo constante.

SUMÁRIO

LISTA DE FIGURAS	IX
LISTA DE TABELAS.....	XI
LISTAS DE ABREVIATURAS E SIGLAS	XIII
RESUMO	XV
ABSTRACT.....	XVI
CAPÍTULO 1 - INTRODUÇÃO	1
1.1 Objetivos.....	2
1.2 Trabalhos Correlatos	3
1.3 Apresentação do Trabalho	5
CAPÍTULO 2 - CORBA - COMMON OBJECT REQUEST BROKER ARCHITECTURE	7
2.1 Introdução	7
2.2 Object Management Group (OMG)	7
2.3 Modelo Objeto OMG	8
2.4 O Modelo Objeto CORBA	8
2.5 A Especificação CORBA	11
2.6 Interface Definition Language – IDL	15
2.7 Considerações Finais	21
CAPÍTULO 3 - ARQUITETURA DE GERENCIAMENTO OSI - OPEN SYSTEM INTERCONNECTION	23
3.1 Introdução	23
3.2 Guidelines for Definition of Managed Objects - GDMO	28
3.3 Notação de Sintaxe Abstrata - ASN.1	51
3.4 Considerações Finais	66
CAPÍTULO 4 - COMPARAÇÃO ENTRE CORBA E OSI	67
4.1 Introdução	67
4.2 Objetivos do Modelo	67
4.3 Interfaces	68
4.4 Características dos Objetos	70
4.5 Especificação e Instância de Objetos	73
4.6 Taxonomia de Objetos	74
4.7 Referência a Objeto	75
4.8 Seleção de Objeto e Resolução do Endereço	78
4.9 Similaridades e Diferenças	79

4.10 Alternativas de Reconciliação dos Modelos	80
4.11 Considerações Finais	81
CAPÍTULO 5 - TRADUÇÃO DE GDMO-ASN.1/OSI PARA IDL/CORBA	83
5.1 Introdução	83
5.2 Tradução de Tipos ASN.1 para Tipos IDL	84
5.3 Tradução de Classes e Módulos GDMO para Interfaces IDL	101
5.4 Considerações Finais	120
CAPÍTULO 6 - DESENVOLVIMENTO DO TRADUTOR DE GDMO/OSI PARA IDL/CORBA	121
6.1 Introdução	121
6.2 Tradutor de ASN.1	122
6.3 Tradutor de GDMO	125
6.4 Considerações Finais	131
CAPÍTULO 7 - CONCLUSÃO E TRABALHOS FUTUROS	133
ANEXOS	137
ANEXO 1 - GRAMÁTICA IDL CORBA	139
ANEXO 2 - DEFINIÇÕES DE DADOS DO TRADUTOR DE GDMO	143
ANEXO 3 - TEMPLATES GDMO	145
ANEXO 4 - NOTAÇÃO ASN.1	149
ANEXO 5 - EXEMPLO DE TRADUÇÃO GDMO PARA IDL	151
ANEXO 6 - SCRIPT DE TRADUÇÃO: TRADUZ.SCT	153
ANEXO 7 - ARQUIVOS ESPECIFICADOS PELO X/OPEN	155
GLOSSÁRIO	157
BIBLIOGRAFIA	161

LISTA DE FIGURAS

Figura 1 - Envio de requisições através de um ORB	11
Figura 2 - Estrutura de um ORB	12
Figura 3 - Repositórios de Interface e Implementação	13
Figura 4 - Definição de um módulo em IDL CORBA	16
Figura 5 - Definição de uma interface em IDL CORBA	17
Figura 6 - Definição de herança de interface em IDL CORBA	17
Figura 7 - Exemplo de operação em IDL CORBA	17
Figura 8 - Declaração de um atributo em IDL CORBA	18
Figura 9 - Declaração de um novo tipo em IDL CORBA	18
Figura 10 - Exemplo de tipo enumerado em IDL CORBA	19
Figura 11 - Declaração de uma estrutura em IDL CORBA	20
Figura 12 - Declaração de uma <i>sequence</i> em IDL CORBA	20
Figura 13 - Declaração de uma <i>string</i> em IDL CORBA	20
Figura 14 - Declaração de uma <i>union</i> em IDL CORBA	20
Figura 15 - Declaração de matriz em IDL CORBA	21
Figura 16 - Declaração do tipo <i>any</i> em IDL CORBA	21
Figura 17 - Declaração de constante em IDL CORBA	21
Figura 18 - Declaração de constante em IDL CORBA	21
Figura 19 - Componentes do sistema de Gerenciamento OSI	25
Figura 20 - O Modelo de Gerenciamento OSI	26
Figura 21 - Normas de Gerência OSI	27
Figura 22 - Exemplo de árvore de nomeação	31
Figura 23 - <i>Template</i> MANAGED OBJECT	32
Figura 24 - Referências entre <i>templates</i>	33
Figura 25 - Modelo da <i>template</i> ATTRIBUTE	36
Figura 26 - Modelo da <i>template</i> ACTION	37
Figura 27 - Modelo da <i>template</i> NOTIFICATION	39
Figura 28 - Modelo da <i>template</i> BEHAVIOUR	40
Figura 29 - Modelo da <i>template</i> ATTRIBUTE GROUP	41
Figura 30 - Modelo da <i>template</i> PACKAGE	43
Figura 31 - Modelo da <i>template</i> MANAGED OBJECT CLASS	45
Figura 32 - Exemplo de utilização de <i>name binding</i>	47
Figura 33 - Exemplo de árvore de nomeação	47
Figura 34 - Modelo da <i>template</i> NAME BINDING	48
Figura 35 - Modelo da <i>template</i> PARAMETER	50
Figura 36 - Mecanismo de tradução	84
Figura 37 - Representação de dados interna do tradutor GDMO	102
Figura 38 - Organograma do processo de transformação	103
Figura 39 - Hierarquia de Herança em IDL/CORBA	106
Figura 40 - Estrutura da <i>template</i> MANAGED OBJECT CLASS	106
Figura 41 - Representação interna da <i>template</i> MANAGED OBJECT CLASS	107
Figura 42 - Exemplo de <i>template</i> MANAGED OBJECT CLASS	108
Figura 43 - Implementação da tradução da <i>template</i> MANAGED OBJECT CLASS	108

Figura 44 - Estrutura da <i>template</i> PACKAGE	109
Figura 45 - Representação interna da <i>template</i> PACKAGE.	110
Figura 46 - Exemplo de <i>template</i> PACKAGE.....	111
Figura 47 - Implementação da tradução da <i>template</i> PACKAGE.....	111
Figura 48 - Estrutura da <i>template</i> ATTRIBUTE	112
Figura 49 - Representação interna da <i>template</i> ATTRIBUTE.	112
Figura 50 - Estrutura da <i>template</i> ATTRIBUTE GROUP	113
Figura 51 - Representação interna da <i>template</i> ATTRIBUTE GROUP.	114
Figura 52 - Estrutura da <i>template</i> PARAMETER.....	114
Figura 53 - Representação interna da <i>template</i> PARAMETER.	115
Figura 54 - Estrutura da <i>template</i> ACTION.....	116
Figura 55 - Representação interna da <i>template</i> ACTION.....	116
Figura 56 - Estrutura da <i>template</i> NOTIFICATION	117
Figura 57 - Representação interna da <i>template</i> NOTIFICATION.	117
Figura 58 - Estrutura da <i>template</i> NAME BINDING.....	118
Figura 59 - Representação interna da <i>template</i> NAME BINDING.....	118
Figura 60 - Estrutura da <i>template</i> BEHAVIOUR.....	119
Figura 61 - Representação interna da <i>template</i> BEHAVIOUR.....	119
Figura 62 - Mecanismo de tradução.....	122
Figura 63 - Tradutor de ASN.1 para IDL/CORBA.....	123
Figura 64 - Estrutura do arquivo ASN1.DB.....	125
Figura 65 - Estrutura de dados inicial utilizada pelo tradutor GDMO.	126
Figura 66 - Tradutor de GDMO para IDL/CORBA.	126
Figura 67 - Estrutura do arquivo MOC.DB.....	127
Figura 68 - Tradutor de GDMO para IDL/CORBA.	127
Figura 69 - Processo de tradução de especificações GDMO e ASN.1 para IDL/CORBA.	128
Figura 70 - Arquitetura da tradução GDMO/OSI para IDL/CORBA.....	131

LISTA DE TABELAS

Tabela 1 - Tipos simples em IDL CORBA	19
Tabela 2 - Relação de <i>tags</i> Universais	62
Tabela 3 - Comparação entre Modelos Objeto OSI e OMG	68
Tabela 4 - Comparação de Interfaces entre OSI e OMG	69
Tabela 5 - Características de Objetos	71
Tabela 6 - Comparação de objetos e instanciação	73
Tabela 7 - Comparação da Taxonomia de Objetos	75
Tabela 8 - Comparação para Referência de Objetos	77
Tabela 9 - Comparação do Mecanismo de Seleção de Objetos	78
Tabela 10 - Principais similaridades entre os modelos OSI e OMG	80
Tabela 11 - Mapeamento de tipos primitivos	88
Tabela 12 - Mapeamento do tipo BOOLEAN	88
Tabela 13 - Mapeamento do tipo INTEGER	88
Tabela 14 - Mapeamento do tipo REAL	89
Tabela 15 - Mapeamento do tipo ENUMERATED	89
Tabela 16 - Mapeamento do tipo BIT STRING	90
Tabela 17 - Mapeamento do tipo OCTET STRING	91
Tabela 18 - Mapeamento do tipo OBJECT IDENTIFIER	93
Tabela 19 - Mapeamento do tipo ANY	93
Tabela 20 - Mapeamento de tipos construtores ASN.1	94
Tabela 21 - Mapeamento do tipo SEQUENCE	95
Tabela 22 - Mapeamento do tipo CHOICE	96
Tabela 23 - Mapeamento do tipo SEQUENCE	98
Tabela 24 - Mapeamento do tipo SEQUENCE OF	99
Tabela 25 - Tradução da <i>template</i> MANAGED OBJECT CLASS	106
Tabela 26 - Definição da estrutura interna da <i>template</i> MANAGED OBJECT CLASS	107
Tabela 27 - Tradução da <i>template</i> PACKAGE	109
Tabela 28 - Definição da estrutura interna da <i>template</i> PACKAGE	110
Tabela 29 - Tradução da <i>template</i> ATTRIBUTE	112
Tabela 30 - Definição da estrutura interna da <i>template</i> ATTRIBUTE	113
Tabela 31 - Definição da estrutura interna da <i>template</i> ATTRIBUTE GROUP	114
Tabela 32 - Tradução da <i>template</i> PARAMETER	115
Tabela 33 - Definição da estrutura interna da <i>template</i> PARAMETER	115
Tabela 34 - Tradução da <i>template</i> ACTION	116
Tabela 35 - Definição da estrutura interna da <i>template</i> ACTION	117
Tabela 36 - Tradução da <i>template</i> NOTIFICATION	117
Tabela 37 - Definição da estrutura interna da <i>template</i> NOTIFICATION	118
Tabela 38 - Definição da estrutura interna da <i>template</i> NAME BINDING	119
Tabela 39 - Definição da estrutura interna da <i>template</i> BEHAVIOUR	119

LISTAS DE ABREVIATURAS E SIGLAS

ACSE	Association Control Service Element
ASE	Application Service Element
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
BNF	Backus-Naur Form
CMIP	Common Management Information Protocol
CMIS	Common Management Information Service
CMISE	Common Management Information Service Element
CORBA	Common Object Request Broker Architecture
COSS	Common Object Services Specification
DMI	Definition of Management Information
GDMO	Guidelines for Definition of Managed Objects
GMI	Generic Management Information
GOM	Generic Object Model
IDL	Interface Definition Language
ISO	International Organization for Standardization
LME	Layer Management Entity
MIB	Management Information Base
MIM	Management Information Model
NMF	Network Management Forum
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open Systems Interconnection

PDU	Protocol Data Unit
RDN	Relative Distinguished Name
RM-OSI	Reference Model Open Systems Interconnection
ROSE	Remote Operations Service Element
SACF	Single Association Control Function
SAP	Service Access Point
SMAE	System Management Application Entity
SMASE	System Management Application Service Element
SMI	Structure of Management Information
SNMP	Simple Network Management Protocol

RESUMO

Atualmente existem diversas tecnologias atuando no serviço de gerência de sistemas, entre elas destacamos o uso do modelo OSI e CORBA. Estas tecnologias podem coexistir dentro de uma organização. Para que esta coexistência seja possível é necessário que as informações entre elas sejam compartilhadas. A proposta deste trabalho é uma ferramenta que possibilite ao gerente destes sistemas, possivelmente seu implementador, a transformação das especificações de gerência do domínio OSI para o domínio CORBA, ou seja, a transformação das especificações em GDMO/OSI para IDL/CORBA.

PALAVRAS-CHAVE

IDL - CORBA / GDMO - OSI / Interoperabilidade

ÁREA / SUB-ÁREA DE CONHECIMENTO

Telemática / Sistemas Distribuídos

ABSTRACT

At present several technologies exist acting in the service of management of systems, among them we distinguished the use of the model OSI and CORBA. These technologies can coexist inside of an organization. To this coexistence was been possible is necessary that the information between OSI and CORBA are shared. The proposal of this work is a tool that facilitates the manager of these systems, possibly its implementador, the transformation of the specifications of management of the domain OSI for the domain CORBA, that is, the transformation of the specifications in GDMO/OSI for IDL/CORBA.

CAPÍTULO 1

INTRODUÇÃO

As redes de computadores têm evoluído continuamente, levando-nos a um ambiente composto por redes e equipamentos heterogêneos. Esta heterogeneidade resulta na complexa tarefa de gerenciar este ambiente. As soluções proprietárias, adotadas por alguns fabricantes, tornam a gerência de redes ineficiente, envolvendo problemas como por exemplo, a presença de múltiplas interfaces para diferentes sistemas, sistemas não interoperáveis, insuficiência de informações coletadas, base de dados específicas e isoladas com informações redundantes e inconsistentes. Assim, a motivação deste trabalho origina-se na necessidade de desenvolver ferramentas que habilitem a cooperação entre o gerenciamento de sistemas baseados em diferentes tecnologias.

No mundo real há diversas tecnologias que são apropriadas para resolver a tarefa de gerenciamento. Cada uma tem qualidades e limitações, e podem ajudar a definir as novas características dos futuros sistemas de gerenciamento de redes. Um grupo de trabalho do X/Open-Network Management Forum (NMF) identificou três tecnologias chaves que podem contribuir neste contexto: CMIP (*Common Management Information Protocol*), SNMP (*Simple Network Management Protocol*) e CORBA (*Common Object Request Broker Architecture*), e está procurando habilitar a interoperabilidade entre elas.

A tecnologia SNMP (*Simple Network Management Protocol*) tem uma larga base no mercado de computação. A tecnologia CMIP (*Common Management Information Protocol*) é o padrão definido para a área de telecomunicações através do modelo TMN (*Telecommunications Management Network*). A tecnologia CORBA (*Common Object Request Broker Architecture*) é reconhecidamente um padrão emergente cobrindo o paradigma de objetos distribuídos. Cada uma destas tecnologias tem sua vantagem, consequentemente uma interoperabilidade completa entre elas possibilitará ao projetistas de sistemas de gerência, selecionar a tecnologia mais apropriada a ser aplicada sobre algum tipo de problema.

Para habilitar a cooperação, é necessário ser capaz de mapear os modelos objetos envolvidos e construir um mecanismo de conversão de protocolo entre diferentes domínios. Para um par particular de domínios, a especificação de mapeamento é dividida em duas partes. A primeira se refere à *Tradução de Especificações*, que é responsável pelo processo de tradução de especificações de um domínio para outro. No contexto deste trabalho, uma *Tradução de Especificação* expressa um mecanismo para tradução entre GDMO/OSI e IDL/CORBA.

A segunda parte é conhecida como *Tradução Interativa*, e corresponde ao mecanismo de conversão dinâmica entre os protocolos de um domínio para protocolos de outros domínios. Isto permite aos objetos de um domínio serem representados em outros domínios, e as interações podem ser governadas pelo domínio de escolha, em vez do domínio em que o objeto alvo está implementado. Para a tradução interativa já existe atualmente diversos trabalhos sendo realizados, principalmente para a implementação em sistemas TMN, mas todos são específicos para cada caso de implementação.

As padronizações atuais apresentam uma modelagem orientada a objeto, havendo hoje algumas organizações que buscam uniformidade, sendo um exemplo disto o Object Management Group (OMG), um consórcio de fornecedores de tecnologia orientada à objetos, que tem se ocupado de especificar a arquitetura de software aberto, no qual os componentes objetos escritos por diferentes fornecedores possam interoperar [OMA92]. A tecnologia de objetos distribuídos possibilita o desenvolvimento de sistemas cliente/servidor mais flexíveis. Isto porque ela encapsula os dados e a lógica dos negócios com objetos que podem se deslocar para qualquer ponto da rede e até rodar em diferentes plataformas.

1.1 Objetivos

Um dos principais objetivos deste trabalho é implementar uma ferramenta de tradução das especificações GDMO e ASN.1 do modelo OSI para a especificação IDL da arquitetura CORBA, conforme a proposta desenvolvida pelo X/Open [OPS95]. Além de disponibilizar um mecanismo genérico para a construção de um protótipo utilizado pelo processo de tradução interativa. O implementador aqui citado, seria a pessoa, ou grupo de pessoas, responsável pela conversão dos sistemas atualmente em uso para as especificações em IDL/CORBA.

Entre os diversos estudos realizados para possibilitar a construção do tradutor, várias ferramentas auxiliares foram analisadas, como a biblioteca de implementação do HP OpenView, disponível no laboratório de redes do DAINF do CEFET-PR, como as funções *OVmdGetMod*, *OVmdGenNameTree* entre outras, mas fazer uma implementação própria eliminaria os problemas de bibliotecas específicas de uma determinada plataforma.

Outro motivo levado em consideração para a construção da ferramenta foi a possibilidade de oferecer um mecanismo diferenciado para acesso a suas estruturas internas, possibilitando com isso uma ferramenta altamente personalizável. E este recurso se mostrou extremamente útil no processo de exportação de informações auxiliares necessárias para a realização da tradução interativa.

1.2 Trabalhos Correlatos

Não se pode falar sobre interoperabilidade inter-domínios sem mencionar os estudos realizados por grupos como JIDM (*Joint Inter-Domain Management*) e NMF (*Network Management Forum*). Estes grupos debatem constantemente várias questões relacionadas ao problema de como interoperar domínios, ou seja, como realizar uma interação entre sistemas de gerência heterogêneos.

Quando se fala em gerência inter-domínios, segundo a definição apresentada por documentos JIDM [OPS95], está se falando na gerência da interação entre sistemas de gerenciamento distintos. Pode-se citar alguns domínios de gerência dentre os mais conhecidos como CMIP e SNMP. O objetivo principal destes grupos de estudo, é apresentar propostas para possibilitar à um certo domínio a gerência de um ou vários domínios diferentes. Mais recentemente os estudos se concentraram na possibilidade da utilização da arquitetura CORBA para suporte ao desenvolvimento das novas ferramentas de gerências, principalmente pelo desenvolvimento através de objetos distribuídos.

Segundo o JIDM [OPS95], existem basicamente, duas maneiras de tornar a interoperabilidade possível. Uma é elaborar um algoritmo de mapeamento das especificações GDMO/ASN.1/OSI para IDL/CORBA. A outra é prover uma forma para que as aplicações de cada domínio possam interagir e trocar informações de gerenciamento.

Um dos primeiros trabalhos mais abrangentes sobre o tema é apresentado em [SOU95c]. O autor apresenta um estudo sobre as possibilidades de harmonização dos domínios OSI e CORBA. Da mesma forma como nos demais trabalhos da área, a harmonização é realizada através da tradução das especificações GDMO para IDL e da interação entre as aplicações de gerência. No trabalho de Soukouti, são apresentadas três propostas. Primeiro são definidas as interações de OSI para CORBA. A segunda proposta abrange as interações de CORBA para OSI e a terceira, propõe uma definição de um sistema de gerência baseado em CORBA. Para realizar a interação entre domínios, o autor criou um elo de ligação que esconde as naturezas distintas CORBA e OSI, de um domínio para outro. Este elo chamado de passarela ou *gateway* faz as traduções em tempo de execução, dos serviços OSI para operações CORBA. No outro sentido,

isto é, de CORBA para OSI, ele propõe um conjunto de interfaces IDL onde cada uma corresponde a um dos sete serviços CMIS. A maior atenção é dada para o sentido de harmonização de OSI para CORBA, onde são descritos os algoritmos necessários para que isso seja possível.

Uma abordagem interessante proposta por Guy Genilloud [GEN96], faz com que as especificações GDMO, ASN.1 e IDL sejam armazenadas em uma base de dados genérica. A tradução das especificações, ocorre inicialmente por uma ferramenta automática, mas pode receber refinamentos realizados pelo implementador, como a seleção de nomes de identificadores. Este mecanismo é chamado pelo autor de tradução flexível (*flexible translation*). Esta forma permite um melhor resultado das especificações traduzidas, pois uma ferramenta totalmente automatizada pode introduzir informações adicionais não necessárias nos arquivos resultantes, bem como não possibilita ao implementador uma escolha de quais informações realmente deseja mapear. Outra característica deste processo, já mencionado anteriormente, é o armazenamento das especificações em uma base de dados, chamado pelo autor de *interface specification database system*, que se torna disponível para um agente-proxy, responsável pela tradução interativa. O acesso a este banco de dados de interfaces é realizado através de operações bem definidas, como *get_all_gdmo_package*, para recuperar as informações sobre as *packages* contidas em um determinado módulo GDMO, ou *get_gdmo_notification*, para obter as informações sobre as notificações definidas em um módulo. Nesta base de dados também estão contidas todas as informações sobre os módulos em ASN.1 e IDL transformados, e acessíveis através de um conjunto de operações similares.

O outro trabalho é o de Bela Ban [BBAN97], que propõe um modelo de programação uniforme que permite manipulação transparente de um conjunto de modelos de objeto sem depender de um modelo objeto em particular. Este modelo denominado GOM (*Generic Object Model*) permite que os sistemas desenvolvidos em CORBA, possam se integrar e fazer uso de funcionalidades de outros sistemas diferentes. Esta integração é realizada sem a necessidade de nova implementação destas funcionalidades em CORBA, reduzindo com isso tempo, custo e conhecimento. Este esquema evita que os sistemas sejam escritos novamente, além de oferecer uma integração imediata dos componentes destes sistemas em CORBA, que futuramente podem ser substituídos por componentes baseados totalmente em CORBA. O modelo GOM identifica novas classes em tempo de execução e possibilita a gerência de aplicações, oferecendo a vantagem de manipulá-las dinamicamente. O modelo também trabalha para a redução dos custos no desenvolvimento dos *small clients*. Estes clientes não contém nenhum conhecimento sobre as classes do sistema destino que manipulam. Somente quando um cliente é executado para

manipular outro sistema através do GOM, as classes ainda não conhecidas são carregadas dinamicamente. A definição de *small clients* pelo modelo GOM, é devido ao fato de que um agente não pode ser compilado com um grande número de classes, por causa das limitações de tamanho. É mais econômico em termos de velocidade e tamanho, enviar pequenos agentes através da rede do que grandes. Em [BBAN97] foi apresentado um trabalho bem mais abrangente e genérico, enfocando também o ambiente os serviços SNMP que aqui não foi abordado.

O problema da interoperabilidade pode ser resolvido de várias maneiras. Nossa concentração é sobre seus esforços na tradução de especificações GDMO/ASN.1 para IDL/CORBA, para possibilitar a interoperabilidade entre aplicações de domínios diferentes.

1.3 Apresentação do Trabalho

No segundo capítulo são apresentadas as características da especificação CORBA, mostrando como ela é considerada atualmente como um ótimo ambiente para o desenvolvimento de aplicações distribuídas. Especial atenção é dada à linguagem de definição de interfaces IDL.

No terceiro capítulo são relacionados os aspectos de gerência definidos pelo modelo OSI, em especial, as suas ferramentas de notação, como o *Guidelines for Definition of Managed Objects* - GDMO e a *Abstract Syntax Notation One* - ASN.1.

No quarto capítulo são mostradas as considerações feitas sobre os ambientes OSI e CORBA, relacionando quais pontos podem ser aproveitados e quais são os mais complexos para serem trabalhados.

No quinto capítulo é apresentada a forma de como poderá ser realizada a transformação das especificações GDMO/ASN.1 para IDL/CORBA, sendo feito um mapeamento completo de todas as funcionalidades existentes no ambiente OSI e que poderão ser descritas no ambiente CORBA.

No sexto capítulo é apresentada a forma como a ferramenta foi desenvolvida para a realização da conversão das especificações, sendo detalhado principalmente como possibilitar ao implementador dos mecanismos de gerência a forma de como acessar as informações das estruturas internas do tradutor de GDMO para IDL/CORBA.

No sétimo capítulo apresenta as considerações sobre este trabalho e as sugestões de trabalhos futuros.

CAPÍTULO 2

CORBA - COMMON OBJECT REQUEST BROKER ARCHITECTURE

2.1 Introdução

A especificação CORBA corresponde a uma coleção de serviços (interfaces e objetos), que providenciam as funções básicas para uso e implementação de objetos distribuídos. Estes serviços fornecem mecanismos para que objetos solicitem e recebam respostas, provendo interoperabilidade de aplicações em diferentes máquinas e ambientes distribuídos heterogêneos, e interconecta múltiplos sistemas de objetos sem complexidade. Dentro deste estudo será dado ênfase a última versão disponível da especificação da arquitetura CORBA [CORBA98], mas não esquecendo o avanço pelo qual a arquitetura passou, compreendendo as referências [CORBA91], [CORBA95] e [CORBA97].

2.2 Object Management Group (OMG)

O OMG é um consórcio de empresas, que busca a redução da complexidade do desenvolvimento de sistemas distribuídos, baixando custos e acelerando a introdução de novas aplicações de software. O OMG trabalha na elaboração e introdução de uma arquitetura para ambientes distribuídos que irá suportar detalhes da especificação de interface. Estas especificações de interfaces irão direcionar a indústria de software para interoperar e reusar componentes de softwares portáteis baseados num padrão de interfaces orientadas a objetos. Um dos principais objetivos é a maximização da portabilidade, reusabilidade e interoperabilidade de software, provendo uma arquitetura de referência com termos e definições sobre os quais todas as especificações são baseadas. O OMG define o paradigma de gerenciamento de objetos como a habilidade de encapsular dados e métodos para o desenvolvimento de software. Isto pode resultar em um desenvolvimento mais rápido de aplicações, fácil manutenção, redução da complexidade dos programas e componentes reusáveis, além de que, um sistema orientado a objetos possui a

característica de crescer em funcionalidade através da extensão de componentes existentes e com a adição de novos objetos ao sistema.

O OMG trabalha para que um dia os usuários iniciem suas aplicações sem se preocuparem com qualquer detalhe em relação aos objetos envolvidos, ou seja, toda aplicação será executada de forma transparente [OMA92].

2.3 Modelo Objeto OMG

Este modelo define uma *semântica comum de objetos* para especificação de características *externamente visíveis* de objetos, dentro de um padrão e independente do modo de implementação. As características externamente visíveis dos objetos são descritas por uma interface que consiste de assinaturas de operações. A visão externa, tanto do comportamento do objeto como do seu estado, são modelados em termos das *assinaturas* das operações.

As operações descrevem uma ação que pode ser aplicada de acordo com parâmetros e toda operação possui um *assinatura* que inclui o nome da operação, um conjunto de parâmetros e uma lista de parâmetros de retorno. Os objetos são agrupados em *tipos* e objetos individuais são instâncias de seus respectivos tipos. Detalhes de implementação são encapsulados pelas operações e nunca diretamente expostos na interface externa. O OMG define um núcleo que é um conjunto de requisitos que devem ser suportados em qualquer sistema, os quais estão de acordo com o *Modelo Objeto* padrão. Dentro do núcleo cada objeto possui uma *identidade* e um *identificador*.

Este Modelo Objeto define o conceito de *componentes* que são extensões compatíveis com o seu núcleo, mas não necessitam ser suportados por todos os sistemas.

O Modelo Objeto OMG também define um mecanismo, chamado *profiles*, que são grupos de componentes que se combinam, para servir como um conjunto útil de extensões em domínios de tecnologias particulares. Dentre alguns exemplos de profiles estão os Intermediários de Requisições de Objetos (*Object Request Broker* - ORB), Banco de Dados de Objetos, Requisitos e Análises, e Interface do Usuário [OMA92].

2.4 O Modelo Objeto CORBA

O modelo objeto CORBA, que é um refinamento do Modelo Objeto OMG, fornece uma apresentação organizada de conceitos sobre objetos e terminologia. Ele define um modelo parcial para computação que engloba as características chave de objetos. O Modelo Objeto OMG é abstrato pois não é realizado por nenhuma tecnologia em particular. O modelo objeto CORBA é

um modelo objeto *concreto*, e que difere do modelo abstrato de várias formas, as quais serão vistas a seguir. [CORBA98]

Um sistema de objetos é uma coleção de objetos que isolam os requisitores de serviços - clientes - dos seus fornecedores, através de uma interface de encapsulamento bem definida. Em particular, os clientes são isolados dos aspectos da implementação, como representação dos dados e código executável. O modelo objeto CORBA descreve inicialmente conceitos que são significativos para os clientes, incluindo criação e identidade de objetos, requisições, operações, tipos e assinaturas.

O modelo objeto CORBA é mais específico e prescritivo na definição de conceitos significativos para clientes. A discussão de implementação de objetos é mais sugestiva, com a intenção de permitir liberdade máxima para que diferentes tecnologias de objetos forneçam diferentes modos de implementar objetos.

O modelo objeto CORBA é um exemplo de um modelo clássico, onde um cliente envia uma mensagem para um objeto. Conceitualmente, o objeto interpreta a mensagem para decidir que serviço realizar. No modelo clássico, uma mensagem identifica um objeto com nenhum ou mais parâmetros. Um parâmetro inicial é necessário, para identificar a operação a ser realizada. A interpretação da mensagem pelo objeto envolve a seleção de um método baseado na operação específica, que pode ser realizada tanto pelo objeto quanto pelo ORB.

Clientes solicitam serviços através de *requisições*. Uma requisição é um evento que ocorre em determinado instante de tempo. A informação associada a uma requisição consiste de uma operação, um objeto destino, com nenhum ou mais parâmetros, e, opcionalmente, um contexto de requisição (configuração da máquina, como por exemplo o *environment* do UNIX).

Uma referência a um objeto é um identificador de um objeto que confiavelmente denota um determinado objeto. Especificamente, uma referência a um objeto identificará o mesmo cada vez que for usada em uma requisição (sujeita a certos limites de tempo e espaço). Um objeto pode ser denotado por várias referências distintas.

Uma *requisição* pode ter parâmetros que são utilizados para transferir dados para o objeto destino. A consequência de se realizar um serviço é devolver ao cliente os resultados definidos para a requisição, quando estes existirem. Os parâmetros da requisição são identificados pela posição. Um parâmetro pode ser de entrada (*in*), saída (*out*) ou entrada/saída (*in/out*). Uma requisição também pode retornar um valor resultado, além dos parâmetros de saída.

Do ponto de vista do cliente, não há nenhum mecanismo especial para a criação ou destruição de um objeto, pois estes são criados e destruídos como consequência da emissão de

requisições. O resultado da criação de um objeto é revelado ao cliente na forma de uma referência ao novo objeto.

Uma *interface*, descrita em IDL-*Interface Definition Language*, é o meio pelo qual uma determinada implementação de objeto diz a seus clientes potenciais quais operações estão disponíveis e como devem ser invocadas. É uma descrição de um conjunto de possíveis operações que um cliente pode requisitar de um objeto. Um objeto satisfaz uma interface se ele pode ser especificado como o *objeto destino* em todas as requisições potenciais descritas por esta interface.

Uma *operação* é uma entidade que denota um serviço que pode ser requisitado e é identificada por um *identificador de operação*. Uma operação possui uma *assinatura* que descreve os parâmetros válidos da requisição e resultados retornados.

Uma *assinatura* consiste de uma especificação dos parâmetros necessários nas requisições para aquela operação, uma especificação do resultado da operação, uma especificação das exceções que podem ser causadas por uma requisição da operação e os tipos dos parâmetros que as acompanham, uma especificação de informação contextual adicional que possa afetar a requisição, e uma indicação da semântica de execução que o cliente deve esperar de uma requisição da operação.

As *operações* são potencialmente genéricas, ou seja, uma mesma operação pode ser requisitada em objetos com diferentes implementações, possivelmente resultando em comportamentos diferentes. Esta generalidade é obtida através da herança de interface em IDL e da inexistência de vínculo de implementação e especificação de interface.

Um parâmetro é caracterizado por seu modo e seu tipo. O modo indica se o valor deve ser passado do cliente para o servidor (*in*), do servidor para o cliente (*out*) ou ambos (*inout*). O tipo restringe os valores que podem ser passados na direção indicada pelo modo. O resultado de retorno é um parâmetro *out*.

Uma *exceção* indica que uma requisição não foi realizada de modo apropriado e pode ser acompanhada por informações adicionais específicas.

A *semântica de execução* é associada com a operação. Isto evita que um cliente e uma implementação de objeto assumam semânticas de execução diferentes. A implementação de um objeto consiste na execução de atividades computacionais necessárias para efetuar o comportamento dos serviços solicitados.

Um *serviço* requisitado é realizado em um sistema computacional através da execução de código que opera sobre algum dado. O dado representa um componente do estado do sistema computacional. O código realiza o serviço requisitado, que pode alterar o estado do sistema. O código executado para realizar um serviço é denominado *método* e a sua execução é denominada ativação do método.

Um sistema baseado em objetos deve fornecer mecanismos para realizar o comportamento de requisições. Estes incluem definições para o estado do objeto, definições de métodos e definições de como a infra-estrutura do objeto seleciona o método a ser executado e as porções do estado do objeto que podem ser acessadas pelos métodos.

2.5 A Especificação CORBA

A especificação CORBA (*Common Object Request Broker Architecture*) foi definida para permitir a integração de uma grande variedade de sistemas com paradigmas de orientação a objetos. Não sendo restrita ao tipo de linguagem ou ambiente operacional. [CORBA98]

A Figura 1 mostra uma requisição sendo enviada por um cliente para uma implementação de objeto. O cliente é uma entidade que deseja realizar uma operação sobre o objeto, e a implementação de objeto compreende o código e os dados que de fato implementam o objeto.

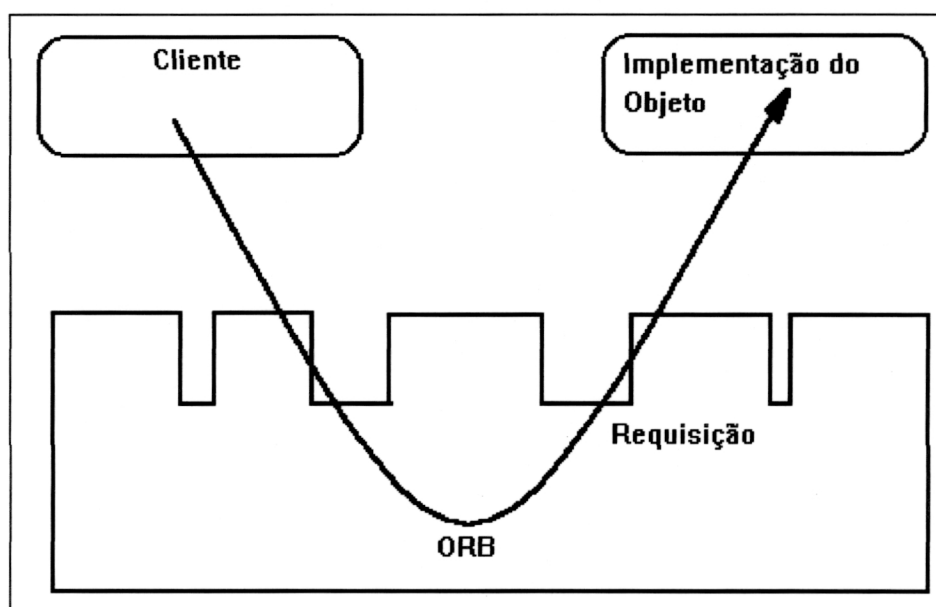


Figura 1 - Envio de requisições através de um ORB.

O ORB (*Object Request Broker*) é responsável por todos os mecanismos necessários para encontrar a implementação do objeto para a requisição, preparar a implementação para receber a requisição e transferir os dados, completando a requisição. A interface que o cliente vê é completamente independente de onde o objeto está localizado, e em qual linguagem de programação está implementado, isto representa a transparência do objeto.

Para fazer uma requisição, o cliente pode utilizar a interface de Invocação Dinâmica (a mesma é independente da interface do objeto destino) ou de um *stub* de objeto específico. Um *stub* é a rotina que organiza o envio de parâmetros para o ORB, isto é, efetua o *marshalling* e *unmarshalling* (codificação e decodificação dos parâmetros das operações em um formato adequado para a comunicação). A implementação do objeto recebe uma requisição como uma chamada através do *skeleton* gerado pela IDL, e pode chamar o adaptador de objeto ou o ORB enquanto processa uma requisição. Os diversos modos de interação dos clientes e implementações com o ORB são ilustrados na Figura 2.

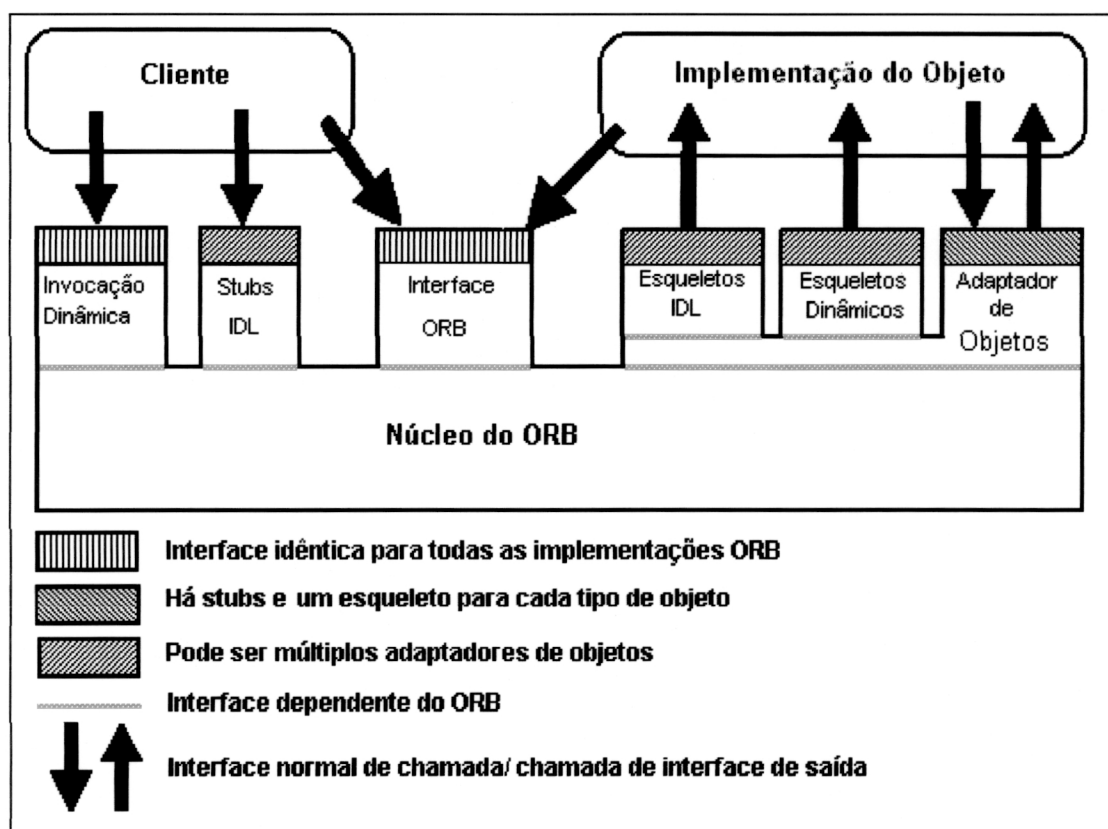


Figura 2 - Estrutura de um ORB.

As interfaces de objetos podem ser definidas estaticamente (em tempo de compilação) através de uma linguagem de definição de interface - IDL, que permite a definição do tipo dos objetos de acordo com as operações a serem realizadas e seus parâmetros. Além disso, as interfaces podem ser adicionadas em um serviço denominado **Repositório de Interfaces**. Este serviço representa os componentes de uma interface como objetos, permitindo acesso em tempo de execução, quando um cliente não conhece a interface definida para o objeto.

O cliente executa uma requisição fazendo acesso a uma referência de objeto e inicia a requisição chamando rotinas do *stub* específicas ao objeto ou construindo-a dinamicamente. As rotinas dos *stubs* fazem chamadas para o ORB utilizando as interfaces privadas, para um núcleo particular do ORB. O ORB localiza o código de implementação apropriado, transmite os

parâmetros e transfere o controle para a implementação do objeto através de um *skeleton*. Na definição CORBA, um *skeleton* é equivalente ao *stub* no lado do servidor. IDL. Os *skeletons* são específicos da interface e do *adaptador de objetos* que estão sendo representados. Durante a realização da requisição, a implementação do objeto pode obter alguns serviços do ORB através do adaptador de objetos. Quando a requisição está completa, o controle e valores de saída retornam para o cliente.

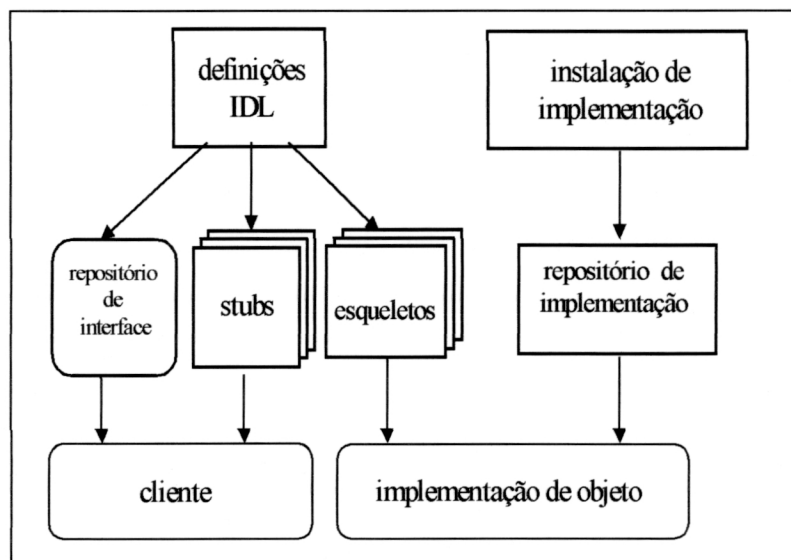


Figura 3 - Repositórios de Interface e Implementação.

Na figura 3 vemos como a interface e informações de implementação são colocadas à disposição do cliente e da implementação do objeto. A interface é definida na IDL ou no Repositório de Interfaces. A definição é utilizada para gerar os *stubs* do cliente e os *skeletons* da implementação do objeto. A informação sobre a implementação do objeto é fornecida no momento da instalação e armazenada no Repositório de Implementação para ser utilizada durante a entrega de requisições.

De acordo com esta arquitetura, o ORB não necessita ser implementado como um único componente, mas é definido por suas interfaces. Qualquer implementação de ORB que forneça a interface apropriada é aceitável. A interface é organizada em três categorias: operações que são as mesmas para todas as implementações de ORBs, operações que são específicas a determinados tipos de objetos, e operações que são específicas para determinados estilos de implementação de objetos.

Diferentes ORBs podem fazer diferentes escolhas de projeto, e, juntamente com compiladores IDL, repositórios e diversos adaptadores de objetos, fornecem um conjunto de serviços para clientes e implementações de objetos que possuem diferentes propriedades e qualidades.

Podem haver diversas implementações de ORBs com diferentes representações de referências a objeto e diferentes meios de realizar invocações. Pode ser possível que um cliente tenha acesso simultaneamente a duas referências a objeto gerenciadas por diferentes implementações de ORBs. Quando dois ORBs desejam trabalhar juntos, eles devem poder distinguir suas referências a objeto. Isto não é de responsabilidade do cliente.

O *núcleo do ORB* é a parte que fornece a representação básica de objetos e comunicação de requisições. A arquitetura CORBA é projetada para suportar diferentes mecanismos de objetos, e o faz estruturando o ORB com componentes acima do núcleo, que fornecem interfaces que mascaram as diferenças entre núcleos de ORBs.

Uma implementação de objetos fornece a definição de dados para a instância do objeto e código para seus métodos, sendo que, freqüentemente, ela utilizará outros objetos ou software adicional para implementar o comportamento do objeto.

Uma grande variedade de implementações pode ser suportada, incluindo servidores separados, bibliotecas, um programa por método, uma aplicação encapsulada ou um banco de dados orientado a objeto. Através do uso de adaptadores de objeto, é possível suportar virtualmente qualquer estilo de implementação de objetos.

Uma referência a objeto é a informação necessária para especificar um objeto dentro de um ORB. Duas implementações de ORBs podem diferir em suas escolhas de representação de objetos. A referência a um objeto ligada a um cliente só é válida durante o tempo de vida (execução) do cliente.

Diferentes linguagens de programação, orientadas a objeto ou não, podem preferir acessar a CORBA de diferentes modos. Para linguagens orientadas a objeto, é desejável ver os objetos CORBA como objetos de linguagens de programação. Um mapeamento de uma determinada linguagem para CORBA deve ser o mesmo para todas as implementações de ORBs. O mapeamento inclui definição dos tipos de dados específicos da linguagem e interfaces de procedimentos para acessar objetos através do ORB, definindo a estrutura da interface do *stub* do cliente, a interface de invocação dinâmica, o *skeleton* da implementação, os adaptadores de objetos e a interface direta do ORB.

Para um mapeamento numa determinada linguagem, haverá uma interface de programação para os *stubs* de cada tipo de interface. Os *stubs* fazem chamadas para o ORB através de interfaces particulares ao núcleo do ORB. Se mais de um ORB está disponível, deve haver mais de um *stub*, correspondendo a cada um dos ORBs.

Também está disponível uma interface que permite a construção dinâmica de invocações de objetos, ou seja, ao invés de chamar uma rotina de *stub* que é específica para uma determinada operação sobre um determinado objeto, um cliente pode especificar o objeto a ser invocado, a operação a ser realizada, e o conjunto de parâmetros para a operação através de uma sequência de chamadas em tempo de execução. O código do cliente deve fornecer informação suficiente sobre a operação a ser realizada e os tipos dos parâmetros, obtendo-a possivelmente de um repositório de interfaces, ou de alguma outra fonte em tempo de execução.

Um *adaptador de objeto* é a principal forma pela qual uma implementação de objeto acessa serviços fornecidos pelo ORB. Os serviços fornecidos pelo ORB através de um adaptador de objeto geralmente incluem: geração e interpretação de referências a objetos, segurança de interações, ativação e desativação de implementações e objetos, mapeamento de referências de objetos para implementações, e registro de implementações.

Existe um objeto especial denominado *ORB*, que deve ser implementado diretamente pelo *Broker*, sendo a mesma para todos os ORBs e não dependendo da interface do objeto nem do adaptador de objeto. Já que a maior parte da funcionalidade do ORB é fornecida pelo adaptador de objetos, *stubs*, *skeleton*, ou invocação dinâmica, existem apenas poucas operações comuns definidas na interface ORB.

O *Repositório de Interfaces* é um serviço que fornece objetos persistentes que representam a informação descrita em IDL, disponíveis em tempo de execução. Através das informações existentes em um Repositório de Interfaces, é possível para um programa encontrar um objeto cuja interface não era conhecida quando o programa foi compilado, possibilitando ainda determinar, em tempo de execução, quais operações do objeto são válidas e invocá-las.

O *Repositório de Implementação* contém informações que permitem ao ORB localizar e ativar implementações de objetos. Embora muitas informações deste repositório sejam específicas para um ORB ou para um ambiente operacional, o Repositório de Implementação é o local padrão para armazenamento destas informações. Normalmente, a instalação de implementações e controle de políticas relacionadas com a ativação e execução dos objetos é feita através de operações sobre o Repositório de Implementações.

2.6 Interface Definition Language – IDL

A *Interface Definition Language* – IDL - é uma linguagem formal padronizada pela OMG. A IDL é utilizada para definir as interfaces de objetos CORBA. Ela define a visibilidade funcional externa de um objeto CORBA. A informação definida em uma interface IDL possui todas as informações necessárias para o desenvolvimento de um cliente de um objeto CORBA. A

IDL é uma linguagem declarativa abstrata, não implica em uma representação concreta dos dados em diferentes sistemas, e também não define como um objeto é implementado. Uma definição de interface especifica as operações que um objeto é capaz de executar, os parâmetros de entrada e saída necessários, e alguma exceção que possa ser gerada. Pode-se dizer que a interface é como um contrato entre um objeto CORBA e seu cliente.

A IDL é a linguagem comum de todas as implementações CORBA. Uma das mais importantes funções providas por um ORB é sua *language binding*. Uma *language binding* é a tradução dos conceitos IDL em conceitos de uma linguagem de programação específica. Atualmente estão definidas as *language binding* de IDL para C, C++, Smalltalk, COBOL e Java [CORBA98][JIDL97].

Apesar da IDL ter sido originalmente desenvolvida para habilitar o acesso remoto aos serviços dos objetos, ela é muitas vezes usada para especificar serviços de objetos que muitas vezes não são acessados remotamente.

Uma especificação IDL consiste de uma ou mais definições de módulos, interfaces, exceções, constantes e tipos. Mas a parte mais importante de uma especificação em IDL são suas interfaces. As interfaces e os tipos relacionados (constantes, especificações de exceções) são organizadas em módulos. Um módulo usualmente agrupa diversos serviços relacionados.

Para qualquer consulta adicional, esta relacionado no anexo 1 a BNF da linguagem IDL definida pela OMG.

2.6.1 Módulos

Um módulo constitui um escopo de definições, os identificadores de especificações são únicos dentro de um módulo. Fora do módulo, um identificador em conjunção com um nome de módulo são usados para identificar a especificação.

```
module X3100 {
    ...
};
```

Figura 4 - Definição de um módulo em IDL CORBA

Uma declaração de módulo pode incluir: declarações de tipos, declarações de constantes, declarações de exceções, declarações de interfaces e outros módulos.

2.6.2 Interfaces

Uma interface especifica uma classe de objetos CORBA. Ela especifica as instâncias de serviços que uma classe possui, seus atributos e as exceções que podem gerar. A Figura 5 apresenta um exemplo de como definir uma interface em IDL CORBA. Quando uma interface é

derivada de outra interface ela especifica uma subclasse especializada da classe especificada pela interface original, uma subclasse herda a interface de sua superclasse. A interface de uma subclasse pode estender e sobrepor partes da interface de sua superclasse. Uma interface pode ser derivada de mais de uma interface. Isto é definido como herança múltipla

```
interface top {
    ...
};
```

Figura 5 - Definição de uma interface em IDL CORBA.

A Figura 6 mostra a herança sendo definida em uma interface IDL CORBA. Neste exemplo a interface do objeto `equipment` herda do objeto `X0721::top` todas as características já existentes nela, acrescentando mais algumas características particulares.

```
interface equipment : X0721::top {
    ...
};
```

Figura 6 - Definição de herança de interface em IDL CORBA.

Uma declaração de interface pode conter: declaração de atributos, declaração de operações, declarações de tipos, declarações de constantes e declarações de exceções.

2.6.3 Operações

O conjunto de declarações de operações em uma interface define os serviços que um objeto é capaz de executar. Uma operação define os parâmetros de entrada e saída necessários para um certo serviço. O tipo de uma operação é o tipo do resultado de sua operação (execução).

Uma operação tem dois modos de execução: *at-most-once* e *best-effort*. Uma operação é executada *at-most-once* por padrão. Para especificar uma operação *best-effort* a palavra chave `oneway` deve ser anexada. Uma operação *at-most-once* pode ter sido executada *at-most-once* quando uma exceção é criada. Se entretanto, nenhuma exceção é executada, então a operação é executada exatamente uma vez. Uma operação `oneway`, ou seja *best-effort*, não pode ser invocada mais que uma vez, mas ela não garante que seja executada com sucesso. Uma operação `oneway` não permite nenhuma indicação de invocação com sucesso, então deve ser especificado um resultado do tipo `void`, como também não é possível especificar parâmetros de saída (do tipo `out`) e não pode criar qualquer exceção.

```
X0721Att::NameBindingType nameBinding_get()
    raises ( CMIP_ATTRIBUTE_ERRORS );
```

Figura 7 - Exemplo de operação em IDL CORBA.

2.6.4 Atributos

Como qualquer objeto, os objetos CORBA podem ter atributos. Os atributos contêm informações sobre o estado de um objeto. Os atributos são declarados na interface de um objeto.

A declaração dos atributos é equivalente a declaração de duas funções complementares: uma para colocar um valor em um atributo e outra para recuperar seu valor. A declaração de um atributo é bem diferente da declaração de uma operação, pois um atributo não pode emitir exceções. Na declaração de um atributo `readonly` é omitida a declaração de função para colocar um valor no atributo.

```
interface Sommelier {  
    readonly attribute WineList wineList;  
    ...  
};
```

Figura 8 - Declaração de um atributo em IDL CORBA.

2.6.5 Declarações de Tipos

A linguagem IDL possui dois mecanismos para a declaração de tipos. O primeiro simplesmente atribui um novo nome a um tipo já existente, com a utilização da palavra chave `typedef`.

```
typedef dummy CMIPHeader;
```

Figura 9 - Declaração de um novo tipo em IDL CORBA.

Na declaração apresentada na Figura 9 verifica-se que o novo tipo criado difere apenas semanticamente do tipo base.

A segunda forma de criação de tipo envolve a construção de um novo tipo usando os tipos já existentes como base. A linguagem IDL define dois grupos de tipos: os simples e os construtores.

2.6.6 Tipos Simples

Os tipos simples são as unidades mais simples que podem ser utilizadas para a declaração de novos tipos. Também não é possível decompor este tipo em componentes menores. Os tipos simples em IDL são apresentados na Tabela 1.

Tipo	Descrição
float	Ponto flutuante de precisão simples
double	Ponto flutuante de precisão dupla
long	Inteiro longo, no intervalo de -2^{31} até $2^{31}-1$
short	Inteiro curto, no intervalo de -2^{15} até $2^{15}-1$
unsigned long	Inteiro longo, sem sinal, no intervalo de 0 até $2^{32}-1$
unsigned short	Inteiro curto, sem sinal, no intervalo de 0 até $2^{16}-1$
boolean	Valores TRUE ou FALSE
octet	Cadeia com 8 <i>bits</i>
char	Caracter com 8 <i>bits</i>
enum	Lista ordenada de constantes nomeadas

Tabela 1 - Tipos simples em IDL CORBA.

Tipo Enumerado

O tipo enumerado é uma lista de valores específicos. Os valores nesta lista são declarados como constantes. Eles são ordenados na ordem na qual são listados na declaração de seu tipo enumerado. A Figura 10 apresenta um exemplo da utilização de tipo enumerado.

```
enum AdministrativeStateType {
    locked,
    unlocked,
    shuttingDown
};
```

Figura 10 - Exemplo de tipo enumerado em IDL CORBA.

2.6.7 Tipos Construtores

Os tipos construtores são utilizados para combinar tipos básicos para criar um novo tipo. Tipos construídos são compostos por nenhum ou mais componentes. Existem quatro tipos construtores em IDL CORBA, são eles: estruturas, seqüências, união discriminada e matrizes.

Estruturas

Uma estrutura define uma lista de um ou mais campos de valores de diferentes tipos. Os elementos em uma estrutura consistem de um tipo e um nome associado. Na Figura 11 é apresentado um exemplo de estrutura.

```

struct ManagementExtensionItem {
    ASN1_ObjectIdentifier    objectIdentifier;
    ASN1_BOOLEAN             significance;
    any                      information;
};

```

Figura 11 - Declaração de uma estrutura em IDL CORBA.

Tipo *Sequence*

Um tipo *sequence* define uma lista ordenada de nenhum ou mais valores de um mesmo tipo. Uma *sequence* pode ser limitada, com um tamanho máximo, especificado na sua declaração, ou ilimitada, quando nenhum tamanho é especificado na sua declaração. A Figura 12 apresenta um exemplo de *sequence*.

```
typedef sequence<AttributeType> AttributeListType;
```

Figura 12 - Declaração de uma *sequence* em IDL CORBA.

Tipo *String*

Um tipo especial de *sequence* é o tipo *string*. Este tipo pode ser declarado como uma *sequence* de caracteres. O tipo *string* é muito usual, e pode possuir propriedades específicas, em diversas linguagens de programação. O principal uso do tipo *string* é poder diferenciar de outros tipos de *sequence*. A Figura 13 mostra um exemplo de declaração de um tipo *string* para guardar informações sobre um nome.

```
string <30> Nome;
```

Figura 13 - Declaração de uma *string* em IDL CORBA.

Unões

O tipo *union* define a sintaxe de valores que podem ter um número de tipos alternativos. A união discriminada lista os tipos alternativos com suas *tags* associadas. As *tags* são valores de um tipo em particular, que escolhe o resultado da *union*. O tipo da escolha é definido na declaração da *union* e é utilizada para distinguir entre as alternativas. Uma cláusula padrão define o tipo do valor quando nenhuma das *tags* é aplicada. Na Figura 14 é mostrado como criar um tipo *union*.

```

union AnyOpt switch ( boolean ) {
    case TRUE :    any    value;
    case FALSE :   string  value;
};

```

Figura 14 - Declaração de uma *union* em IDL CORBA.

Matrizes

O tipo *array* é usada para declarar uma coleção de valores indexados. Uma matriz pode ter uma ou mais dimensões. Na sua declaração deve estar explícita o tamanho de cada dimensão. A Figura 15 mostra a declaração de uma matriz de duas dimensões.

```
MenuCard [ 50 ] [ 2 ];
```

Figura 15 - Declaração de matriz em IDL CORBA.

2.6.8 Tipo any

Existe uma opção em IDL CORBA que permite a declaração de um tipo que não se refere a nenhum tipo em específico, mas sim a qualquer tipo desejado, o tipo *any*.

O tipo *any* é usado quando um valor qualquer pode assumir qualquer tipo. O tipo atual de um valor em tempo de execução é um dos tipos conhecidos naquele contexto em específico.

```
any newAttributeValue;
```

Figura 16 - Declaração do tipo *any* em IDL CORBA.

2.6.9 Constantes

Constantes são identificadores que representam valores de um determinado tipo. Eles podem ser declarados em qualquer lugar em uma especificação IDL CORBA. A Figura 17 apresenta como é possível realizar uma declaração de constante.

```
const constant_type identifier = constant
```

Figura 17 - Declaração de constante em IDL CORBA.

Todos os tipos simples e seus subtipos, com exceção dos tipos enumerados, são tipos válidos para declaração de um valor constante.

```
const ASN1_ObjectIdentifier smi2AttributeID = "2.9.3.2.7";
```

Figura 18 - Declaração de constante em IDL CORBA.

2.7 Considerações Finais

Neste capítulo foi apresentada a arquitetura CORBA, sendo dado enfoque ao seu modelo objeto, ao funcionamento básico e aos conceitos fundamentais para seu entendimento. Uma atenção especial foi destinada a linguagem IDL - *Interface Definition Language*, que é utilizada para a representação das interfaces dos objetos definidos em CORBA. Sua abordagem permite a real compreensão dos arquivos resultantes do mecanismo de tradução das especificações.

CAPÍTULO 3

ARQUITETURA DE GERENCIAMENTO OSI - OPEN SYSTEM INTERCONNECTION

3.1 Introdução

A necessidade de uma arquitetura de gerenciamento capaz de atender à enorme diversidade de elementos gerenciáveis existentes em uma rede e que tivesse características de integração, simplicidade, segurança e flexibilidade fez com que a ISO (*International Organization for Standardization*), apresentasse, em 1989, o esquema básico da Arquitetura de Gerenciamento de Rede OSI, em adição ao modelo de referência OSI [X-701].

Os sistemas de gerenciamento de redes devem prover mecanismos que permitam monitorar, controlar, contabilizar, organizar e planejar a utilização de recursos durante as comunicações realizadas dentro do ambiente OSI. Para isso, a atuação de tais sistemas deve, englobar cinco áreas funcionais: **gerenciamento de falhas**, inclui facilidades que permitem detectar, isolar e corrigir situações de erro relativas à operação dos recursos gerenciados dentro do ambiente OSI; **gerenciamento de contabilização**, permite determinar o índice de utilização de recursos dentro do ambiente OSI e associar a ele tarifas e custos; **gerenciamento de configuração**, possibilita obter informações, supervisionar e exercer controle sobre a configuração dos componentes de rede; **gerenciamento de desempenho**, inclui mecanismos de avaliação do comportamento dos recursos gerenciados e da eficiência das atividades de comunicação; e, por último, **gerenciamento de segurança**, que permite monitorar e gerenciar o emprego de mecanismos de segurança referentes ao controle de acesso, autenticação, criptografia, entre outros, utilizados dentro do ambiente OSI.

Outros quatro documentos definem a Estrutura da Informação de Gerenciamento (SMI-*Structure Management Information*) fornecendo um modelo de informação e guias para a definição dos objetos gerenciados. O serviço e o protocolo para a troca de informações de

gerenciamento são especificados, respectivamente, através do CMIS (*Common Management Information Service*) [X.710] e do CMIP (*Common Management Information Protocol*) [X.711].

A Arquitetura OSI de Gerenciamento define uma estrutura de gerenciamento, os componentes de gerenciamento, a estrutura da informação de gerenciamento, os serviços e o protocolo para a troca de informações de gerenciamento.

O ambiente de gerenciamento OSI é composto por gerentes, agentes e objetos gerenciados. Um gerente transmite operações de gerenciamento aos agentes a fim de obter informações atualizadas sobre os objetos gerenciados e controlá-los.

Dentro deste contexto, as padronizações da ISO especificam um sistema de gerenciamento integrado de rede baseado na definição de: objetos gerenciados que mapeam os recursos reais propriamente ditos, funções de gerenciamento de sistemas, e um protocolo de comunicação para troca de informações de gerenciamento. A Figura 19 mostra o modelo de gerenciamento OSI e seus principais componentes.

Um objeto gerenciado pode conter atributos, executar ações, emitir notificações e exibir comportamentos até um determinado limite, que está encapsulado pelo recurso. As classes de objetos gerenciados são formalmente especificados em GDMO (*Guidelines for Definition of Management Objects*) [X.722] e os tipos de dados são definidos em ASN.1 (*Abstract Syntax Notation One*) [X.208]. Nesta estrutura são definidos os dois aspectos principais na modelagem da informação de gerência: a herança e a nomeação.

Nesta figura, podem-se identificar dois sistemas abertos interconectados entre si, exercendo, respectivamente, os papéis de **agente** e **gerente**. O sistema agente tem como função coletar informações sobre os recursos gerenciados e armazená-las, adequadamente, na **Base de Informações de Gerenciamento (MIB - Management Information Base)** a ele associado. Além disso, executa operações solicitadas pelo sistema gerente e emite notificações ao mesmo referentes à ocorrência de quaisquer eventos ou falhas relativas aos recursos em questão. O sistema gerente, por sua vez, obtém informações atualizadas sobre os recursos gerenciados junto aos sistemas agentes, processa tais informações, podendo solicitar ou não a execução de operações de gerenciamento aos sistemas agentes a fim de controlar tais recursos. Um sistema pode exercer, concomitantemente, os papéis de gerente e agente. Vários sistemas agentes podem estar associados a um ou mais sistemas gerentes.

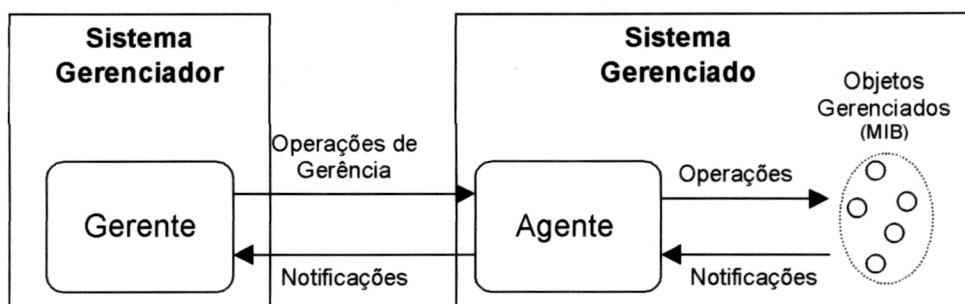


Figura 19 - Componentes do sistema de Gerenciamento OSI.

Com relação à MIB, ela armazena as informações transferidas ou modificadas pelo uso de protocolos de gerenciamento OSI. Essas informações podem ser fornecidas por agentes administrativos locais ou remotos, conforme mostrado na Figura 20. O modelo OSI, fornece uma interface via MIB, com cada uma das sete camadas. A interface específica para cada camada é obtida através das Entidades de Gerenciamento de Camadas (*LME - Layer Management Entity*). Cada LME concentra a funcionalidade de gerência da camada de sua responsabilidade. A integração destas entidades e a função de interface com o gerente é feita pela Entidade de Aplicação de Gerenciamento de Sistema (*SMAE - Systems Management Application Entity*). Esta entidade, providencia a interface entre as LMEs de um nó da rede com as LMEs de outro nó (LMEs pares), através do CMIP, que é o protocolo definido para a troca de informações de gerenciamento de sistemas.

Dentro desse contexto, pode-se dizer que a MIB contém uma coleção de objetos gerenciados que representam recursos reais, tais como um placa de rede, um algoritmo de roteamento em execução, uma conexão da camada de transporte, entre outros. Um objeto gerenciado é definido, então, em termos de seus atributos, das operações que pode efetuar, das notificações que pode emitir e de seu relacionamento com outros objetos gerenciados. Dentro da concepção de programação orientada a objetos adotada pelo modelo OSI, cada objeto é na realidade uma instância de uma classe de objetos, que possuem em comum os mesmos tipos de atributos, operações e notificações de gerenciamento.

Na camada de aplicação existe a Entidade de Aplicação de Gerenciamento (*SMAE*). As *SMAE* residentes nos sistemas agente e gerente comunicam-se entre si através do Protocolo de Informações de Gerenciamento Comum (*CMIP - Common Management Information Protocol*) usado para transportar as operações de gerenciamento, os resultados correspondentes e as notificações de gerenciamento, emitidas pelos próprios objetos gerenciados. Por último, o Processo de Aplicação de Gerenciamento de Sistemas (*SMAP - Systems Management Application Process*) corresponde ao software local de um dado sistema, que utiliza os serviços

de gerenciamento prestados pela camada de aplicação, com o objetivo de executar as funções de gerenciamento de rede dentro das áreas funcionais mencionadas anteriormente, coordenando suas atividades com as atividades de outros SMAP residentes em outros sistemas abertos.

Na camada de aplicação, uma SMAE é, logicamente, constituída de vários ASE (*Application Service Element*), dentre os quais se destacam: ACSE (*Association Control Service Element*), ROSE (*Remote Operations Service Element*), CMISE (*Common Management Information Service Element*) e SMASE (*System Management Application Service Element*). Os CMISE e SMASE são usados especificamente no caso de gerenciamento de redes. O SMASE implementa as funções básicas de gerenciamento correspondentes a cada uma das áreas funcionais referentes a gerenciamento de falhas, contabilização, configuração, desempenho e segurança, utilizando os serviços prestados pelo CMIS e por outros ASE. O CMISE, por sua vez, implementa funções básicas de gerenciamento comuns a todas as áreas funcionais, comunicando-se com os CMISE residentes em outros sistemas para a troca de operações (por exemplo, leitura e alteração de atributos de um objeto gerenciado, remoção e criação de um objeto gerenciado) e notificações de gerenciamento [BRISA94].

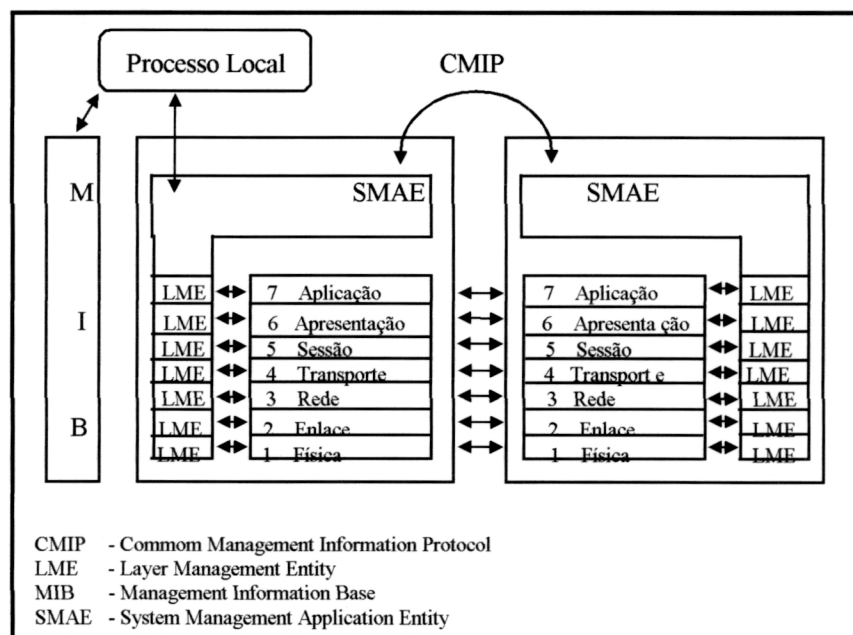


Figura 20 - O Modelo de Gerenciamento OSI.

Entre as principais características determinadas pela gerência OSI, estão um conjunto de normas para a definição das informações de gerenciamento. Estas definições são mostradas na Figura 21.

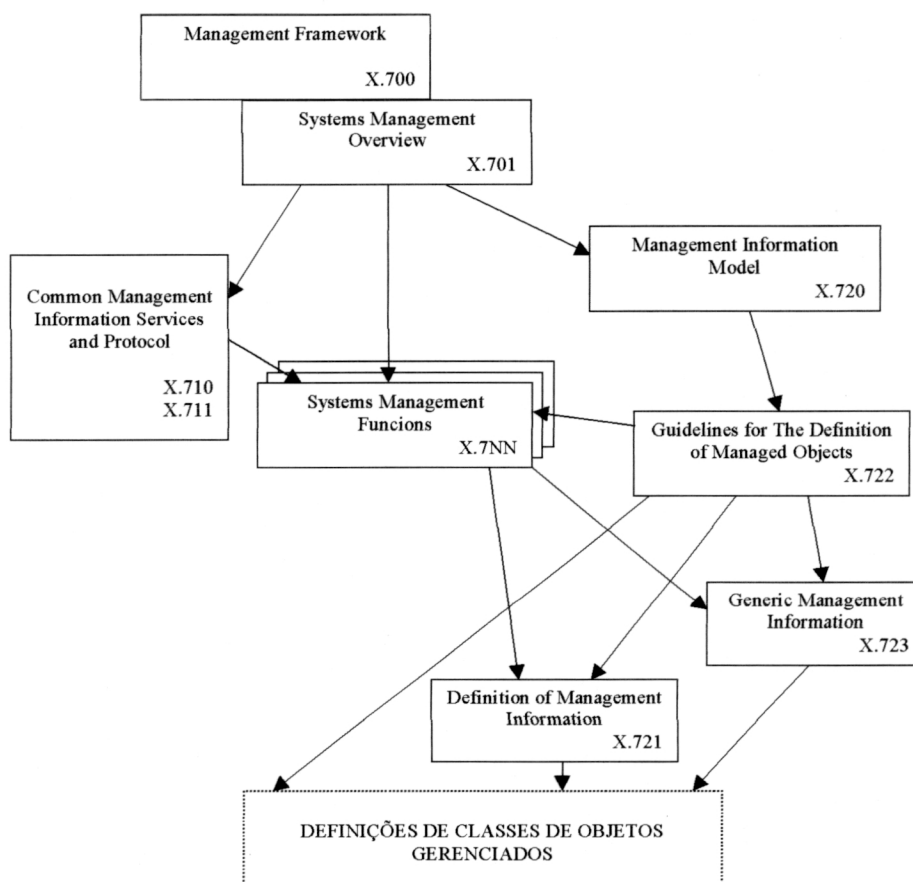


Figura 21 - Normas de Gerência OSI.

As normas de gerência OSI que nos interessam neste ponto, são as referentes ao modelo de informação, que caracterizam a SMI (*Structure of Management Information*), as normas relacionadas são:

- *Management Information Model (MIM)* X.720
- *Definition of Management Information (DMI)* X.721
- *Guidelines for the Definition of Management Objects (GDMO)* X.722
- *Generic Management Information (GMI)* X.723

Um dos principais objetivos da padronização é definir os conceitos, a sintaxe e as informações genéricas de gerência para a definição dos objetos gerenciados. Os conceitos são definidos pela norma X.720, a sintaxe é definida pela sintaxe X.722 e os conceitos de genéricos são definidos pela X.723. Estas especificações definem o conteúdo das mensagens transmitidas pelas interfaces de gerência. O principal conceito de um objeto gerenciado corresponde às visões abstratas adicionadas a um ou mais recursos de um sistema gerenciado.

3.2 Guidelines for Definition of Managed Objects - GDMO

O Guia para Definição de Objetos Gerenciados (*Guidelines for the Definition of Managed Object* - GDMO) [X.722] é o padrão internacional que define as notações usadas para especificar classes de objetos gerenciados, que permitem o gerenciamento de recursos em um ambiente OSI. O padrão também fornece a definição do gerenciamento de objetos com as informações auxiliares e um guia para a definição dos objetos gerenciados.

O GDMO, fornece a ligação entre o conceito de modelamento abstrato contido no **Modelo de Gerenciamento de Informações** e os requisitos concretos para a especificação particular do gerenciador das classes de objetos. O GDMO também inclui definições das sintaxes e semânticas da notação que o gerente de objetos deve usar quando especificar a gerência das classes de objetos e seus componentes. A intenção do padrão é fornecer ao gerente de objetos informações auxiliares e ferramentas notacionais que permitirão uma funcionalidade maior no gerenciamento dos recursos.

3.2.1 Princípios Gerais

Na definição de classes de objetos gerenciados, é relativamente fácil se envolver no processo e se perder do objetivo, que é fornecer um conjunto de ferramentas de gerenciamento para gerenciar um recurso particular. Deste modo, salientamos a importância de manter uma visão clara dos requisitos principais, pelo qual as classes de objetos gerenciados estão sendo desenvolvidas. Parte do processo envolve a aplicação de mecanismos estruturados (subclasses, heranças múltiplas, pacotes, conteúdos e grupos de atributos) de maneira apropriada. Escolhas certas nas técnicas estruturadas, podem reduzir a complexidade do processo de definição por permitir o reuso de especificações em diferentes contextos.

É importante manter o nível de complexidade da funcionalidade de gerência dentro de um sistema gerenciado, consistente com a complexidade do recurso sendo gerenciado. Se o próprio recurso é simples, mantendo a funcionalidade da gerência simples, minimizará o impacto do gerenciamento sobre toda a complexidade dos produtos resultantes. Isto pode, entretanto, ser o caso que as técnicas de gerenciamento complexas e os procedimentos são requisitados a fim de gerenciar mecanismos, nos quais a funcionalidade da gerência é simples; em cada caso, o peso de qualquer complexidade adicional envolvida no gerenciamento pode ser suportado por um sistema de gerenciamento.

3.2.2 Registro

O GDMO descreve a estrutura da árvore de registro que foi adotada nos padrões de gerência OSI por alocar globalmente identificadores únicos (identificadores de objeto).

Conforme será mostrado no tópico sobre ASN.1, os valores da árvore de registro servem para identificar unicamente uma *template* em particular.

O valor do identificador de objeto consiste de uma sucessão de valores de inteiros, conhecida como arcos, que definem a estrutura da árvore de registro. O primeiro arco define o ramo inicial da árvore, e identifica a autoridade de registro responsável para alocar o segundo arco, e assim por diante para arcos subseqüentes. São definidos três arcos de primeiro nível: um para uso da ISO, um para uso da CCITT e um para uso conjunto da ISO-CCITT.

O arco de terceiro nível é usado para identificar grupos de padrões; no momento, são alocados quatro arcos a este nível e correspondem à *Systems Management Overview*, o *Common Management Information Protocol* (CMIP), *Systems Management Function* e *Structure of Management Information* (SMI).

Os arcos de quarto nível identificam padrões particulares do grupo em questão; a distribuição de arcos debaixo deste nível é então a responsabilidade do padrão individual.

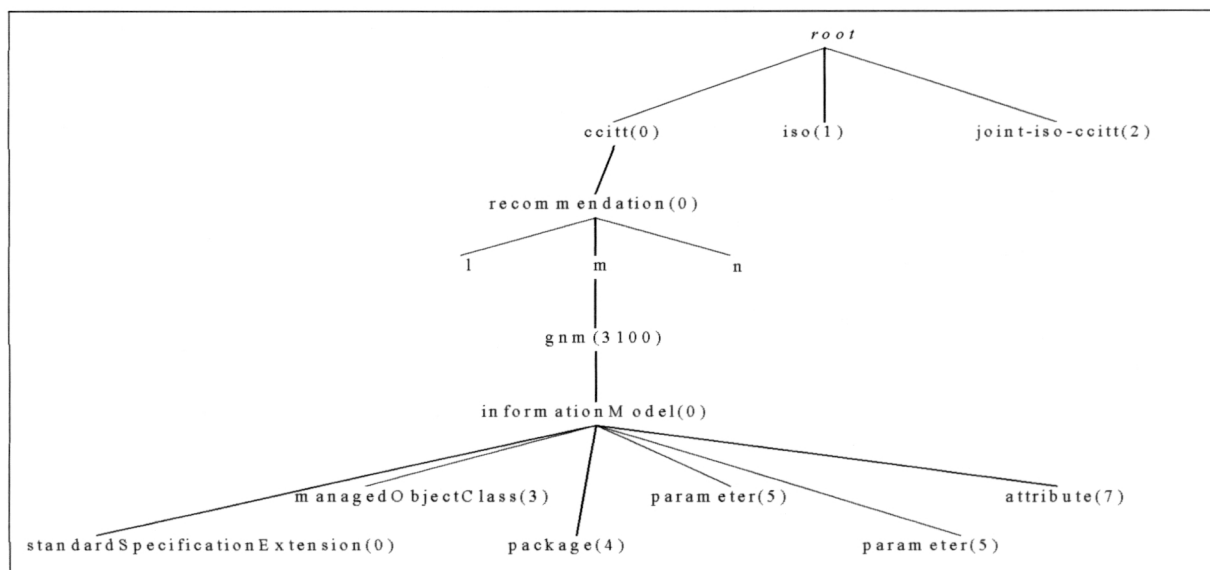
No caso da SMI um esquema comum existe para a distribuição dos arcos de quinto nível; estes valores são usados para identificar categorias de objetos registrados, como os módulos da ASN.1, classificando os objetos gerenciados, e os tipos de atributos utilizados. São usados os arcos de sexto nível para identificar instâncias daquela categoria de um objeto em particular. Assim, um valor de identificador de objeto alocado em GDMO à 49ª definição de tipo de atributo teria arcos alocados como segue:

{ joint-iso-ccitt ms(9) smi(3) part4(4) attribute(7) type-49(49) }

Abaixo é representado o registro para uso da norma M.3100.

```
m3100InformationModel OBJECT IDENTIFIER ::= {ccitt recommendation m gnm(3100) informationModel(0)}
m3100standardSpecificExtension OBJECT IDENTIFIER ::= {m3100InformationModel standardSpecificExtension(0)}
m3100ObjectClass OBJECT IDENTIFIER ::= {m3100InformationModel managedObjectClass(3)}
m3100Package OBJECT IDENTIFIER ::= {m3100InformationModel package(4)}
m3100Parameter OBJECT IDENTIFIER ::= {m3100InformationModel parameter(5)}
m3100Attribute OBJECT IDENTIFIER ::= {m3100InformationModel attribute(7)}
m3100NameBinding OBJECT IDENTIFIER ::= {m3100InformationModel nameBinding(6)}
m3100Action OBJECT IDENTIFIER ::= {m3100InformationModel action(9)}
m3100Notification OBJECT IDENTIFIER ::= {m3100InformationModel notification(10)}
```

Esta informação é representada pela seguinte árvore de registro:



3.2.3 Nomeação

As características dos recursos gerenciados são descritos em um modelo de informações usando definições de classes de objetos gerenciados. Para cada definição de objeto, instâncias são criadas para corresponder ao recurso sendo gerenciado. Quando múltiplas instâncias de classes são criadas, identificadores distintos de objetos gerenciados são necessários. O escopo dentro do qual um nome de objeto gerenciado é único pode variar de global, referente a todo o ambiente, ou relativo, referente ao sistema responsável pelo gerenciamento do recurso.

Um nome de objeto gerenciado é construído de várias partes, a partir do nível mais alto da árvore de nomeação. O nome especificado no processo de ligação de um objeto subordinado à outro, fornece o *Relative Distinguished Name* - RDN de um objeto gerenciado, como o CircuitPackID da Figura 22. Quando é necessário definir um objeto específico, dentre tantos outros é necessário especificar o *Distinguished Name* – DN, que é composto por todos os nomes necessários para o acesso ao objeto específico.

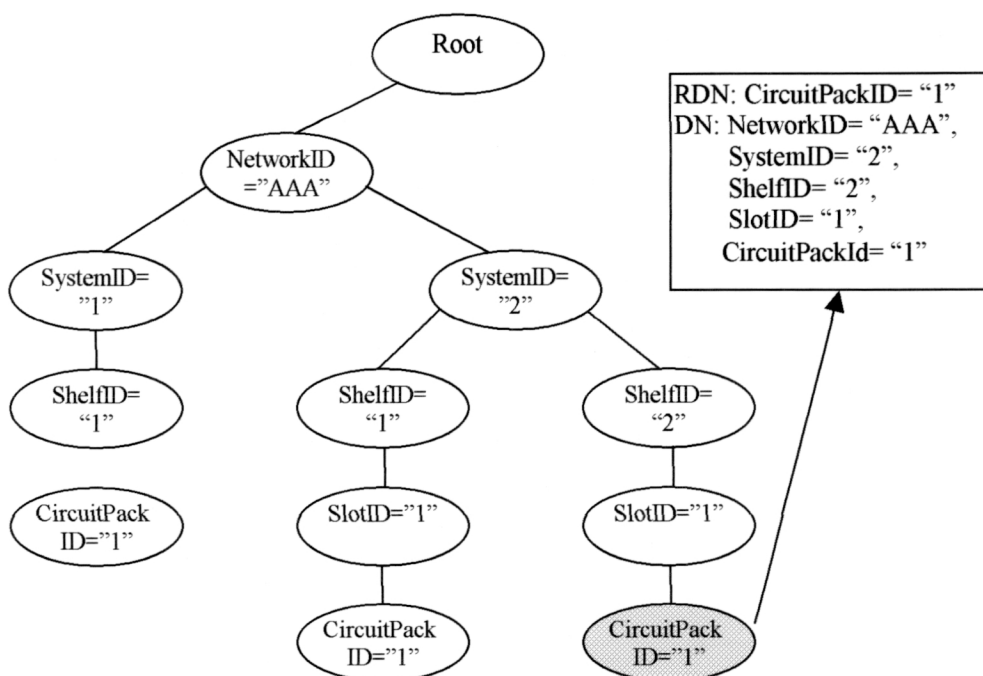


Figura 22 - Exemplo de árvore de nomeação.

3.2.4 Opções

As possibilidades de opções em padrões conduzem a problemas no relacionamento *interworking*, assim GDMO propõem que a opcionalidade não seja permitida em definições de objeto gerenciadas, a não ser que esteja relacionada a algumas características opcionais do recurso gerenciado, ou à alguns subconjuntos padronizados providos pela gerência OSI. Por exemplo, se um padrão de protocolo inclui um número opcional de classes, a classe de objeto gerenciada correspondente poderia ser definida de tal modo que sua funcionalidade varie de acordo com as classes que são implementadas em um determinado sistema.

3.2.5 Consistência

O GDMO contém vários itens necessários para manter a consistência dos vários padrões nos quais as classes de objetos gerenciados serão desenvolvidos. O objetivo é facilitar a implementação pelo gerente de objetos gerenciados, encorajando-o a usar as definições existentes de outros componentes de classes de objetos gerenciados, e reduzir a complexidade para o usuário, permitindo instalações das facilidades de gerência com um número reduzindo de soluções diferentes para resolver problemas semelhantes. O principal motivo é levar o gerente de sistemas gerenciados à reutilizar componentes disponíveis.

3.2.6 Notação da GDMO

Neste tópico serão descritos todos os aspectos utilizados pelo GDMO para a definição das classes de objetos gerenciados, utilizando para isso um conceito chamado *template*.

Templates

A notação usada para definir classes de objetos gerenciados está baseada no conceito de *templates*. Uma *template* é um modelo para a especificação de algum aspecto em uma classe de objeto gerenciado. Por exemplo, para a definição de um atributo, a definição da *template* de atributo identifica que itens podem ou que serão especificados para produzir uma definição de atributo válida.

As *templates* podem ser consideradas como um conjunto de formas pré-estabelecidas, com partes que o usuário (o gerente de objeto gerenciados) precisa. Cada *template* pode ser combinada com outras *templates* para construir uma definição completa de uma classe de objeto gerenciado. Na Figura 23 é apresentado um exemplo de *template*, definida em M.3100, representando um objeto gerenciado chamado *network*.

```

network MANAGED OBJECT CLASS
  DERIVED FROM    "Recommendation X.721: 1992":top;
  CHARACTERIZED BY
    networkPackage PACKAGE
      BEHAVIOUR
        networkDefinition;
      ATTRIBUTES
        networkId    GET;;;;
    CONDITIONAL PACKAGES
      userLabelPackage PRESENT IF "an instance supports it";
  REGISTERED AS {m3100ObjectClass 1};

```

Figura 23 - *Template* MANAGED OBJECT

Referências entre templates

A maioria das *templates* definidas em GDMO fazem referências a outras *templates*, como mostrado na Figura 24. Por exemplo, a *template* **MANAGED OBJECT CLASS** pode referenciar uma ou mais *templates* **PACKAGE**. O efeito de tais referências é que parte da especificação incluída na *template* de referência é importada para a *template* que cria a referência.

Os rótulos nas *templates* são os meios pelos quais são feitas as referências; se a *template* referenciada está no mesmo documento com a *template* de origem da referência, então o próprio rótulo da *template* é suficiente para identificar a referência. Porém, se as *templates* referenciadas estão em um documento diferente, o rótulo da *template* é precedido por um identificador único do documento, que pode ser seu nome ou um identificador de objeto (OBJECT IDENTIFIER) definido em ASN.1. Este mecanismo de referência provê os meios através dos quais podem ser desenvolvidas definições modulares, e é o mecanismo que permite a reutilização de definições, de componentes ou de classes de objetos gerenciados.

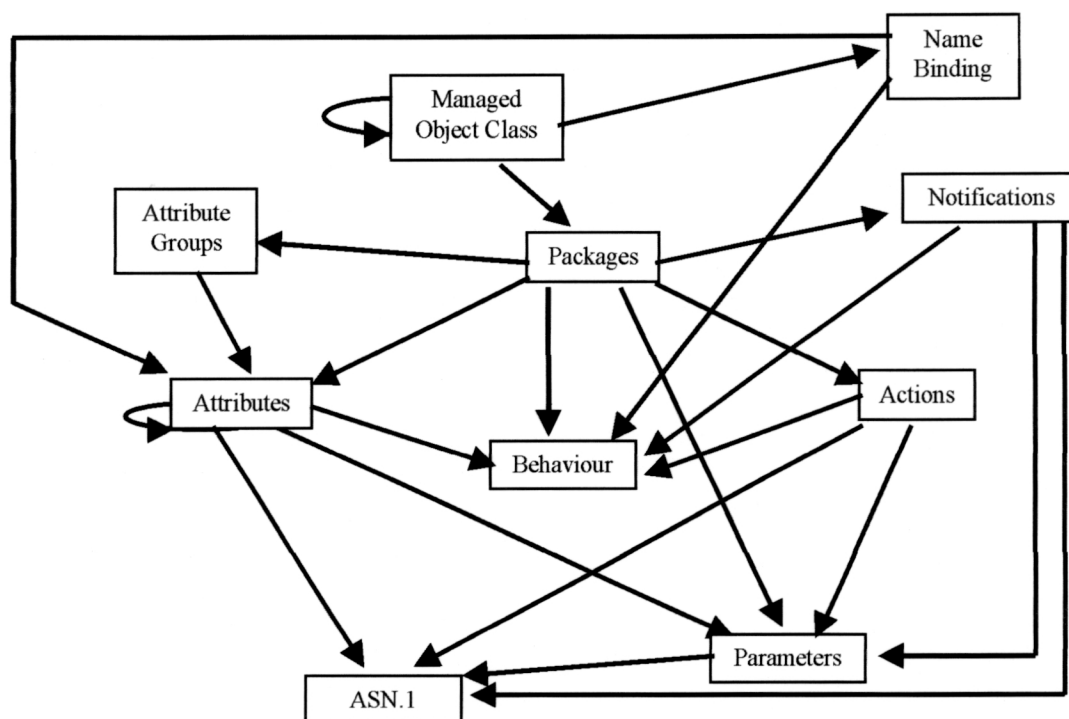


Figura 24 - Referências entre *templates*.

Referências para módulos ASN.1

Algumas das *templates* definidas em GDMO incluem referências para tipos de dados ou definição de valores em ASN.1, utilizados para especificar elementos de dados que serão transportados no protocolo de gerenciamento - CMIP. Isto é conseguido por meio de uma referência a um tipo ASN.1 ou definição de valor incluído em um módulo ASN.1 que é parte da mesma *template*.

3.2.7 As Templates

Neste tópico serão apresentadas as *templates* utilizadas pelo GDMO para fazer a definição de classes de objetos gerenciados.

Elementos da classe de objeto gerenciado

O processo de definir uma classe de objeto gerenciado envolve a especificação dos seguintes elementos:

- atributos e suas faixas de valor
- operações em atributos
- outras operações no objeto gerenciado
- notificações que podem ser emitidas
- definições de comportamento

- grupo de atributos
- *packages*
- regras de nomeação

Para especificar os elementos necessários às classes de objetos gerenciados, foram definidos nove *templates* diferentes. As *templates* definidas são:

- *template* MANAGED OBJECT CLASS – identifica os *packages* obrigatórios e condicionais de um objeto gerenciado, define a herança existente com outros *managed object class*.
- *template* PACKAGE – define uma coleção de atributos, notificações, operações e comportamento (*behaviour*).
- *template* PARAMETER – define o comportamento, a sintaxe de parâmetros e erros associados com atributos, ações, notificações, criação e remoção de um objeto.
- *template* ATTRIBUTE GROUP – define o agrupamento de atributos. Permite que as operações sejam executadas sobre um grupo de atributos.
- *template* ATTRIBUTE – define o tipo, a estrutura e característica de um atributo.
- *template* BEHAVIOUR – define uma descrição textual dos aspectos comportamentais de uma classe de objeto, de uma notificação, de uma operação ou de um atributo.
- *template* ACTION – define a sintaxe e uma descrição comportamental de uma ação em particular.
- *template* NOTIFICATION – define a sintaxe e uma descrição comportamental de uma notificação em particular.
- *template* NAME BINDING – define a estrutura de nomes para as instâncias de uma determinada classe de objeto gerenciado.

Template **ATTRIBUTE**

A *template* **ATTRIBUTE** é usada para definir a sintaxe de um tipo de atributo e seu comportamento. Sua estrutura é mostrada na Figura 25. A sintaxe de um atributo define como os

valores do atributo serão carregados no protocolo de gerenciamento. Um atributo pode ser *set-valued* ou *single-value*¹.

A construção **MATCHES FOR** permite a especificação de regras de pesquisa que podem ser aplicadas ao valor do atributo, por exemplo, no caso de um atributo *set-valued*, poderia ser legítimo testar o atributo para igualdade, mas não para testar ordem. Estas regras de pesquisa determinam quais operadores de filtro são legítimos quando um filtro de CMIS é aplicado ao atributo. Por exemplo, um operador de filtro que testa o valor atual de um atributo para determinar se é igual a um valor especificado será válido se o construtor **MATCHES FOR** da definição de atributo especificar **EQUALITY**.

O construtor **BEHAVIOUR** permite a especificação do comportamento do atributo. Este uso particular deve ser limitado àqueles aspectos do atributo que se aplicam a todos os *packages* (e consequentemente a classe de objeto gerenciado) nos quais o atributo pode aparecer, não sendo apropriado a inclusão de aspectos de comportamento específicos a um *package* particular, a menos que seja certo que o atributo só será usado naquele *package*.

O construtor **PARAMETERS** permite especificar parâmetros ao tipo do atributo a serem incluídos em sua definição. Neste contexto, os parâmetros são geralmente usados para estender o erro de CMIS para expressar as condições de erro específicas do atributo.

¹ atributos que são criados em ASN.1 com base em SET OF são chamados *set-valued*, todos os outros são *single-value*, ou seja, é definido por um único valor no momento da codificação concreta.

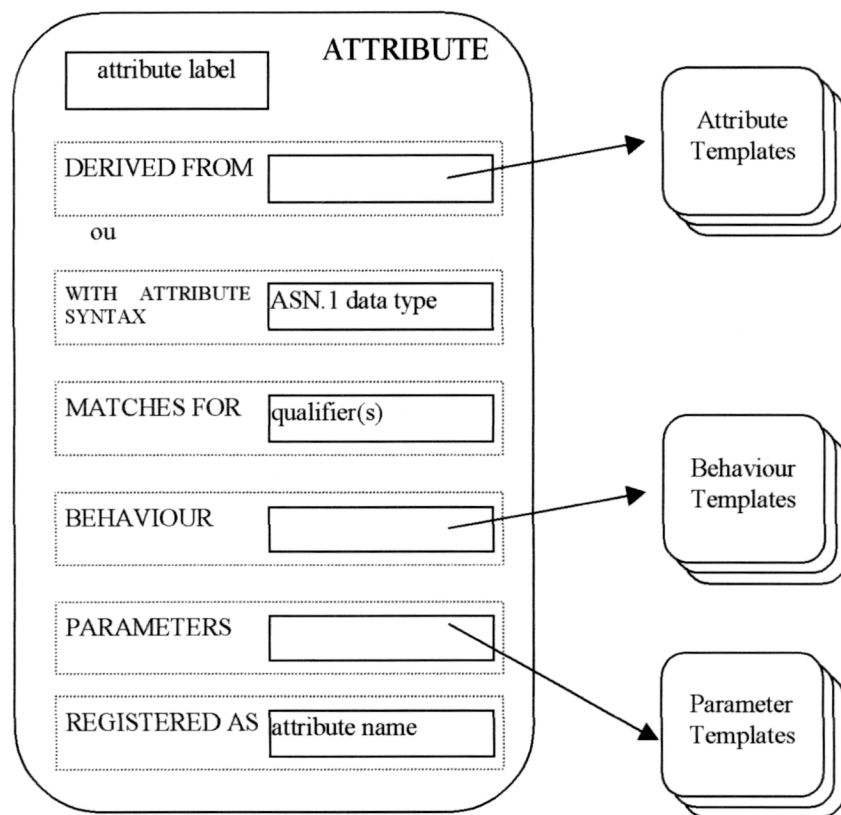


Figura 25 - Modelo da template ATTRIBUTE

Um identificador único atribuído pelo construtor **REGISTERED AS** é usado como o nome de tipo do atributo. Este identificador é levado nos parâmetros de serviço CMIS primitivos quando é necessário identificar o tipo do atributo. Em particular, é usado na construção dos componentes de nomes de instância de objetos gerenciados, conhecido como *Relative Distinguished Name* - RDN, e para identificar o atributo nas operações **Get** e **Replace**.

Template ACTION

A *template ACTION* é usada para definir operações em um objeto gerenciado que não pode ser modelado pelo emprego de operações predefinidas, como **Get** ou **Replace**. A estrutura deste modelo é mostrada na Figura 26.

O construtor **BEHAVIOUR** é usado para definir um comportamento específico para a operação *Action*. Os elementos que deverão ser listados aqui são:

- efeito sobre um objeto gerenciado de uma *action*;
- efeito sobre um recurso de uma *action*;
- restrições que podem ser aplicadas à uma *action*, por exemplo, em quais condições uma *action* poderia ser rejeitada.

A presença ou ausência do construtor **MODE CONFIRMED** definem os modos de operação que podem ser aplicados a *action*. Se **MODE CONFIRMED** está presente, a *action* só pode operar em modo confirmado, quer dizer, uma resposta sempre será gerada. Se não estiver presente, a *action* pode operar tanto no modo confirmado, como não confirmado.

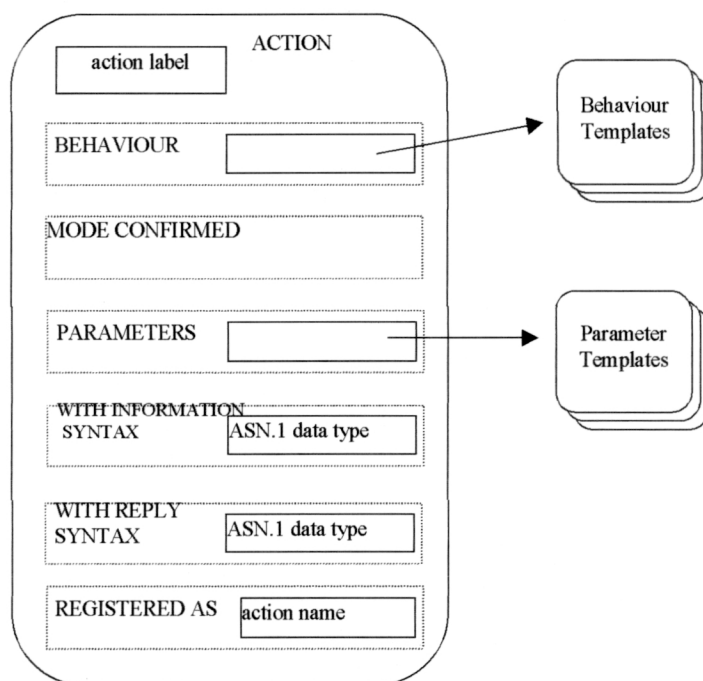


Figura 26 - Modelo da template ACTION

O construtor **PARAMETERS** permite a especificação de parâmetros específicos da *action* aplicáveis sobre todos os usos da *action*. Neste contexto, os parâmetros são só usados com a finalidade de especificar extensões para relatório de erros do CMIS para satisfazer as necessidades específicas da *action*. É tecnicamente possível usar parâmetros dentro das *templates ACTION* que definem os conteúdos da informação da *action* e/ou *reply-syntax*; porém, isto é impróprio, como os construtores **WITH INFORMATION SYNTAX** e **WITH REPLY SYNTAX** que são providos especificamente para aquele propósito.

Os construtores **WITH INFORMATION SYNTAX** e **WITH REPLY SYNTAX** são usados para especificar as sintaxes a serem colocadas em uma invocação *action* pelo protocolo CMIP, e na resposta de *action* correspondente se o modo confirmado é usado. Qualquer construtor pode estar ausente, independentemente do modo de operação da *action*. Em uma invocação *action*, não pode haver nenhuma necessidade para colocar qualquer argumento adicional diferente do tipo da *action*, e em modo confirmado, não há nenhuma necessidade para especificar uma sintaxe de resposta (*with reply syntax*) a menos que haja informação específica

para ser passado na confirmação diferente de simplesmente indicar que a operação foi completada.

O construtor **REGISTERED AS** atribui um identificador único a ser usado como nome da *action* nome. Este identificador é usado nos parâmetros de serviços CMIS primitivos, quando é necessário identificar o tipo da *action*. Seu valor estabelece o contexto no qual a estrutura da informação de *action* e sintaxes de resposta podem ser entendidas.

Template NOTIFICATION

A *template NOTIFICATION* é usada para definir as notificações que podem ser emitidas pelos de objetos gerenciados. A estrutura da *template* é mostrada na Figura 27.

O construtor **BEHAVIOR** é usado para definir um comportamento específico para a notificação. Os elementos que são apropriado aqui são:

- se a notificação é gerada como um resultado direto de algum evento *resource-related*, ou como resultado de alguns eventos derivados como um *threshold* em um contador que foi excedido;
- a sucessão de eventos em um recurso que ativaria uma notificação.

Não existe forma de definir os modos de operação que se aplicam a uma notificação, todas as notificações podem operar em modo confirmado ou modo não confirmado, conforme descrição do serviço CMIS.

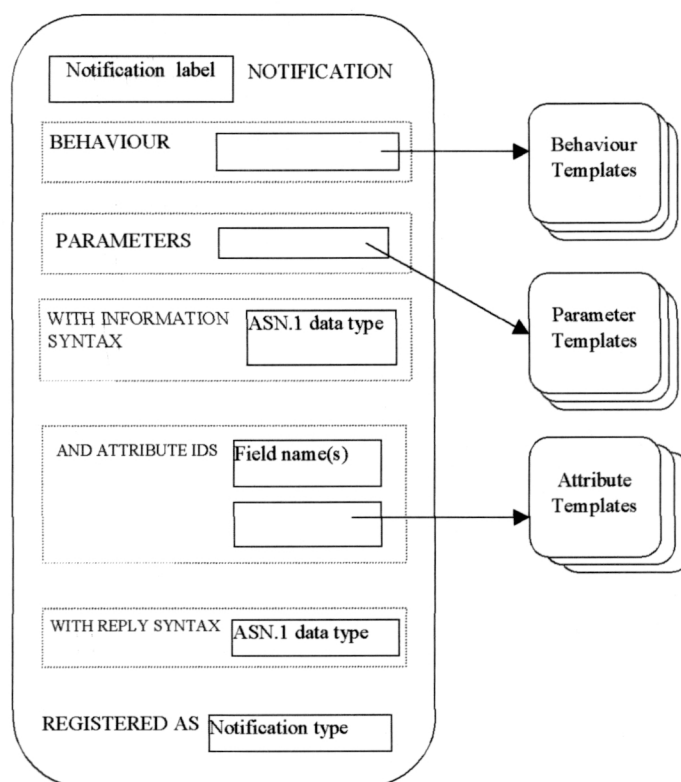


Figura 27 - Modelo da template NOTIFICATION

O construtor **PARAMETERS** permite a especificação de parâmetros para a notificação, aplicável à todos os usos da notificação. Neste contexto, os parâmetros são só usados com a finalidade de especificar extensões para a emissão de erro CMIS, para satisfazer as necessidades específicas da notificação. É tecnicamente possível usar parâmetros, como com a *template ACTION*, para definir os conteúdos da informação e/ou das *reply syntax*; porém, isto é impróprio, como os construtores **WITH INFORMATION SYNTAX** e **WITH REPLY SYNTAX** são especificamente providos para este propósito.

O construtores **WITH INFORMATION SYNTAX** e **WITH REPLY SYNTAX** são usados para especificar as sintaxes a serem levadas pelo CMIP em uma ativação da notificação, e na resposta da *action* correspondente se o modo confirmado é usado. Qualquer construtor pode estar ausente; na ativação da notificação, não pode haver nenhuma necessidade em prover qualquer informação adicional diferente do tipo da notificação, e em modo confirmado, não há necessidade para especificar uma sintaxe de resposta a menos que haja uma informação específica à ser passada na confirmação, diferente de simplesmente indicar que a notificação foi recebida.

Um identificador único é atribuído, usando o construtor **REGISTERED AS**, para ser usado como nome do tipo da notificação. Este identificador é usado nos parâmetros de serviço CMIS, quando é necessário identificar o tipo da notificação. Seu valor distingue uma forma de

tipo de notificação particular de todos os outros, e então permite entender a estrutura da informação de notificação e das sintaxes de resposta.

Template *BEHAVIOUR*

A *template BEHAVIOUR*, ilustrada na Figura 28, é usada para definir o comportamento objeto gerenciado ou de um componente deste. A definição de comportamento não é restrita de qualquer forma, e pode ser descrita por meio de texto legível, técnicas de descrição formais, linguagens de alto nível, ou outras técnicas que envolvam uma descrição textual, ou combinações de ambas. Talvez um desenvolvimento futuro da sintaxe de GDMO possa resultar na especificação de mecanismos unificados para descrições formais nesta *template*.

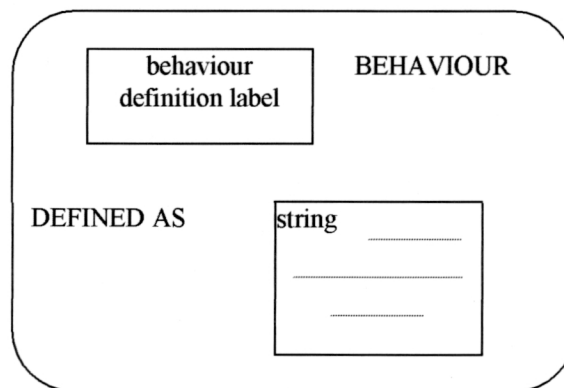


Figura 28 - Modelo da template BEHAVIOUR

Template *ATTRIBUTE GROUP*

Os grupos de atributos permitem agrupar atributos para possibilitar a execução de operações sobre o grupo. As operações disponíveis são **Get** e **Replace** com valor *default*, e o agrupamento de atributos especifica um nome que pode ser usado com estas operações, sendo interpretado como se fosse diretamente equivalente ao conjunto de nomes dos atributos individuais.

A *template ATTRIBUTE GROUP* define o relacionamento de um grupo de atributos, se o grupo é fixo ou extensível, e descreve o propósito do agrupamento. A Figura 29 apresenta a estrutura da *template ATTRIBUTE GROUP*.

O construtor **GROUP ELEMENTS** define todos os atributos que são os membros obrigatórios do grupo.

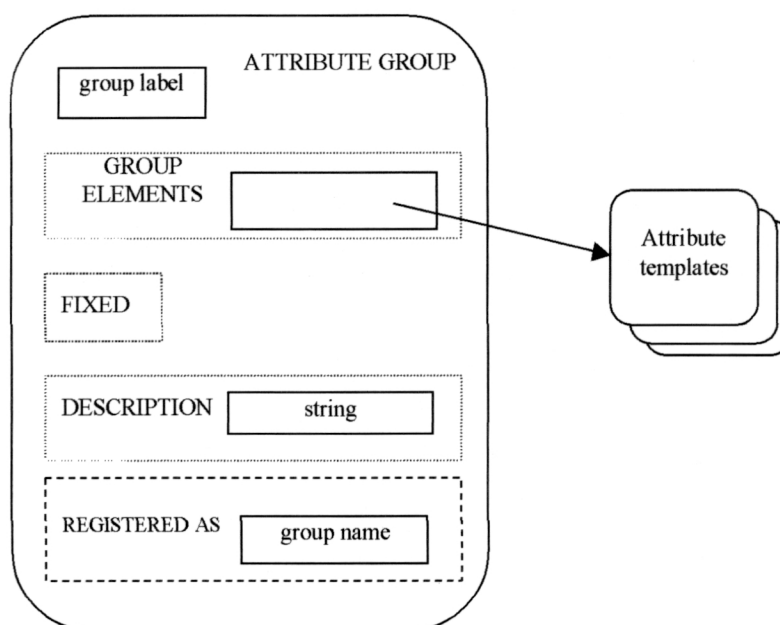


Figura 29 - Modelo da template ATTRIBUTE GROUP

Se o grupo de atributo é definido para ser extensível (omitindo o construtor **FIXED**), o construtor **GROUP ELEMENTS** pode ser omitido completamente, ou pode ser usado para definir o centro da sociedade do grupo. A extensão do grupo é conseguida somando atributos para o grupo quando este é referenciado na *template* **PACKAGE**.

Se o grupo de atributo é definido para ser fixo (incluindo o construtor **FIXED**), os atributos listados no construtor **GROUP ELEMENTS** definem o relacionamento do grupo.

Quando um objeto gerenciado é instanciado, a definição de qualquer grupo de atributos que contenha deve ser consistente com os atributos que o objeto gerenciado contém; quer dizer, o grupo de atributo não incluirá os atributos que não estão instanciados no objeto gerenciado. Para assegurar isto, exige-se que qualquer atributo definido para ser membro de um grupo deve ser referenciado no construtor **ATTRIBUTES** de um ou mais *packages* obrigatórios da classe de objeto gerenciado.

O construtor **DESCRIPTION** permite uma descrição textual do propósito do agrupamento possa ser incluído.

Um identificador de grupo usado no construtor **REGISTERED AS** define um identificador de acesso rápido, que pode ser usado nos parâmetros de serviços CMIS, que executam as operações em atributos, em particular nas operações **Get** e **Replace**. Quando usada deste modo, a operação é executada como se o identificador de grupo tivesse sido substituído pelos identificadores do conjunto de atributos, que são membros do grupo no objeto gerenciado consultado.

Template **PACKAGE**

A *template* **PACKAGE** permite agrupar um conjunto relacionado logicamente de características de uma classe de objeto gerenciado. Esta *template* reúne elementos definidos em *templates* **BEHAVIOUR**, **ATTRIBUTE**, **NOTIFICATION**, **ACTION** e **PARAMETER**, como mostrado na Figura 30. Como indicado na descrição da *template* **MANAGED OBJECT CLASS**, um *package* é definido em separado ou incorporado na definição de classe, como um elemento obrigatório ou condicional.

O construtor **BEHAVIOUR DEFINITIONS** permite especificar aspectos do comportamento da classe de objeto gerenciado que é específico ao *package*. Tal comportamento pode incluir relacionamentos entre os valores de atributos, relacionamentos entre os elementos de comportamento (*behaviour*) do *package* e o recurso propriamente dito, que o objeto gerenciado controlará, e a relação entre o comportamento do objeto gerenciado e as operações e notificações que reconhece.

O construtor **ATTRIBUTES** lista qualquer atributo que é incluído no *package*, junto com uma lista de propriedades para cada atributo, que define:

- operações disponíveis no atributo (GET, REPLACE, ADD, REMOVE);
- valores *default*, inicial, requerido e permitido;
- parâmetros que especificam qualquer extensão para a relação de erros do CMIS, que informa um mecanismo que aplica às operações disponíveis no atributo.

O valor *default* especificado para um atributo é usado em operações **REPLACE WITH DEFAULT**, e também pode ser usado no momento da criação do objeto gerenciado. O valor pode ser especificado como um valor fixo, pelo emprego de uma especificação de valor ASN.1, ou como um valor derivado, por meio de uma descrição textual das regras de derivação a serem utilizadas. Estas regras permitem que o gerente de objeto gerenciado faça considerações dinâmicas fixando valores *default*. Por exemplo, o valor *default* pode ser definido de maneira dependente de valores de atributos de outro objeto gerenciado.

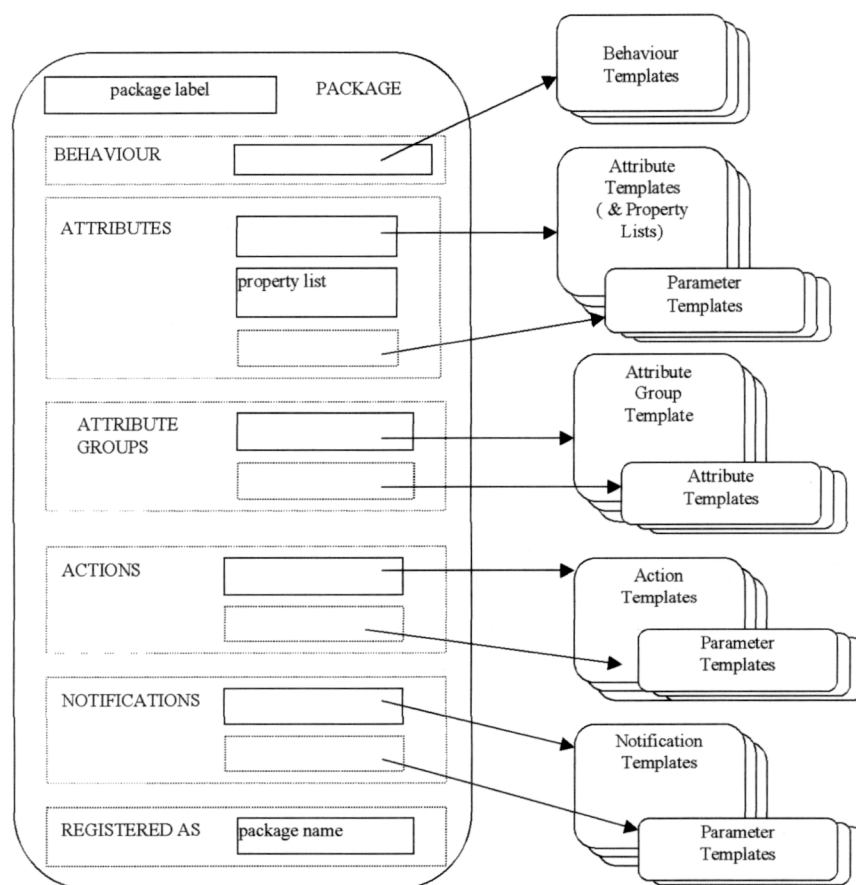


Figura 30 - Modelo da template PACKAGE

Um valor inicial obrigatório pode ser definido para uso quando é criado um objeto gerenciado. Como os valores *default*, o valor inicial pode ser um valor estático, ou um valor derivado de uma regra. Se a especificação de valor inicial é omitida, valores iniciais para o atributo são determinados no momento da criação por valores especificados na operação **Create**.

A especificação de valores permitidos deixa ao gerente restringir um conjunto de valores que um atributo pode assumir de algum subconjunto dos valores que são possíveis para o tipo do atributo. É aconselhável evitar a especificação dos valores permitidos, a menos que a semântica do atributo seja tão inconcebível, que os valores excluídos nunca serão relevantes em uma subclasse. Se nenhuma especificação de quais valores permitidos é incluída, então não existe restrição nos valores que os atributos podem possuir.

O construtor **ATTRIBUTE GROUPS** permite a inclusão de grupos de atributo na definição do *package*. Os grupos de atributo fornecem um meio mais fácil para acessar atributos logicamente relacionados, executando uma operação **Get** em um grupo de atributos tem os mesmos efeitos como executando operações **Get** individuais em cada membro do grupo. Quando uma operação **Get** é executada usando um nome de grupo de atributo, a resposta é idêntica à

resposta que seria devolvida em uma operação **Get** que individualmente identificou os membros do grupo.

Os construtores **ACTIONS** e **NOTIFICATIONS** permitem a inclusão de ações e notificações na definição de uma *package*, junto com qualquer parâmetro que pode ser associado com eles. Os parâmetros usados neste contexto são específicos para:

- definir qualquer extensão necessária para o erro de CMIS que informa o vocabulário que é requerido pelo uso da operação ou notificação;
- definir qualquer extensão específica *package* para as sintaxes da operação ou notificação.

O uso posterior de parâmetros aplicados em casos onde uma operação genericamente definida ou notificação será usada (como definido em uma função de administração de sistemas, por exemplo) e a definição original das sintaxes associadas com a operação ou notificação foi projetada para ser extensível para satisfazer exigências mais específicas.

O construtor **REGISTERED AS** permite atribuir um identificador único global como o nome de uma definição de *package*. Se um *package* é referenciado por um construtor **CONDITIONAL PACKAGES**, tem que ter um identificador global, este identificador é usado para identificar o *package* no atributo *Packages*, que é herdado por todas as classes de objeto gerenciado da classe *top*. Verificando o conteúdo do atributo *Packages* então o gerente permite determinar qual dos *packages* condicionais foram instanciados em um determinado objeto gerenciado.

Template **MANAGED OBJECT CLASS**

A *template* **MANAGED OBJECT CLASS** forma o núcleo da definição de uma classe de objeto gerenciada. Com exceção da *template* **NAME BINDING**, todas as outras *templates* que formam uma parte da definição da classe de objeto gerenciado são referenciados, diretamente ou indiretamente, a partir desta *template*, como mostrado na Figura 31.

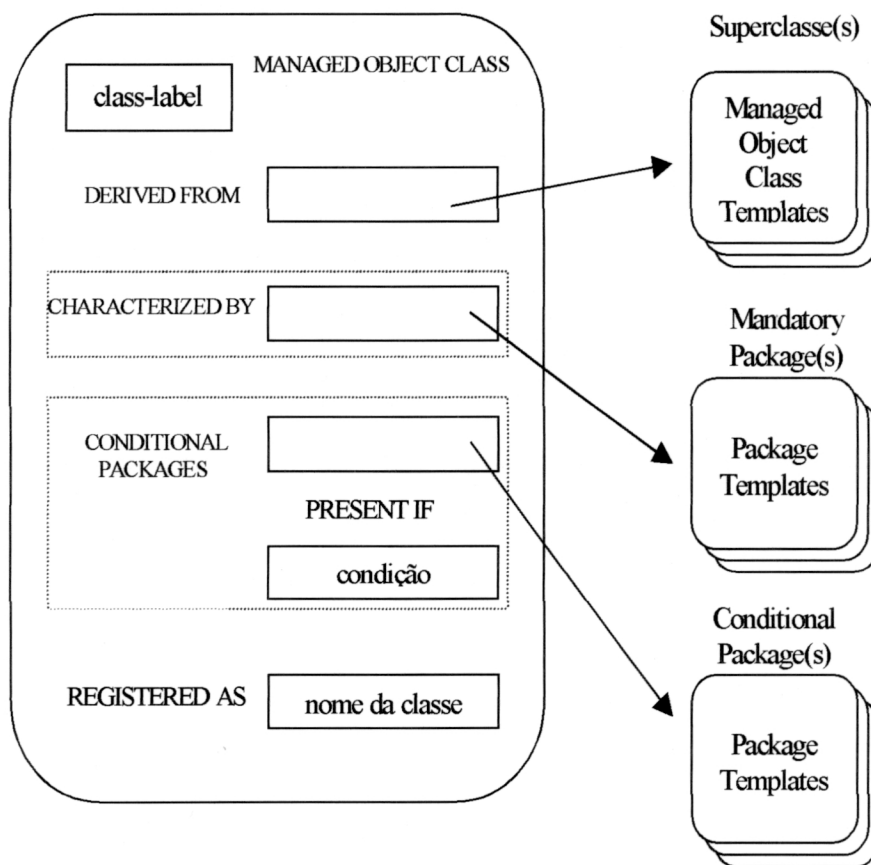


Figura 31 - Modelo da template **MANAGED OBJECT CLASS**

Todas as classes de objeto gerenciadas herdam características de uma ou mais superclasses; em última instância, todas as classes são derivadas de uma classe de objeto gerenciada especial, chamado *top*, que forma a raiz da hierarquia de herança. Os construtores **DERIVED FROM** da *template MANAGED OBJECT CLASS* provê os meios para especificar as superclasses da qual uma classe de objeto gerenciada foi derivada, e de quem herda as características. Embora marcado como opcional na definição do modelo formal, este construtor deve estar presente em qualquer definição de classe de objeto gerenciado diferente de *top*. As regras que especificam de que maneira a herança e especificação adicionais são combinadas para formar a definição da classe de objeto gerenciada é descrita em *Management Information Model* [X.720].

Os demais elementos da *template* se preocupam com a especificação de refinamentos para as superclasses relacionadas na construção **DERIVED FROM**. Isto envolve a adição da especificação de pacotes para a referida especificação, através de referências a um ou mais *templates PACKAGE*. Os elementos *packages* permitem definir uma coleção de atributos, operações, notificações, parâmetros e comportamentos. Um *package* pode conter elementos que

aumentam a especialização da superclasse, ou que especificam uma faixa de atributos possíveis de serem herdados.

O construtor **CHARACTERIZED BY** lista todos os *packages* obrigatórios que serão incluídos em todas as instâncias da classe, e a construção **CONDITIONAL PACKAGES** indicam listas de *packages* cuja inclusão em uma instância da classe é dependente de uma condição que é avaliada no momento da instanciação. Este construtor pode ser usado para modificar as condições que foram aplicadas aos *packages* herdados. Por exemplo, um *package* condicional que aparece em uma superclasse pode ser convertido a um *package* obrigatório em uma subclasse.

O construtor **REGISTERED AS** é usado para atribuir um identificador único à classe de objeto gerenciado. Este identificador é levado nos parâmetros de serviço CMIS quando é necessário identificar a classe de objeto gerenciada.

A herança e o uso de *packages* provêm dois mecanismos poderosos para estruturar definições de objeto gerenciados, e para reuso de especificações que podem ser aplicadas a mais de uma classe de objeto gerenciado. A herança forma um mecanismo natural para definir versões refinadas de classes de objeto gerenciado existentes. Isto pode ocorrer como resultado de revisão de um padrão de classe de objeto gerenciado onde o padrão revisado apresenta uma funcionalidade adicional, mas mantém compatibilidade com as definições anteriores.

Os *packages* condicionais formam um mecanismo natural para definir classes de objeto gerenciados que têm um núcleo de funcionalidade para o qual podem ser somadas outras capacidades além das já definidas.

Template NAME BINDING

A nomeação de objetos gerenciados está baseada na idéia de que os objetos estão contidos dentro de outros objetos gerenciados. E a *template NAME BINDING* provê o mecanismo na qual os objetos gerenciados são instanciados e onde estão contidos. Sua estrutura é mostrada na Figura 34.

A *name binding* define, para as instâncias da classe de objeto gerenciado identificadas pelo construtor **SUBORDINATE OBJECT CLASS**, o atributo que será usado para nomear essas instâncias, quando eles são contidos dentro de instâncias de classe de objeto gerenciados identificadas pelo construtor **NAMED BY SUPERIOR OBJECT CLASS**. O qualificador **AND SUBCLASSES** pode ser adicionado a qualquer um (ou ambos) destes construtores. Este qualificador permite usar a *name binding* não somente com a base da classe de objeto gerenciado

nomeada no construtor, mas também com qualquer subclasses daquela classe, a qualquer profundidade na hierarquia de herança. A característica **AND SUBCLASSES** é útil em circunstâncias onde é antecipado que uma determinada classe de objeto gerenciado será refinada futuramente, e permite fazer isto sem a necessidade da redefinição das ligações de nome.

O exemplo da Figura 32 apresenta como ocorre o processo de contenção de objetos em uma definição de NAME BINDING.

```

coffeMakerClass-locationClass  NAME BINDING
  SUBORDINATE OBJECT CLASS      coffeMakerClass;
  NAMED BY SUPERIOR OBJECT CLASS locationClass;
  WITH ATTRIBUTE                  cmNumber;
  CREATE;
  DELETE;
REGISTERED AS { appNameB 1 };

coffeMakerClass-roomClass      NAME BINDING
  SUBORDINATE OBJECT CLASS      coffeMakerClass;
  NAMED BY SUPERIOR OBJECT CLASS roomClass;
  WITH ATTRIBUTE                  cmNumber;
  CREATE;
  DELETE;
REGISTERED AS { appNameB 2 };

locationClass-floorClass      NAME BINDING
  SUBORDINATE OBJECT CLASS      roomClass;
  NAMED BY SUPERIOR OBJECT CLASS floorClass;
  WITH ATTRIBUTE                  locationNumber;
  CREATE;
  DELETE;
REGISTERED AS { siteNameB 3 };

```

Figura 32 - Exemplo de utilização de *name binding*.

Esta estrutura é instanciada da seguinte forma:

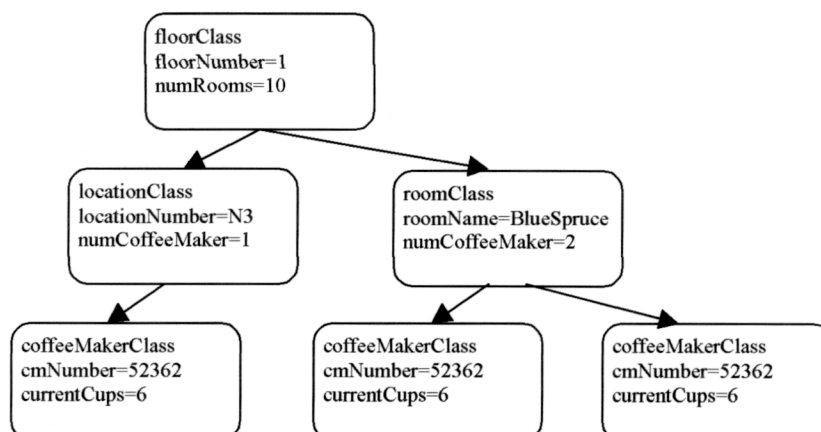


Figura 33 - Exemplo de árvore de nomeação.

O modelo também define se um objeto gerenciado pode ou não ser criado, se ele pode ou não ser apagado em um objeto superior, através de atividades de gerenciamento remota. Os

construtores **CREATE** e **DELETE** permitem que esta especificação seja feita junto com parâmetros que são usados com o CMIS.

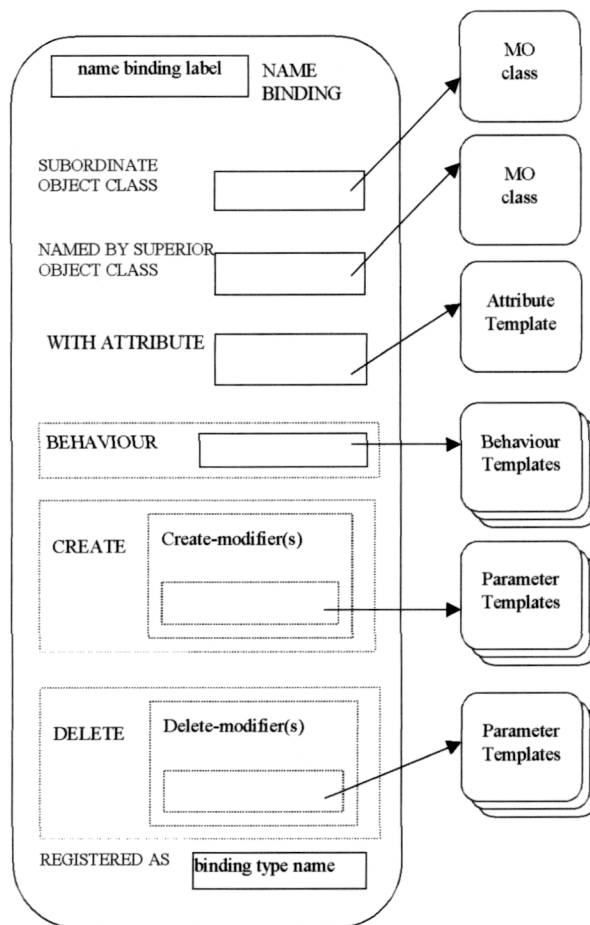


Figura 34 - Modelo da *template* NAME BINDING

O construtor **CREATE** permite ao gerente especificar opções de referência de objetos ou uma instância de nomeação automática, disponível quando da criação de instâncias subordinadas às classes de objeto gerenciado. Se o construtor **CREATE** está ausente na *template*, não é permitida a criação da classe de objetos gerenciados através de uma ação de gerenciamento remota.

O construtor **DELETE** permite ao gerente especificar se todos os objetos contidos devem ser apagados antes de apagar um objeto superior, ou se a remoção do objeto superior destrói todos os objetos nele contidos. As regras definidas por este construtor são aplicadas recursivamente. Se um objeto gerenciado *A* contém um objeto gerenciado *B*, que contém um objeto gerenciado *C*, que é o último na hierarquia, *A* só pode ser apagado (e assim *B* e *C* apagados) se a *name binding*, no qual *A* e *B* foram criados, utilizou o qualificador **DELETES-CONTAINED-OBJECTS**. Se o construtor **DELETE** está ausente da *template*, a remoção de objetos gerenciados da classe subordinada por meio de ação de gerenciamento não é permitida.

Como com a maioria dos outros modelos, o construtor **REGISTERED AS** é usado para nomear um identificador único para cada *name binding*. Este identificador permite um gerente falar para um agente *name binding* o qual será usado em uma operação **Create**.

Template **PARAMETER**

A *template* **PARAMETER**, cuja estrutura aparece na Figura 35, é um mecanismo projetado para definir argumentos específicos usados dentro de um módulo GDMO. Ela pode ser referenciada em campos *notification*, *action* e *attribute*. Ela possui uma estrutura sintática bem definida, mas é possível definir quais valores que podem aparecer nos campos.

O construtor **CONTEXT** identifica o contexto dentro do qual o parâmetro é usado. Vários contextos possíveis são especificados. Os contextos **ACTION-INFO**, **ACTION-REPLY**, **EVENT-REPLY**, indicam que o parâmetro é usado para preencher um **ANY DEFINED BY**, que é parte da estrutura da informação ou resposta definida para uma ação ou uma notificação. Isto permite a definição de ações e notificações que podem ser estendidas para incluir informação adicional que não foi colocada na definição original. O contexto **SPECIFIC-ERROR** permite o uso do processamento de falhas do CMIS para levar a informação de erro da classe do objeto gerenciado, que não pode ser expressada por meio dos valores padrões de erro do CMIS.

As *templates* **PARAMETER** podem ser referenciadas por outras *templates*, em uma variedade de contextos e para diversos propósitos. Os parâmetros podem ser fixos aos atributos, ou na própria *template* **ATTRIBUTE** (este parâmetro é aplicável a todos os usos do atributo) ou na *template* **PACKAGE**. O tipo do contexto que se aplica a este uso é o **SPECIFIC-ERROR**, onde o parâmetro é usado para definir condições de erro para operações **Get** ou **Replace** no atributo, que não pode ser representado usando os códigos de erro CMIS existentes.

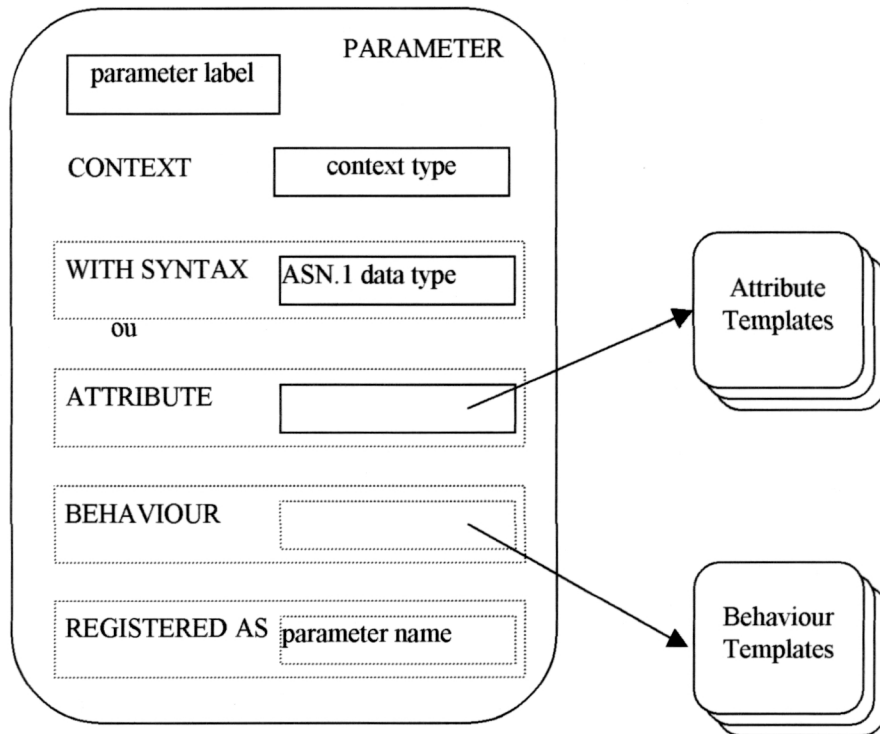


Figura 35 - Modelo da template PARAMETER

Podem ser usados parâmetros com a palavra-chave **CONTEXT**, como **ACTION-INFO**, **ACTION-REPLY**, **EVENT-INFO** e **EVENT-REPLY** para definir elementos de sintaxe associados com ações, informações de notificação e sintaxes de resposta. Estes contextos podem ser usados nas *templates* **NOTIFICATION** ou **ACTION**, ou nas *templates* **PACKAGE** onde as ações e notificações são referenciadas.

O construtor *syntax-or-attribute-choice* define a sintaxe ASN.1 que é usada para preencher o campo identificado pelo **CONTEXT**. Isto pode ser feito também através de referência direta a um tipo de definição ASN.1, ou através da referenciação a um modelo **ATTRIBUTE** cuja sintaxe será usada para este propósito.

Um identificador único atribuído pelo construtor **REGISTERED AS** é usado como nome de parâmetro. Este identificador estabelece o contexto que permite que o valor do identificador do objeto seja transportado para o campo **OBJECT IDENTIFIER** referenciado pelo construtor **ANY DEFINED BY**.

3.3 Notação de Sintaxe Abstrata - ASN.1

Um dos problemas fundamentais encontrado na interconexão de sistemas abertos tem sido o tratamento das diversas formas existentes de representação de dados. As técnicas utilizadas para definir as unidades de dados dos protocolos (PDU) dos níveis mais baixos do modelo de referência OSI são inadequadas para representar as estruturas de dados complexas e variáveis usadas nos padrões da camada de aplicação. A solução para os problemas de representação, codificação, transmissão e decodificação de estruturas de dados é ter um modo de descrevê-las que seja suficientemente flexível para ser utilizada em várias aplicações, isto é, as estruturas devem ser definidas de maneira independente de máquina e sem ambigüidades.

No início da utilização das redes de computadores, o termo *virtualização* foi utilizado para definir o acoplamento dos tipos de dados virtuais trocados por um protocolo, a estrutura de dados e a implementação particular. Assim, esta separação é baseada em duas noções, utilizadas até hoje:

- **Representação Abstrata:** cada tipo de dado é descrito de modo independente das estruturas orientadas à máquina e às suas restrições;
- **Representação Concreta:** é uma dada instância de um tipo de dado que pode ser transmitida na rede usando um octeto ou uma cadeia de bits. A representação deve ser compreensível, sem ambigüidades, entre emissor e receptor, como o valor de um tipo de dado.

A Notação de Sintaxe Abstrata – ASN.1 (*Abstract Syntax Notation One*) é uma notação formal para definição de sintaxes abstratas. Esta notação é um padrão genérico usada nas camadas de apresentação e aplicação para a definição de PDU e algumas informações referenciadas ou utilizadas pelas aplicações. A ASN.1 é definida através de duas especificações distintas: a linguagem propriamente dita e uma notação de sintaxe concreta conhecida como BER (*Basic Encode Rules*), que especificam um conjunto de regras associadas para a representação, em forma de bits, dos dados definidos através da notação abstrata.

A linguagem ASN.1 e a notação de sintaxe concreta BER são definidas, respectivamente, pelas normas [X.208] e [X.209], que são as normas utilizadas pelo GDMO.

Segundo a OSI, a ASN.1 é uma linguagem de descrição usada para definir **tipos de dados**. A notação usa um conjunto de dados primitivos e proporciona a construção de novos

elementos derivados desses tipos primitivos. Isso permite que novos tipos de dados sejam definidos especificamente para uma aplicação.

3.3.1 Convenções Léxicas

A gramática definida para notação ASN.1 [X.208] é descrita por uma BNF, conforme relacionado no anexo 5, neste documento são apresentadas as palavras reservadas. Neste texto as palavras reservadas aparecem sempre em letras maiúsculas.

Uma descrição ASN.1 consiste em vários *tokens* ou *lexemes*. Há apenas quatro tipos de *tokens* [ROSE90]:

- **Palavras:** constituídas de letras maiúsculas e minúsculas, dígitos e hífens, iniciando sempre por uma letra;
- **Números:** compostos de dígitos;
- **Cadeias:** que podem tomar três formas:
 - “cadeia de caracteres”, para cadeia de caracteres,
 - ‘0123456789ABCDEF’H, para cadeia de hexadecimais e
 - ‘01’B, para cadeia de bits;
- **Pontuação:** demais sinais utilizado;

Espaços em branco entre *tokens* são ignorados. Comentários podem ser utilizados onde um espaço em branco for válido, isto é entre quaisquer dois *tokens*. O texto de comentário deve iniciar com um sequência de caracteres “--” e terminar com a mesma sequência ou com fim de linha.

3.3.2 Módulos

Uma coleção de descrições ASN.1 deve estar sempre contida em um **módulo**. Uma definição de módulo ASN.1, segundo os adendos mencionados anteriormente, tem a seguinte forma:

```

<ModuleIdentifier> DEFINITIONS ::=
    BEGIN
        <Linkages>
        <Declarations>
    END
  
```

O elemento <ModuleIdentifier> pode ser um nome e, opcionalmente, Object Identifier que descreve a identificação de algum órgão de padronização com o objetivo de distinguir um módulo de qualquer outro módulo ASN.1. Um exemplo poderia ser [ROSE90]:

```
MTSAbstractService{
    joint-iso-ccitt mhs-motis(6) mts(3) modules(0) mts-abstract-service(3)
}
```

onde o MTSAbstractService é o nome do módulo e a descrição entre chaves é um *Object Identifier*.

Os elementos <Linkages> relacionam esse módulo com outros módulos.

```
EXPORTS
    SubmissionPort, DeliveryPort
IMPORTS
    Name FROM InformationFramework {
        joint-iso-ccitt ds(5) modules(1) information-framework(1)
    }
```

A parte referente a <Declarations> contém as definições ASN.1 propriamente ditas.

- para um tipo, o nome é iniciado sempre com uma letra maiúscula;
- para um valor (ou uma instância de um tipo), o nome é iniciado sempre com uma letra minúscula;
- para uma macro, o nome é composto somente de letras maiúsculas.

3.3.3 Tipos

A sintaxe para definição de um tipo ASN.1 é bastante simples:

```
NomeDoTipo ::= TYPE
```

Similarmente, uma “variável” ou, mais propriamente, uma instância de um tipo de dado correspondendo a um valor predefinido para ele é definida como:

```
nomeDoValor NomeDoTipo ::= VALUE
```

Dentro de um módulo em ASN.1 podemos classificar os tipos em:

- Tipos Simples
- Tipos Construtores
- Metatipos
- Tipos *Tagged*
- Tipos de Cadeia de Caracteres
- Subtipos

TIPOS SIMPLES

A notação ASN.1 define um conjunto de tipos simples que são especificados através de palavras reservadas e escritas integralmente com letras maiúsculas.

BOOLEAN

Representa um tipo de dado que pode assumir um dos dois valores: TRUE ou FALSE.

Por exemplo:

```
Multiple-defined-contextes ::= BOOLEAN  
simple Multiple-defined-contexts ::= TRUE
```

INTEGER

É um tipo de dado que pode assumir como valores os cardinais usados para contagem.

```
ContentLength ::= INTEGER  
length ContentLength ::= 100
```

Outra utilização é a enumeração de todos os valores possíveis para o tipo.

```
RegistrationMailType ::= INTEGER {  
    non-registered-mail(0),  
    registered-mail(1)  
}  
mymail RegistrationMailType ::= registered-mail
```

A grande utilidade desta forma levou a criação do tipo ENUMERATED quando da definição dos adendos à ASN.1.

ENUMERATED

É um tipo de dado que pode assumir qualquer um dos valores enumerados em sua definição. Os valores utilizados na enumeração representam apenas um conjunto de valores que uma instância do tipo de dados pode assumir. Não são permitidas operações como adição ou subtração.

```
RegistrationMailType ::= ENUMERATED {  
    non-registered-mail(0),  
    registered-mail(1)  
}  
mymail RegistrationMailType ::= registered-mail
```

REAL

É um tipo de dado que pode assumir qualquer valor real. Os valores reais são especificados através de um conjunto de três valores inteiros mantissa, base e expoente.

$$\text{Real} = \text{mantissa} * \text{base}^{\text{expoente}}$$

A mantissa e o expoente podem assumir qualquer valor inteiro, enquanto a base está limitada aos valores 2 ou 10.

```
TranscendentalNumbers ::= REAL
TranscendentalNumbers ::= { 27182818, 10, -7 }
```

BIT STRING

É um tipo de dado que define uma seqüência de zero ou mais bits, sem delimitação de tamanho. Não existe vinculação do número de bits com o valor 8 ou com qualquer outro valor.

```
RandomNumber ::= BIT STRING
number RandomNumber ::= '1010'B
```

Este tipo permite também a utilização de bits nomeados. O primeiro bit é sempre nomeado com 0 (zero) e os valores são definidos entre chaves ({}).

```
Attribute-Groups ::= BIT STRING {
    storage (0),
    security (1),
    private (2)
}
groups Attribute-Groups ::= {storage, private} -- '101'B
```

OCTET STRING

É um tipo de dado que define uma seqüência de zero ou mais octetos, sem delimitação de tamanho. Pode ser usado para representar caracteres ou qualquer outro dado orientado a byte.

```
UserName ::= OCTET STRING
initiator UserName ::= "anon" -- ou '616E6F6E'H
```

Relacionados ao tipo OCTET STRING estão todos os tipos cadeia de caracteres.

NULL

É um tipo de dado que representa a possibilidade de ausência de informação ligada a um tipo.

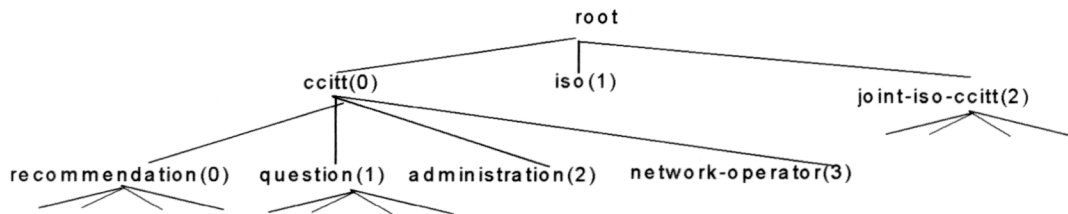
```
TimeOfDay ::= CHOICE {
    UTCTime,
    NULL
}
unixEpoch TimeOfDay ::= { "700101000000Z" }
unknownTime TimeOfDay ::= { NULL }
```

OBJECT IDENTIFIER

É um tipo de dado que identifica de forma única um objeto de informação registrado, por exemplo, em algum documento de padronização ou em algum módulo ASN.1 já definido. A sintaxe é bastante simples:

```
Document-Type-Name ::= OBJECT IDENTIFIER
```

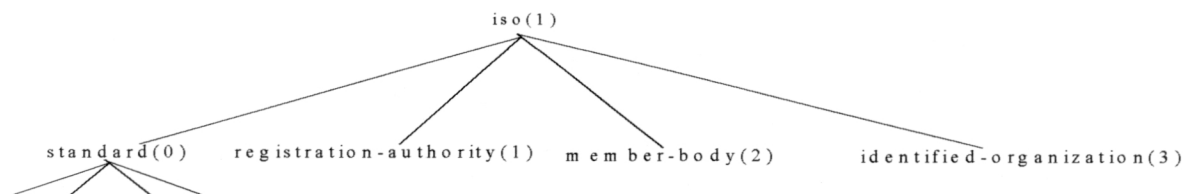

Este valor é representado por uma sequência de números inteiros positivos e possivelmente por uma breve descrição textual, que representam valores de uma árvore. Os valores são definidos a partir da raiz e continuam na estrutura até o objeto em questão.



Inicialmente, esta árvore é definida através de três arcos subordinados à raiz, indicando organismos com autoridade de registro, quais sejam:

- **ccitt(0)**: administrado pelo *Consultative Committee for International Telegraph and Telephone* (CCITT);
- **iso(1)**: administrado pelo *International Organization for Standardization* e pelo *International Electrotechnical Committee* (ISO/IEC);
- **joint-iso-ccitt(2)**: administrado conjuntamente pela ISO/IEC e CCITT.

Para ISO/IEC, foram definidos quatro nodos, subordinados da seguinte forma:



Eles podem ser assim descritos:

- **standard(0)**: tem como nodos subordinados cada *International Standard*, como, por exemplo, FTAM ou IS 8571. Assim, quando FTAM define objetos, eles se iniciam com o prefixo 1.0.8571. Cada norma é, então, responsável pela hierarquia de nomes usada sob seu prefixo.
- **registration-authority(1)**: reservado para uso pelas autoridades de registro OSI, quando elas são criadas.
- **member-body(2)**: tem um nodo subordinado para cada membro da ISO/IEC. O valor do *label* de cada nodo é um *Decimal Country Code* (DCC). Cada membro é, então, responsável pela organização posterior do seu respectivo espaço de nomes.

- **identified-organization(3)**: tem um nodo subordinado para qualquer organização que a ISO/IEC deseja favorecer. Isto possibilita a qualquer organização nomear objetos (sempre objetos proprietários) sem perigo de colisão na árvore de nomes.

A principal noção do tipo **OBJECT IDENTIFIER** é a compreensão do significado do caminho na árvore, ou seja, que este caminho representa o controle administrativo dos objetos sendo utilizados.

A descrição de um **OBJECT IDENTIFIER** pode ser feita de várias formas. O formato mais reduzido é listar os valores encontrados nos *labels* da árvore (1.0.8571.5.1) identificando um objeto:

```

fTAM-1 Document-Type-Name ::= { 1 0 8571 5 1 }
fTAM-1 Document-Type-Name ::= { iso standard 8571 5 1 }
fTAM-1 Document-Type-Name ::= { iso(1) standard(0) 8571 5 1 }
ftam-documents OBJECT IDENTIFIER ::= { iso(1) standard(0) 8571 5 }
fTAM-1 Document-Type-Name ::= { ftam-documents 1 }

```

OBJECT DESCRIPTOR

É um tipo de dado que identifica textualmente um objeto de informação. Diferente de um **OBJECT IDENTIFIER**, um **ObjectDescriptor** não é único e isto pode ocasionar problemas de duplicidade de nomes.

```

NickName ::= ObjectDescriptor
ber NickName ::= "Basic Encoding of a single ASN.1 type"

```

TIPOS CONSTRUTORES

Os tipos simples podem ser combinados para a construção de tipos mais complexos. Os tipos construtores são definidos por palavras reservadas e devem ser sempre escritos em letras maiúsculas.

SEQUENCE

É um tipo de dado que define uma lista ordenada de nenhum ou mais elementos, cada um deles definido por um tipo ASN.1.

```

VarBind ::= SEQUENCE {
    ObjectName,
    ObjectSyntax
}

```

Uma forma alternativa é a identificação dos tipos utilizados na lista, usando um rótulo para cada elemento. Esta identificação é usada para tornar a definição mais legível, e é bastante útil quando há vários elementos de mesmo tipo na lista.

```

VarBind ::= SEQUENCE {
                name      ObjectName,
                value      ObjectSyntax
            }

```

SEQUENCE OF

É um tipo de dado que define uma lista ordenada de nenhum ou mais elementos de mesmo tipo ASN.1. Este tipo é similar aos *arrays* dinâmicos de algumas linguagens de programação.

```

RoutingTable ::= SEQUENCE OF
                RoutingEntry

```

SET

É um tipo de dado que define uma lista não ordenada de nenhum ou mais membros, cada um deles definido por um tipo ASN.1. A diferença entre este tipo e o SEQUENCE é que a ordem dos membros não é significativa.

```

IMPIIdentifier ::= SET {
                user          ORAddress,
                user-relative-identifier LocalIPMIdentifier
            }

```

SET OF

É um tipo de dado que define uma lista não ordenada de nenhum ou mais membros de mesmo tipo ASN.1

```

MulticastGroup ::= SET OF Peer

```

ATRIBUTOS

Os dois tipos de atributos que podem ser utilizados nos tipos construtores são OPTIONAL e DEFAULT.

- **OPTIONAL** possibilita a declaração de algum elemento que pode não estar presente em alguma instância do tipo;
- **DEFAULT** é utilizado para determinar um valor predefinido de um tipo de dado, caso não lhe seja atribuído algum valor.

Por exemplo:

```

Interrupt-Request ::= SEQUENCE {
                fatal-error  BOOLEAN          DEFAULT TRUE,
                message       PrintableString  OPTIONAL
            }

```

É importante ressaltar que o uso do atributo **OPTIONAL** pode levar a problemas de ambigüidade quando de sua utilização. Por exemplo:

```

Example ::= SEQUENCE {
    alpha INTEGER OPTIONAL,
    beta  INTEGER OPTIONAL
}

```

Se em uma instância fossem atribuídos dois valores, o primeiro equivaleria a **alpha** e o segundo a **beta**, mas se fosse atribuído um único valor, não estaria claro se ele seria **alpha** ou **beta**.

A utilidade do atributo **COMPONENTS OF** está relacionada a definição de um tipo **SEQUENCE** que inclui todos os elementos de algum outro tipo construtor, sem ter de listá-los novamente. Por exemplo:

```

ORName ::= SEQUENCE {
    address COMPONENTS OF ORAddress,
    diretory-name Name OPTIONAL
}
ORAddress ::= SEQUENCE {
    ...
}

```

No exemplo anterior, o tipo **ORAddress** é uma seqüência de vários elementos. Segundo ASN.1, o uso do atributo **COMPONENTS OF** define a inclusão de todos os elementos da referida seqüência. Isto é bastante diferente da seguinte definição:

```

AddressAndName ::= SEQUENCE {
    address          ORAddress,
    diretory-name    Name OPTIONAL
}

```

que contém apenas dois elementos.

METATIPOS

São alguns tipos ASN.1 que não se enquadram nos tipos simples e construtores. Permitem uma melhor seleção de tipos.

CHOICE

É um tipo de dado definido como uma união de um ou mais tipos, e uma certa instância deste tipo pode tomar o valor de apenas um dos membros desta união.

```

TimeOfDay ::= CHOICE {
    actual-value UTCTime,
    not-available NULL
}

```

```

unixEpoch TimeOfDay ::= { actual-value "700101000000Z" }
unknownTime TimeOfDay ::= { not-available NULL }

```

ANY

É um tipo de dado definido como uma união de todos os possíveis tipos ASN.1. Este tipo é utilizado sempre que um tipo de dado não é definido pelo autor do módulo ASN.1

```

AP-title ::= ANY

```

Há ainda outra sintaxe para este tipo:

```

AlgorithmIdentifier ::= SEQUENCE {
    algorithm OBJECT IDENTIFIER,
    parameters ANY DEFINED BY algorithm OPTIONAL
}

```

onde *parameters* pode ser qualquer tipo ASN.1 válido, o qual é refinado por qualquer valor dinâmico tomado por *algorithm*.

O tipo **ANY** pode ainda ser utilizado através de abreviações. Sintaticamente, o tipo de dado **SEQUENCE OF ANY** é idêntico a **SEQUENCE** assim como **SET OF ANY** é idêntico a **SET**.

EXTERNAL

É um tipo de dado definido em algum outro documento fora do módulo corrente. Diferentemente do tipo **ANY**, um tipo **EXTERNAL** não necessariamente se referencia a um tipo definido usando ASN.1.

Se algum dado não pode ser modelado por algum tipo definido pela notação ASN.1, ele pode ser definido como **EXTERNAL**. O tipo **EXTERNAL** não é um tipo ASN.1 essencial.

```

EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
    direct-reference OBJECT IDENTIFIER
OPTIONAL,
    indirect-reference INTEGER OPTIONAL,
    data-value-descriptor ObjectDescriptor
OPTIONAL,
    encoding CHOICE {
        single-ASN1-type [0] ANY,
        octet-aligned [1] IMPLICIT OCTET STRING,
        arbitrary [2] IMPLICIT BIT STRING
    }
}

```

TIPO TAGGED

Os tipos *tagged* de ASN.1 são utilizados para identificar unicamente um tipo de dado. Para isto, ASN.1 associa um *tag*, ou rótulo, a cada tipo de dado, distinguindo um tipo de outro. Um *tag* possui duas partes: classe e número (não negativo).

As classes de *tags* são representadas por quatro palavras reservadas: **UNIVERSAL**, **APPLICATION**, **PRIVATE** e **Context Specific**. A palavra reservada pode ser suprimida no caso de **Context Specific**.

- **UNIVERSAL:** utilizado como identificação única no contexto global; são os *tags* definidos no padrão ASN.1;
- **APPLICATION:** utilizado para identificação única dentro de um dado módulo ASN.1, onde os tipos de dados devem ser definidos com *tags* distintos;
- **PRIVATE:** utilizado para identificação de tipos definidos pelas aplicações dentro de empresas ou organizações;
- **Context Specific:** utilizado para identificação única de elementos em tipos construtores, como em um tipo **SEQUENCE**, não possuindo nenhum significado fora deste contexto, tal como:

Example::= SEQUENCE {				
alpha	[0]	INTEGER	OPTIONAL,	
beta	[1]	INTEGER	OPTIONAL	
}				

A Tabela 2 relaciona todas as *tags* definidas nas construções em ASN.1:

Tag Universal	Tipo ASN.1
1	BOOLEAN
2	INTEGER
3	BIT STRING
4	OCTET STRING
5	NULL
6	OBJECT IDENTIFIER
7	Object Descriptor
8	EXTERNAL
9	REAL
10	ENUMERATED
11 – 15	Reservado para adendos
16	SEQUENCE, SEQUENCE OF
17	SET, SET OF
18	NumericString
19	PrintableString
20	TeletexString
21	VideotexString
22	IA5String
23	UTCTime
24	GeneralizedTime
25	GraphicsString
26	VisibleString
27	GeneralString
28	CharacterString
29 - ...	Reservado para adendos

Tabela 2 - Relação de *tags* Universais

A seguir são apresentados exemplos de utilização dos três primeiros tipos de *tags*:

```

IA5String ::= [UNIVERSAL 22] IMPLICIT OCTET STRING
Priority ::= [APPLICATION 7] ENUMERATED {
    normal(0), non-urgent(1), urgent(2)
}
MagicCookie ::= [PRIVATE 0] BIT STRING

```

TIPOS DE CADEIA DE CARACTERES

ASN.1 define alguns refinamentos do tipo OCTET STRING para representação dos tipos de cadeia de caracteres. São definidos oito diferentes tipos de cadeias - os três primeiros são mais utilizados:

- **NumericString**: define cadeias de dígitos numéricos (de 0 a 9);
- **PrintableString**: define cadeias de letras maiúsculas, minúsculas, dígitos, sinais de pontuação e espaços;
- **IA5String**: define cadeias de caracteres em código ASCII;

- **TeletexString;**
- **VideotexString;**
- **VisibleString;**
- **GraphicString;**
- **GeneralString;**

Tipos de dados para representação do tempo:

- **GeneralizedTime:** contém a data, expressa por quatro dígitos para o ano, dois dígitos para o mês e dois dígitos para o dia; hora, expressa em horas e em minutos e, opcionalmente, em segundos e fração de segundos com precisão fracionária qualquer; e um indicador opcional da relação desta hora com o *Coordinated Universal Time* (UTC), por exemplo [BRISA94]:

dateAndTime GeneralizedTime ::=

"19980304181906.740004Z"

- **UTCTime:** contém a data, expressa por dois dígitos para o ano, dois dígitos para o dia; hora, expressa até em segundos e um indicador opcional da relação desta hora com o *Coordinated Universal Time* (UTC), por exemplo:

atTheToneTheTimels UTCTime ::=

"890304101500-0800"

SUBTIPOS

Um subtipo ASN.1 é um refinamento de algum outro tipo ASN.1, denominado seu tipo pai (ou *prototype*). O tipo pai pode ser ainda outro subtipo definido. Para a maior parte dos tipos, a especificação de um subtipo consiste em uma lista entre parênteses, seguindo a definição do tipo:

SubType ::= ParentType (-- especificação do subtipo --)

Dentro da lista pode haver uma ou mais especificações de subtipo, todas separadas por “|”.

Para os tipos **SEQUENCE OF** e **SET OF** pode haver ambigüidades quando da utilização de subtipos. Vejamos o exemplo:

AmbiguousSubType ::= SET OF ParentType (-- especificação do subtipo --)

Uma especificação de subtipo *single value* é utilizada para indicar um subconjunto de valores válidos para o tipo pai.

```

ProtoType ::= ENUMERATED {
    first-class(0),
    business-class(1),
    coach-class(2),
    economy-class(3)
}
SubType ::= Prototype (coach-class | economy-class)

```

Um subtipo *contained* é utilizado para combinar valores definidos em tipos diferentes.

```

OtherType ::= Prototype ( INCLUDES SubType | business-class)

```

A definição OtherType pode assumir todos os valores de SubType, e ainda o valor business-class.

O subtipo *permitted alphabet* é utilizado para refinar cadeias de caracteres.

```

TenDigitString ::=
    PrintableString ( FROM ( "1" | "2" | "3" | "4" | "5" |
                           "6" | "7" | "8" | "9" | "0" ) )

```

Um subtipo *value range* é utilizado para limitar os valores máximo e mínimo que um tipo de dado pode assumir.

```

Prototype ::= INTEGER
NegativeNumber ::= Prototype ( MIN .. < 0 )
NonPositiveNumber ::= Prototype ( MIN .. 0 )
PositiveNumber ::= Prototype ( 0 <.. MAX )
NonNegativeNumber ::= Prototype ( 0 .. MAX )

```

Apenas os tipos pai INTEGER e REAL podem ser refinados da forma exemplificada anteriormente.

O subtipo *size range* é utilizado para limitar o tamanho que um valor pode assumir. Ele está dividido em dois casos:

Tipos pai que modelam *strings* (BIT STRING e OCTET STRING), o tamanho do subtipo indica quantas unidades lógicas podem estar presentes, definindo o menor e maior tamanho do *string*.

Tipos pai que modelam *arrays* (SEQUENCE OF e SET OF), o tamanho do subtipo indica quantos elementos ou membros podem estar presentes, definindo o menor e maior tamanho do *array*.

Consideremos os exemplos:

- Tipo onde os números utilizados podem ter de 7 a 10 dígitos

NANplan ::= TenDigitString (SIZE (7..10))

- Tipo composto por uma sequência de até 5 componentes do tipo **VarBind**

VarBindList ::= SEQUENCE OF (SIZE (1..5)) OF VarBind

Finalmente o subtipo *inner* é utilizado para refinar tipos construtores. Esses subtipos permitem que novos tipos construtores sejam definidos de modo que porções de outros tipos construtores estejam presentes e, opcionalmente, se presentes, que valores eles podem assumir. Exemplos dos subtipos *inner*:

```

PDU ::= SEQUENCE {
    requestId      RequestId,
    operation      ENUMERATED {
        get(0),
        put(1)
    },
    error-occurring Error-Occuring OPTIONAL,
    binding        Binding          OPTIONAL
}
Binding ::= SEQUENCE {
    Name      ObjectName,
    Value     ObjectSyntax OPTIONAL
}
GetRequest ::= PDU ( WITH COMPONENTS {
    operation      (get),
    error-occurring ABSENT,
    binding        PRESENT
    (WITH COMPONENTS { value ABSENT } )
} )
GetResponse ::= PDU ( WITH COMPONENTS {
    operation (get),
    binding ( WITH COMPONENTS { value PRESENT } )
} )

```

VALORES

O padrão ASN.1 define uma sintaxe para atribuir valores às instâncias de tipos de dados. Cada instância pode ser visualizada como uma variável correspondendo a um valor predefinido de um tipo de dado.

nomeDoValor NomeDoTipo ::= VALUE

Uma instância de uma PDU GetResponse poderia ser:

```

{
    requestId      17,
    operation      get,
    binding {
        name 1.3.6.1.2.1.1.1.1.0,
        value "unix"
    }
}

```

3.4 Considerações Finais

Neste capítulo foram apresentadas as informações sobre a arquitetura de gerenciamento OSI, relacionando suas áreas funcionais, as entidades utilizadas para a função de gerência, e principalmente o conjunto de normas que são necessárias para o seu entendimento. Também foi feita uma abordagem sobre a linguagem de especificação de objetos gerenciados GDMO - *Guidelines for Definition of Managed Objects*, e a linguagem de definição de tipos ASN.1 - *Abstract Syntax Notation One*.

CAPÍTULO 4

COMPARAÇÃO ENTRE CORBA E OSI

4.1 Introdução

Ao longo dos últimos anos a indústria da tecnologia de informação, tem desenvolvido ambientes de desenvolvimento de softwares orientados a objeto (linguagens, ferramentas de bibliotecas, banco de dados), para a realização de projetos orientados a objeto em sistemas de computadores. É, portanto, natural que os implementadores da gerência OSI tenham expectativas relativas a esses ambientes para o desenvolvimento de produtos para gerência de rede. O fator fundamental na realização desta expectativa, é a compatibilidade entre os modelos objeto subjacentes, sendo necessária uma comparação entre o modelo objeto da gerência OSI em relação aos modelos objetos usados nos ambientes distribuídos, mais especificamente CORBA.

Neste sentido, um grupo de trabalho conjunto do NMF e X/Open realizou um trabalho de comparação [OPS95]. Este capítulo baseia-se neste estudo. As seguintes comparações foram realizadas: objetivos do modelo, interfaces, características dos objetos, especificações de objeto e instanciação, taxonomia, referência a objetos e seleção de objetos e resolução de endereço.

4.2 Objetivos do Modelo

A Tabela 3 resume a comparação entre os objetivos dos modelos objetos da gerência OSI e OMG.

	Gerência OSI	OMG
Intenção de uso	Gerência de redes distribuídas.	Desenvolvimento de sistemas distribuídos orientados a objeto.
Interoperabilidade/portabilidade	Ao nível sintático e semântico. Nenhuma portabilidade de código, e interoperabilidade de comunicações.	Ao nível sintático e semântico portabilidade de código. Nenhuma portabilidade de comunicações. (CORBA 2 provê interoperabilidade).
Vantagens do uso	Gerência de componentes de rede heterogêneos.	Transparência para aplicações de plataformas heterogêneas básicas.
Componentes reusáveis	Biblioteca/catálogo de informação de gerência.	Biblioteca de Tipos de Interface.

Tabela 3 - Comparação entre Modelos Objeto OSI e OMG

O principal aspecto para entender a interoperabilidade é o entendimento sobre a sintaxe e a semântica da troca de informações. Para alcançar a interoperabilidade deve existir concordância em ambos.

O objetivo de cada grupo é promover interoperabilidade sintática e semântica, entre componentes de computação heterogêneos, os modelos diferem somente nas suas abordagens em especificar a interoperabilidade, e suas escolhas de pontos de interface nos quais ela existe.

Outro aspecto é a reusabilidade, um tema comum que é encontrado através da declaração dos objetivos do Gerência OSI e modelos OMG, é o dos componentes reusáveis com ênfase sobre biblioteca de objetos, portabilidade de aplicações e componentes de gerenciamento heterogêneos.

4.3 Interfaces

Os dois modelos escolheram posições diferentes para suas interfaces. A Gerência OSI apontou uma interface de comunicação entre sistemas de computadores, ao passo que o OMG optou por uma interface programável entre componentes de um sistema de computadores. Essas escolhas ditaram um estilo diferente de interface, entretanto, um número importante de conceitos são comuns aos dois modelos.

A Tabela 4 compara as interfaces definidas nos dois modelos objetos.

	Gerência OSI	OMG
Tipo de Interface	Interface de comunicação entre um sistema gerenciador e um sistema gerenciado; um agente, residente num sistema gerenciado, faz a mediação entre o sistema gerenciador e o objeto gerenciado.	Interface programável entre um chamador e um objeto. Interface inclui somente a assinatura
Protocolo Portador	A sintaxe e semântica do protocolo portador são definidos em ISO/IEC 9596-1 [X.710].	Independente do protocolo portador. A interoperabilidade dita um ou mais protocolos.
Interface Aberta	A ISO/IEC 9596-1 é um padrão internacional e muitas especificações de objetos gerenciados estão disponíveis publicamente.	Interfaces padrão têm sido definidas usando-se IDL para especificação de serviços de objetos.
Modelo de Protocolo	O modelo de protocolo para operações remotas baseia-se na passagem de mensagens não-bloqueante, com respostas do tipo normal e exceção; uma única mensagem inicial pode resultar em múltiplas respostas.	O modelo de protocolo tem dois estilos de semânticas de execução: no máximo uma vez (bloqueante com exceções); pelo menos uma vez (não bloqueante, sem garantia de entrega). A entrega de eventos também não é garantida pelos serviços de eventos COSS.
Concorrência de Interface	Uma associação de gerência que suporta operações e transferência de relatório de eventos, requer que as interações sejam processadas concorrentemente. Suporte para relatórios de eventos confirmados, requer concorrência de interface.	Não impedida pelo modelo objeto.

Tabela 4 - Comparação de Interfaces entre OSI e OMG.

Quanto ao tipo de interface, a escolha de interface feita pela Gerência OSI e OMG, reflete suas diferentes preocupações: para o primeiro as comunicações de gerência de rede, e para o outro o desenvolvimento de *software*. Estas escolhas são complementares, já que sistemas de gerência de rede orientado a objeto precisam ser desenvolvidos, e os implementadores necessitarão de ferramentas para o desenvolvimento de *software* orientado a objeto.

O uso de *stubs* cliente OMG implica um modelo de protocolo baseado em chamada de procedimento bloqueante. Entretanto, é definida uma interface de invocação dinâmica, que permite submeter operações síncronas atrasadas (a operação é chamada assincronamente e os resultados coletados posteriormente de modo síncrono). Uma implementação OMG, pode ser projetada para empregar *callbacks* à aplicação cliente invocada, para retornar os resultados de uma operação, permitindo assim a implementação de operações assíncronas. Entretanto, algumas implementações de CORBA não provêm um local de *callback* como parte de uma invocação de operação, impedindo o procedimento anterior. Assim, uma operação CORBA que é projetada para ser seguida por uma operação *callback*, deve prover um parâmetro de entrada contendo uma Referência ao Objeto (*ObjectReference*) sobre o qual será invocado o *callback*.

4.4 Características dos Objetos

Nesta seção são comparados os aspectos mais importantes dos modelos objetos, isto é, os próprios objetos, considerando o que são e do que tratam. Serão consideradas a noção de encapsulamento, as abstrações essenciais do interior de um objeto (estado, comportamento e métodos), os atributos visíveis na interface e os sinais que cruzam a interface.

A Tabela 5 apresenta a comparação das características dos objetos nos dois modelos.

	Gerência OSI	OMG
Descrição	Um objeto gerenciado é descrito na <i>Framework</i> de Gerência OSI como "uma visão de recursos que podem ser geridos através do uso de protocolos de Gerência OSI". Um objeto gerenciado obriga o encapsulamento; e é caracterizado pelos atributos que torna disponíveis, pelo seu comportamento e pelas invocações e notificações que cruzam suas fronteiras. Um objeto gerenciado pode se apresentar como se fosse uma instância da superclasse da classe real em que foi instanciado (alomorfismo).	Um objeto é descrito como "um pacote de dados e código usado para implementar construções computacionais ou modelar uma entidade de aplicação". Um Objeto OMG obriga o encapsulamento e é caracterizado pelos atributos que torna disponível, pelo comportamento, e pelos sinais que cruzam suas fronteiras. Um objeto pode ter mais que uma interface, mas somente através de herança.
Operações de Objetos	Uma operação pode ser inicializada pelo recebimento de uma invocação. Uma operação confirmada gerará uma ou mais respostas, e a operação não confirmada não gerará nenhuma; exceções são sinalizadas por meio de repostas de exceções. Invocações e respostas podem ser parametrizadas	Uma operação é iniciada de duas maneiras: ou através de um <i>stub</i> ou pela interface de invocação dinâmica. A execução normal de uma operação, resulta numa resposta (e retorna o controle para o chamador); a exceção é indicada por meio de um sinal de exceção e a transferência do controle vai para o manipulador de exceções. Todos podem ser parametrizados
Eventos de Objetos	Um comportamento não associado a uma operação pode ser sinalizado por meio de uma notificação. Uma notificação confirmada requer uma resposta, uma não confirmada não requer. Notificações e as respectivas respostas podem ser parametrizadas.	Objetos que geram eventos invocam uma operação sobre o serviço de notificação de evento. Distribuição de eventos é via <i>push model</i> ou <i>pull model</i> . Ainda não se definiu nenhum padrão de filtragem de eventos. Um receptor de eventos tipados tem que indicar o suporte para cada evento.
Comportamento	Suportado, mas somente via texto.	São permitidos efeitos colaterais mas não são especificados em IDL.
Atributos	Um atributo é caracterizado por um conjunto de valores de dados e qualificadores que especificam regras de comparação, restrição de escopo, valores iniciais e <i>default</i> e restrições de acesso. Atributos podem também ser manipulados através de um nome de grupo de atributos que identifica um ou mais atributos	Um atributo é caracterizado por um identificador, um tipo e um qualificador opcional " <i>read-only</i> ". Não há nenhum mecanismo para atributos agrupados.
Operações de Atributos	As operações embutidas sobre atributos são: <i>get attribute</i> ; <i>replace attribute</i> e <i>replace attribute with default</i> . Atributos <i>set-valued</i> incluem as operações: <i>add-member</i> e <i>remove-member</i> . Operações embutidas, suportam manipulação de múltiplos atributos ou grupo de atributos	As operações embutidas sobre atributos são: <i>set attribute</i> e <i>get attribute</i> . Operações embutidas podem somente obter/configurar (<i>get/set</i>) os valores de um único atributo em uma invocação.

	dentro de uma operação simples.	
Operações do Ciclo-de-Vida do Objeto	Operações embutidas para a criação e exclusão de instâncias de objetos. O criador pode especificar o nome ou pode delegar ao agente a definição do nome. Não prevê nenhum protocolo padrão para mover instâncias de objetos entre sistemas.	Nenhuma operação embutida para instaciação de interfaces de objeto por demanda do cliente. Entretanto objetos do tipo fábrica podem ser definidos, permitindo que clientes criem objetos específicos a uma operação. Os serviços do ciclo-de-vida COSS especificam exclusão de objetos, bem como operações cópia e movimento de implementações de objeto.
Comportamento do Atributo	O comportamento das operações orientadas a atributo é bem definido. A interação entre operações e atributos é deixada para a definição de <i>package</i> . As operações sobre atributos podem definir exceções específicas.	São definidas as operações orientadas a atributo <i>get</i> e <i>set</i> . A especificação de comportamento está somente em comentários textuais. Somente exceções padrão são suportadas sobre operações de atributo.
Tipo de Dados	Todos os tipos de dados ASN.1 primitivos são suportados, incluindo: BOOLEAN; INTEGER; BIT STRING; OCTET STRING; NULL; ANY DEFINED BY; CharacterString; OBJECT IDENTIFIER; REAL; ENUMERATED; e Generalized Time. Os seguintes tipos construídos são suportados: SET; SET OF; SEQUENCE, SEQUENCE OF, CHOICE	Tipos primitivos: integer, floating point, character, octet, boolean e ANY. Existem métodos de acesso pré-definidos para o tipo ANY. Valores de atributos e parâmetros de operações agregados suportados pelos tipos de coleção: sequence (lista ordenada do tamanho da variável); struct, Union, Enum, Array (tamanho fixo). Em IDL, a recursão dentro de declarações de tipos construídos, é permitida somente quando ocorrem em uma sequence.
Encapsulamento	Suportado.	Suportado.
Tipo de Dado Referência a Objeto	Realizado através de instância de objeto que são definidas por um nome distinguível (DN). É usado extensivamente como parâmetro na interface de comunicação, sendo usada a sintaxe X.500.	Realizado através do tipo referência a objeto (<i>Object reference</i>). Operações podem empregar uma referência a objeto como um de seus parâmetros. Sua sintaxe é transparente para a operação que a usa.
Referência a Tipo de Interface	Classes de objetos gerenciados, identificadores de atributos, notificações e ações são todos especificados através de um valor globalmente único denominado OBJECT IDENTIFIER, bem como através de um label de referência de um documento ASN.1 textual específico. Maiúsculas e minúsculas são levadas em consideração.	O tipo da interface tem um identificador de referência global através de nomes de Módulos. Tipos de atributos e operações são referidos usando nomes definidos dentro da especificações de interface. Maiúsculas e minúsculas não são levadas em consideração.

Tabela 5 - Características de Objetos

Ambos os modelos obrigam o encapsulamento estrito, isto é, os objetos são caracterizados pelo modo como eles aparecem ao mundo real e não como são construídos internamente. Com exceção das notificações e o suporte à operações embutidas, para manipulação de múltiplos atributos com uma simples invocação, os dois modelos apresentam poucas diferenças. Na prática, os objetos de ambos os modelos são construídos com códigos e dados para serem executados em sistemas de computadores.

As operações em ambos os modelos suportam a noção de um conjunto extensível de tipos, onde um objeto particular precisa obrigatoriamente suportar apenas um conjunto fixo de tipos de operações. O mecanismo para invocar tais operações é definido pelo protocolo subjacente.

Há a necessidade de definir uma *factory* OMG de interfaces de objetos genérica, para suportar a operação M-CREATE para objetos gerenciados OSI mapeados para interfaces CORBA.

Quanto aos eventos, a maior diferença entre os modelos de gerência OSI e OMG é o modo de especificação de eventos autônomos.

Na gerência OSI, eventos são comportamentos sem conexão com operações. Tipicamente, notificações de eventos são causadas por condições de alarmes, por exemplo perda de sinal, no recurso representado por um objeto gerenciado. Uma função de gerência de relatório de evento determina quais notificações de evento são encaminhadas adiante, como relatórios de evento para designadas destinações.

A noção de eventos foi acomodada nos serviços de objeto OMG. A principal diferença é que os eventos OMG são especificados como sendo suportados pelo receptor dos eventos. Um evento OMG é modelado como uma operação.

Os atributos na gerência OSI e OMG são diferentes de objetos:

- há sistemas de tipos distintos para objetos e atributos;
- objetos precedem atributos, isto é, atributos podem somente ser acessados dentro do contexto dos objetos;
- há operações embutidas para manipular atributos.

A gerência OSI permite especificações de erros específicos definidos pelo usuário, sobre operações de atributos, e define operações para manipular membros de atributos *set-valued* (*add*, *remove*). Se atributos da gerência OSI são mapeados para atributos OMG, então somente os erros padrão CORBA podem ser usados, resultando numa perda de informação. Além disso, não haveria meio de usar operações CORBA para adicionar ou remover membros de atributos *set-valued*. Uma alternativa seria mapear atributos de objetos gerenciados OSI para operações CORBA IDL, em casos onde exceções definidas pelo usuário, ou onde operações em membros de conjuntos são requeridas.

O encapsulamento em ambos os modelos de gerência OSI e OMG definem noções similares de encapsulamento, onde o estado de um objeto pode somente ser acessado, por meio de sinais que cruzam a fronteira do objeto, ou por meio de relacionamentos entre objetos.

Entretanto, a definição de interface do gerência OSI inclui além das operações que o objeto pode realizar, as notificações que um objeto pode emitir.

4.5 Especificação e Instância de Objetos

A abordagem de gerência OSI de separar especificações ligadas dentro de *packages* que são subsequentemente ligados dentro de classes de objeto, é reivindicada para melhorar o potencial para reuso das especificações. *Packages* condicionalmente ligados possibilitam a ligação *a posteriori* (*late binding*) e atendem a necessidade da gerência OSI para a variação controlada em especificações de classe. No modelo de gerência OSI, a união dos identificadores de classe de objeto e a identificação dos pacotes presentes, serve como o indicador das características de um objeto gerenciado particular. Para implementadores, os pacotes complicam a hierarquia de tipo. Na Tabela 6 são apresentados os principais aspectos dos objetos e suas instanciações no modelo OSI e OMG.

	Gerência OSI	OMG
Especificação de Atributo	Atributos e qualificadores são especificados independentemente de classes de objetos gerenciados.	Atributos e qualificadores são definidos somente dentro do escopo de tipos de interface de Objeto.
Ligação de Especificações	Especificações de atributos, operações, e eventos podem ser ligados dentro de <i>packages</i> por meio de uma especificação de <i>package</i> . Uma especificação de <i>package</i> pode especificar restrições adicionais sobre os atributos (ex. restrições de faixa) e sobre operações e eventos (ex. restrições sobre o tipo dos parâmetros); ela também especifica a semântica do <i>package</i> como um todo. As especificações podem ser ligadas dentro de classes de objetos gerenciados por meio de uma especificação de classe de objeto gerenciado. Estas ligações podem ser obrigatórias ou condicionais.	Especificações de atributos e operações são combinadas dentro de interfaces. Interfaces podem ser combinadas dentro de módulos. A opcionalidade não é suportada na especificação de interface. Os atributos e valores de parâmetros podem no entanto, conter escolhas nulas, existindo ainda um tipo de exceção pré-definida NOT-IMPLEMENTED. O Serviço de Evento OMG possui eventos definidos como operações de uma interface que opera na direção reversa (eventos são promovidos pela invocação do Objeto numa operação). Assim, um objeto gerenciado pode definir uma ou mais interfaces contendo todos os eventos que ele pode gerar; o gerente necessita suportar a interface.
Instanciação de Objeto	Instanciação pode ser <i>a priori</i> , por meio local, ou por meio de uma operação <i>create</i> CMIP.	Instanciação pode ser <i>a priori</i> ou por meio de uma invocação da operação sobre uma Interface <i>factory</i> específica da aplicação. Nenhuma interface <i>factory</i> específica é definida para CORBA.
Especificação de Comportamento	Prevê uma cláusula para definição de comportamento em linguagem natural, com uma recomendação para especificar pré-condições, pós-condições e invariantes.	Linguagem natural não estruturada, usando comentários IDL.
Ferramentas de Especificação	<i>Templates</i> GDMO - objetos. ASN.1 - tipos de dados e valores.	IDL

Tabela 6 - Comparação de objetos e instanciação.

A técnica de especificação OMG é mais monolítica sobre o tipo e não admite ligações posteriores: o tipo do objeto é uma indicação absoluta das características de uma instância do tipo. O reuso é alcançado pela herança (possivelmente múltipla) de definições de interface. Entretanto, há uma exceção padrão denominada “NOT-IMPLEMENTED” que deve ser tratada pelos invocadores quando uma implementação de objeto não suportar uma operação definida dentro destas interfaces.

As *packages* condicionais podem levar a situações onde todos os membros de uma classe não se comportam da mesma maneira: o uso de valores nulos em atributos e parâmetros tem o mesmo efeito em CORBA.

Os aspectos de comportamento de objetos gerenciados são especificados usando linguagem natural, embora seja recomendado que tal especificação seja acomodada em termos de pré-condições, pós-condições e invariantes. Tem sido tentado o uso de técnicas formais para a especificação do comportamento de classes particulares de objetos gerenciados. O OMG também propõe o uso de linguagem natural para especificar os aspectos comportamentais dos objetos.

A gerência OSI, fornece duas ferramentas para especificar formalmente a sintaxe de objetos gerenciados e mensagens associadas:

- a linguagem de *templates* GDMO para definir as características de objetos gerenciados;
- uma linguagem de definição de dados ASN.1 para definir os tipos de dados associados com atributos e parâmetros.

4.6 Taxonomia de Objetos

Uma taxonomia permite-nos raciocinar sobre coisas do mundo real, pelo agrupamento de instâncias em classes ou tipos e pelo relacionamento de diferentes classes ou tipos. Em geral, usa-se os termos classe e tipo como sinônimos, mas em sistemas orientados a objetos, classes e tipos são diferentes. Sendo apresentada as principais características na Tabela 7.

Um predicado sobre as características de um objeto; objetos são ditos estar em conformidade com um tipo. Um predicado adicional sobre as características de um supertipo.

O conjunto de objetos que são do mesmo tipo. Objetos dentro da classe podem ter uma ou mais *templates* usadas para a criação e gerenciamento de objetos. Todo tipo define uma classe, mas nem toda classe define um tipo. Uma subclasse de objetos permite a especialização da *template* de uma superclasse.

É um mecanismo para derivar a especificação de um descendente a partir de uma especificação de um ancestral tal que, além de suas próprias características, um descendente adquire todas (herança estrita), ou parte, das características de seu ancestral. Um descendente pode ter um único ancestral imediato (herança simples) ou múltiplos ancestrais imediatos (herança múltipla). Os relacionamentos resultantes de herança simples, podem ser representados por uma árvore; e os resultantes de herança múltipla, por um grafo direcionado acíclico.

	Gerência OSI	OMG
Classe de Objeto	O modelo de informação de gerência, classifica objetos gerenciados em classes. Especificações de classe são documentados por <i>templates</i> .	O modelo de objeto OMG especifica objetos pela sua interface. Não há um conceito direto de Classe de Objeto, entretanto uma <i>template</i> de objeto pode ser derivada a partir do grafo de herança de uma definição de interface.
Taxonomia	Classes de objetos gerenciados são relacionadas na hierarquia de tipos por meio da herança múltipla e estrita. A raiz da hierarquia é a classe de objeto gerenciado top , que especifica atributos para especificação de propriedades estáticas comuns para toda classe. É possível herdar de duas definições de classe que incluem a mesma ação ou atributo.	Tipos de objeto são relacionados numa hierarquia de tipo de interfaces suportadas, por meio de herança múltipla. A raiz da hierarquia é o tipo de objeto Object . É ilegal a herança de duas interfaces com o mesmo nome de operação ou atributo, ou redefinir uma operação ou nome de atributo dentro de interfaces derivadas.
Sistema de Tipo	Os tipos de objetos gerenciados são distintos de outros tipos (atributos, ações, notificações) dentro do modelo.	Tipos de interface de objeto são integrados dentro de uma hierarquia de tipo simples para todos os tipos dentro do modelo.

Tabela 7 - Comparação da Taxonomia de Objetos.

As taxonomias usadas nos modelos de gerenciamentos OSI e OMG, são muito similares: ambos são especificados em termos de interface de objeto e ambos usam a disciplina de herança múltipla. O modelo de gerência OSI especifica um número de atributos em **top** (*objectClass* e *nameBinding*); assim todos os objetos gerenciados contêm instâncias destes atributos.

O modelo OMG suporta a noção de substituição por meio de subtipos. O modelo de gerência OSI suporta uma noção similar por meio de uma capacidade limitada do subtipo chamada compatibilidade.

A representação da herança de objetos gerenciados em GDMO, no caso de atributos ou ações repetidas em *packages* herdados, requer tratamento especial quando traduzida para IDL CORBA.

4.7 Referência a Objeto

Existem dois conceitos distintos que não devem ser confundidos: identidade de um objeto e o seu nome. Informalmente, a identidade de um objeto é aquela que distingue-o de todos os

outros. Um nome é um manejo mnemônico usado para se referir a um objeto. A identidade de um objeto (referência a objeto) é difícil de se definir num senso formal, mas intuitivamente, é algo duradouro sobre um objeto; um nome por outro lado é fácil de se definir, sendo um símbolo (*token*) usado para referenciar um objeto.

Geralmente referências a objeto não têm estrutura e são implementadas como longos *bit-strings* para facilitar seu tratamento em computador. Nomes são normalmente simbólicos e mnemônicos.

A identidade de um objeto é uma propriedade única de um objeto que sem ambigüidade distingue-o de todos os outros objetos no universo ao longo do tempo.

Para se fazer referência a um objeto, um símbolo (*token*) único usado para referenciá-lo sem ambigüidade, durante seu tempo de vida; referências a objeto são associadas no momento da criação do objeto, e nunca são subsequentemente substituídas. A função de mapeamento entre uma referência a objeto e o objeto pode ser válida (a referência está associada a um objeto) ou inválida. Quando válida, a função é univalorada e singular, isto é, determina relacionamento 1:1.

Um símbolo (*token*) usado para referenciar um objeto; objetos diferentes podem ser associados a um mesmo nome, e um mesmo objeto pode ser associado a nomes diferentes; nomes podem ser substituídos durante o tempo de vida de objeto. Assim, a função de mapeamento do nome para objeto, quando válido, pode ser multi-valorada e não-singular, além de poder alterar com o tempo.

A referência de um objeto não revela ou define o protocolo portador para aquele objeto, não revelando também a sua localização. A referência a um objeto permanece válida após uma mudança de localização do objeto. A Tabela 8 apresenta a comparação entre os modelos CORBA e OSI.

	Gerência OSI	OMG
Referência a Objeto	O único conceito de referência a um objeto gerenciado é o conceito de instância de objeto, definido por um nome distinguível (DN).	Referência a Objeto em CORBA é um tipo transparente (não é visível externamente não tendo nenhum formato particular). Objetos podem ter mais que uma referência e são associados com referências a objeto durante sua criação.
Nome	Um objeto gerenciado deve ter um único nome distinguível, compreendido por uma seqüência de Nomes Distinguíveis Relativos (RDN), correspondendo a uma hierarquia de nomes constituída através da relação de conteúdo. Cada RDN é um único, par atributo/valor obrigatório, associado com o objeto gerenciado. O valor do atributo RDN é associado no momento da criação do objeto e não pode ser mudado durante o tempo de vida do objeto. O mesmo nome pode, entretanto, ser usado posteriormente.	Objetos podem ter um número arbitrário de nomes. O escopo é um contexto de nomes. Nomes são distintos de objetos; objetos necessitam ter nomes. Há mapeamento entre valores de nome e objetos; o mapeamento pode ser mudado a qualquer hora.
Modelo de Nomeação	É definido um conjunto hierarquicamente qualificado de contextos de nomes, baseado sobre uma relação de conteúdo para assegurar que nomes de objetos sejam únicos e não ambíguos dentro do contexto local do sistema agente, bem como num contexto global. Apelidos não são permitidos.	O Serviço de Nome fornece um grafo de contexto de nomes, potencialmente cíclico que pode possuir diversas raízes. Todos os nomes são mapeados para um <i>ObjectRef</i> (possivelmente de um outro contexto de nomes), assim um Objeto pode ter muitos nomes, e um único ponto no grafo pode ser alcançado via muitas rotas.
Transparência de Acesso	Não suportada. O protocolo CMIP é obrigatório.	Suportada - ORBs podem usar qualquer protocolo que seja mais apropriado. ORBs podem empregar camadas baixas OSI ou Internet. A interoperabilidade CORBA requer suporte de um protocolo padrão-IIOP.
Transparência de Localização	Não suportada para objetos nomeados no escopo <i>System</i> .	Suportada no CORBA. Os serviços de ciclo de vida, incluem operações <i>move</i> .
Independência de Localização	Não suportada para objetos gerenciados nomeados dentro do escopo do <i>System</i> .	Suportada no CORBA.

Tabela 8 - Comparação para Referência de Objetos.

A primeira vista, o esquema de referência a objeto mostra muitas diferenças: no modelo OMG de identidade do objeto é determinada por um valor transparente de referência do objeto, enquanto que no modelo de gerência OSI é ela é determinado pelo nome. Entretanto, se for proibida a renomeação ou movimentação de objetos gerenciados, e for cuidadosamente gerido o reuso de nomes de objetos gerenciados, os dois mecanismos são amplamente equivalentes, uma vez que nomes de objetos gerenciados são únicos e não ambíguos, e não são permitidos apelidos.

4.8 Seleção de Objeto e Resolução do Endereço

Dá-se o nome de **seleção** ao mecanismo pelo qual instâncias de objetos são selecionadas para receber invocações.

A seleção associativa é uma referência construída com predicados sobre as características dos objetos; este tipo de referência pode resolver para nenhum ou mais objetos.

A resolução de endereço do objeto é o mecanismo pelo qual um nome é mapeado para um endereço.

Sumário	Gerência OSI	OMG
Seleção Direta	Pelo nome.	Em OMG uma Referência a Objeto é um endereço, isto é, ela contém toda informação necessária para que haja a conexão com aquele objeto, independente de sua localização corrente. A interoperabilidade entre ORBs irá ser fornecida na próxima versão do CORBA, que irá requerer resolução de referências de objeto entre implementações ORB diferentes.
Seleção Associativa	É opcionalmente suportado o uso de referências associativas por meio de filtros. A semântica e sintaxe dos filtros são bem definidas. Com CMIP, uma invocação é feita diretamente indicando um escopo (sub-árvore de objetos gerenciados) e um filtro, cabendo ao agente encontrar os objetos alvo. O escopo pode ser toda a árvore de objetos gerenciados ou uma parte dela.	Uma referência associativa para um conjunto de nomes pode ser feita pela travessia do grafo de nomes ou pelo Serviço de Negociação (<i>Trader</i>). Um mecanismo padrão OMG ainda não foi definido. Comandos endereçados a um conjunto de objetos via uma referência associativa necessitaria de um mecanismo em duas fases: uma que resolva para um conjunto de objetos através de restrições sobre propriedades dos serviços suportados pelos objetos; e a outra que interaja com os objetos invocando as operações sobre eles.
Resolução do Endereço	O modelo não prescreve um mecanismo para resolução de endereço, mas isto pode ser provido pelo Diretório.	O ORB fornece resolução de endereço. O mapeamento do nome para uma referência do objeto é feito pelo Serviço de Nome.

Tabela 9 - Comparação do Mecanismo de Seleção de Objetos.

Ambos os modelos possuem aspectos similares com respeito à seleção do objeto: uma seleção direta de um único objeto através de seu nome ou através do identificador do objeto respectivamente; e seleção associativa sobre algum subconjunto de objetos que pode resolver para nenhum ou mais objetos. Em ambos os casos, a responsabilidade da resolução de endereço é delegada.

O modelo de gerência OSI define sintaxe e semântica específicas para os mecanismos suportados, já o modelo OMG não. A Tabela 9 apresenta a comparação entre os mecanismos de seleção de objetos do modelo OSI e da arquitetura CORBA.

4.9 Similaridades e Diferenças

A Tabela 10 abaixo apresenta as principais diferenças entre dos dois modelos.

Sumário	Gerência OSI	OMG
Interoperabilidade e Portabilidade	CMIP permite diferentes implementações interoperarem através de interfaces comunicantes. Não há portabilidade de aplicação.	Portabilidade de aplicação é providenciada pela especificação de uma interface. Não há protocolo comum para interoperar diferentes implementações.
Componentes Reusáveis	Biblioteca/catálogo de informações de gerenciamento	Biblioteca de tipos (repositório de interfaces)
Encapsulamento	Suportado	Suportado
Operações com Objetos	Suportado	Suportado. Não existe um serviço de criação embutido.
Comportamento	Suportado	Suportado
Atributos e Operações com Atributos	Suportado. Exceções específicas podem ser definidas.	Suportado. Exceções específicas não podem ser definidas.
Taxonomia	Grafo de hierarquia de tipos de objetos gerenciados suportado através de herança múltipla. Atributos, ações e notificações podem ser repetidas em classes derivadas, com propriedades adicionais.	Grafo de hierarquia de tipos de objetos suportado através de herança múltipla. Atributos e operações não podem ser repetidos nas especificações de interface derivadas.
Seleção Direta	Por nome global ou local.	Por referência a objeto.
Intenção de Uso	Gerência de redes distribuídas.	Desenvolvimento de software orientado a objeto.
Tipo Interface	Comunicações	Programação
Concorrência na Interface	Suporte necessário para eventos confirmados e operações / notificações sobre a mesma associação.	Não requerido, concorrência é implícita no modelo, especialmente onde chamadas com sincronização atrasadas são feitas.
Modelo de Protocolo	Operações remotas	Chamada de procedimentos. CORBA2 define protocolo de interoperabilidade.
Múltiplas Respostas	Suportada explicitamente pelo CMIP.	Embora não especificada explicitamente no CORBA, múltiplas respostas podem ser conseguidas pela passagem de uma referência na chamada.
Eventos de Objetos	Suportada pelo CMIP. Notificações são especificadas como parte da definição de um objeto gerenciado.	Um modelo de eventos padrão é definido nos serviços CORBA. Notificações não são especificadas como parte da interface do transmissor de evento. O receptor de evento registra uma interface para recepção dos eventos.
Ligação Tardia (Late Binding)	Alguns graus de opcionalidade são manipulados pelo GDMO através de ligações condicionais.	A implementação de uma instância de um objeto particular é determinada quando a instância é criada. É possível alterar uma implementação em tempo de execução, embora a semântica não esteja bem definida. Valores de parâmetros e atributos podem ser definidos através do

		tipo union , podendo incluir alternativas nulas. O fato de um objeto suportar uma interface não significa que todas as operações definidas serão implementadas. Um objeto pode simplesmente não implementar uma operação, e produzir uma exceção quando ela for invocada.
Seleção Associativa	Suportada através de definição de escopo na sub-árvore de instâncias de objetos gerenciados e através de filtros.	Permitido. Pode ser implementado através de seleção em duas fases por meio de um negociador (<i>Trader</i>).
Especificação	Atributos são definidos primeiro, e então combinados dentro de pacotes, que são combinados dentro de classes de definições de objetos gerenciados.	Não monolítica, já que um objeto pode suportar várias interfaces (através de herança).
Ferramentas de Especificação	GDMO / ASN.1	IDL
Formalidade de Especificação	Sintaxe	Sintaxe

Tabela 10 - Principais similaridades entre os modelos OSI e OMG.

4.10 Alternativas de Reconciliação dos Modelos

Os modelos poderiam ser reconciliados através de mudanças em um, ou outro, de modo a simplificar os mapeamentos entre eles. O grupo JIDM (*Joint XOpen/NM Forum Inter-Domain Management*) decidiu que este método não é viável, já que grandes investimentos foram feitos, para produzir cada um dos modelos existentes [OPS95]. Ele assumiu que os mapeamentos entre as especificações em cada modelo são restringidos pelas definições existentes em cada modelo. [MAZ95]

Duas diferenças entre modelos são complementares, e podem ser exploradas neste sentido. O fato das interfaces serem orientadas a comunicação em um e à programação no outro, poderia ser explorado usando-se a interface de programação OMG, sobre protocolos padronizados OSI. Deste modo, os dois modelos seriam contemplados. Entretanto, isto não é uma tarefa trivial, e dificuldades tais como a operação síncrona do modelo OSI versus a operação síncrona no modelo OMG deveriam ser superadas.

Tendo explorado as diferenças complementares, é necessário ainda reconciliar as diferenças fundamentais e acidentais:

- alinhar os modelos;
- providenciar mediação em tempo de execução entre as implementações dos modelos;
- providenciar ferramentas para mapeamento das notações.

Idealmente, todas as diferenças conflitantes entre os modelos deveriam ser resolvidas por um alinhamento completo, mas isto é impraticável. Cada modelo é o resultado de muitos anos de

trabalho consensual, e é pouco provável que um consenso de alinhamento seja rapidamente obtido. Entretanto, tentar desenvolver subconjuntos destes modelos que sejam mais próximos entre si parece realizável. Particularmente, as estruturas de sintaxe de atributos recursivas do GDMO são problemáticas no que concerne sua tradução para IDL CORBA, necessitando de considerações especiais durante o mapeamento e eventualmente necessitando de intervenção manual [GEN96].

4.11 Considerações Finais

Neste capítulo foram apresentadas as considerações realizadas sobre as arquiteturas OSI e CORBA. Na arquitetura OSI foram estudados os aspectos de gerência e implementação, na arquitetura CORBA os principais pontos que foram analisados se referem ao modelo objeto concreto, que é um refinamento do modelo abstrato definido pela própria OMG [OMA92].

Ficou claro que é possível a utilização dos elementos definidos dentro do modelo OSI através da arquitetura CORBA, principalmente pela facilidade de implementação dos objetos.

CAPÍTULO 5

TRADUÇÃO DE GDMO-ASN.1/OSI PARA IDL/CORBA

5.1 Introdução

Para ser possível realizar a tradução entre GDMO e IDL, é necessário primeiramente mapear as definições de tipos e valores existentes nas definições ASN.1 para declarações de tipos em IDL/CORBA. A notação ASN.1 tem um sistema de tipo muito mais poderoso que os sistemas de tipos IDL. Portanto, a tradução necessariamente implica em perda de informação, por exemplo, perda de valores *tag*, não representatividade dos tipos *sub-range* e dos tipos de constantes compostas. Deve ser necessário definir um mecanismo para capturar estas informações para um possível uso no sistema da tradução interativa, nossa abordagem fornece uma ferramenta auxiliar para o mecanismo de tradução das informações contidas nas *templates* GDMO,

Este capítulo apresenta uma forma de realizar a tradução das especificações em GDMO e ASN.1 para IDL/CORBA, este trabalho foi realizado sobre as especificações definidas pelo X/Open. [OPS95]. É apresentado um mecanismo pelo qual se torna possível que ao implementador² exportar características adicionais que podem ser utilizadas pelo mecanismo de tradução interativa, ver Figura 36.

² É definido como *implementador* à pessoa responsável pela implementação da especificação transformada.

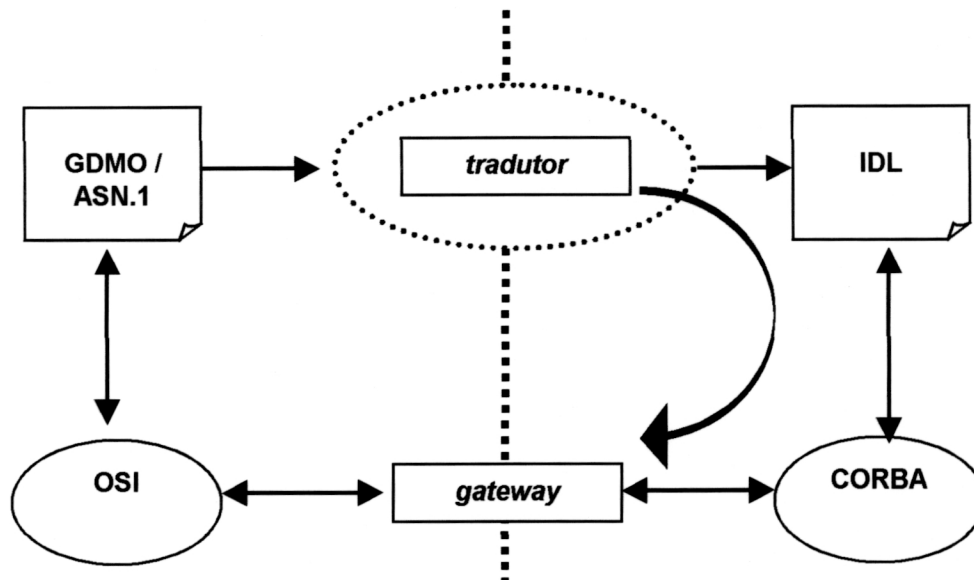


Figura 36 - Mecanismo de tradução

5.2 Tradução de Tipos ASN.1 para Tipos IDL

Nesta seção serão mostrados como os tipos declarados em ASN.1 poderão ser representados em IDL/CORBA, podendo ocorrer, em determinadas situações, alguma perda de informação.

5.2.1 Sumário das Regras de Tradução do Módulo ASN.1

A seguir é apresentado um sumário da tradução dos tipos ASN.1, definidos dentro de um módulo ASN.1, para definição de tipos IDL, dentro de um módulo IDL.

1. Mapear cada módulo ASN.1 para um módulo IDL;
2. Mapear somente as atribuições de tipo e de valores;
3. Resolver os conflitos de nomes entre itens ASN.1;
4. Criar nomes para tipos IDL definidos a partir de tipos ASN.1 anônimos;
5. Ordenar os tipos gerados para evitar a dependência entre tipos;
6. Declarar os tipos importados em módulos ASN.1 como *typedef* do tipo IDL;
7. As atribuições de valor podem mapeadas como declaração de constantes em IDL, ou como declaração de operações.

Mapear cada módulo ASN.1 para um módulo IDL;

Dentro de um módulo em GDMO podem ser definidos diversos módulos em ASN.1, cada um destes módulos devem ser transformados para IDL/CORBA. Dentro dos módulos em GDMO que forem transformados para IDL deverá conter uma declaração do tipo `#include "modulo_asn1_transformado"`, para que as definições sejam utilizadas nos módulos em IDL.

Mapear somente as atribuições de tipo e de valores;

Somente as declarações de tipos e valores utilizadas nos módulos em GDMO devem ser transformadas, todas as outras declarações podem ser ignorados.

Resolver os conflitos de nomes entre itens ASN.1;

Um dos principais problemas de conversão entre ASN.1 e IDL é a diferença existente quanto a sua sensibilidade ao uso de maiúsculas e minúsculas. Em ASN.1 o identificador AX é diferente do identificador Ax, que é diferente de ax, que por sua vez é diferente de aX, mas em IDL não existe esta diferença, este problema é resolvido conforme o item 5.2.2.

Criar nomes para tipos IDL definidos a partir de tipos ASN.1 anônimos;

Existe em ASN.1 situações em que uma declaração, especialmente as do tipo SEQUENCE, não realizam a colocação de nomes em alguns de seus componentes, como IDL não suporta este tipo de construção é necessário que seja criado, de alguma forma, um nome IDL para ela. A solução para este problema esta descrita no item 5.2.5.

Ordenar os tipos gerados para evitar a dependência entre tipos;

Em IDL é necessário que todas as declarações de tipos sejam definidas antes de serem utilizadas, em ASN.1 não existe esta necessidade. Para evitar que ocorram erros durante a tradução é necessário que seja feita a ordenação dos tipos criados.

Declarar os tipos importados em módulos ASN.1 como *typedef* do tipo IDL;

Para que as referências aos tipos ASN.1 importados através da cláusula IMPORTS sejam resolvidas, é necessário que nos módulos resultantes exista uma referência a estes tipos. Esta solução é descrita no item 5.2.6.

As atribuições de valor podem mapeadas como declaração de constantes em IDL, ou como declaração de operações.

As declarações de valores em ASN.1 podem ser traduzidas de duas formas para IDL, a primeira forma faz com que as declarações sejam transformadas na definições de constantes em IDL, a segunda faz com que as declarações sejam transformadas em operações IDL. Nossa abordagem trabalha com a primeira forma, pois acredita que os documentos IDL resultantes devem conter o maior número possível de informações sobre os módulos em ASN.1.

5.2.2 Tradução Léxica

A tradução léxica corresponde ao processo de mapeamento dos caracteres e das equivalências de nomes necessárias para o entendimentos dos compiladores IDL utilizados.

Mapeamento do Conjunto de Caracteres ASN.1

Os nomes ASN.1 são sensíveis à maiúsculas e minúsculas, mas os identificadores IDL não. Além disso, ASN.1 tem espaços de nomes diferentes para referências de tipos, identificadores e referências de valores, enquanto que IDL tem um único espaço de nome. Ambos têm espaços de nomeação dentro de um escopo, mas os escopos não são diretamente mapeados, por exemplo, tipos enumerados criam um novo escopo de nome em ASN.1, mas não em IDL. Assim não pode-se simplesmente mapear nomes ASN.1 para identificadores IDL. Para tratar esta dificuldade, o tradutor deve manter uma tabela de todos os identificadores em cada escopo resultante IDL e modificar identificadores em colisão para evitar conflitos.

Nem todos os conflitos são resultantes dos nomes ASN.1, conflitos adicionais podem emergir por causa da colisão com palavras reservadas definidas em arquivos gerados pelo próprio sistema. O tradutor deve considerar estas definições e usar os mesmos mecanismos para evitar as colisões. Todos os identificadores devem ser modificados anteriormente para resolver as ambigüidades, entretanto o objetivo do mecanismo de tradução é gerar o mapeamento mais simples possível, o que conduz a um algoritmo um pouco mais complexo, mas que preserva um mapeamento direto onde nenhuma ambigüidade aparece.

As regras para garantir a não colisão entre diversos identificadores ambíguos são as seguintes:

- o primeiro identificador em ASN.1 é mapeado para si mesmo, isto é, um identificador em IDL onde as maiúsculas e minúsculas são preservadas. O próximo identificador e os subseqüentes, que diferirem somente em maiúsculas e minúsculas serão sufixados com duas sublinhas seguidas por um número para diferenciá-los, isto é, “__n” onde n é a contagem dos identificadores colidentes para maiúsculas e minúsculas. Assim o primeiro identificador não terá um sufixo, o segundo e os identificadores subseqüentes que colidem terão sufixos “__1”, “__2” e assim por diante. Por exemplo, os identificadores ASN.1 aab, aAB e aaB devem ser traduzidos como **aab**, **aAB__1** e **aaB__2** respectivamente.
- a primeira referência de um tipo é mapeado com o sufixo “**Type**”. A segunda referência de tipo e as subseqüentes que diferem somente em maiúsculas e minúsculas serão sufixadas com “**Type**n” onde n é a contagem de tais instâncias. As letras maiúsculas e minúsculas da referência de tipo são sempre preservadas. Por exemplo, as referências de tipo ASN.1 Aab, AAB e AaB serão traduzidas como **AabType**, **AABType1** e **AaBType2** respectivamente.
- referências de valores são manipuladas como identificadores.

Através deste esquema, referências de tipos não colidirão com identificadores ou referências de valores, e estes dois últimos são tornados não ambíguos através do sufixo `<__n>`. Uma vez que o caso mais comum de ocorrência de colisões é entre tipos e instâncias diferenciadas, somente pela primeira letra ser maiúsculas e minúscula, este algoritmo evita a maioria das colisões.

Além disso, colisões podem surgir por causa do processo anterior. Por exemplo, um tipo pode ter o nome *MyData*, e uma variável pode ter o nome *myDataType*. A tradução de *MyData* seria *MyDataType* o que colidiria com a tradução direta de *myDataType*. Este tipo de colisão deve ser resolvido pelas próprias regras de mapeamento. Além disso, traduções de construtores GDMO podem anexar *Unconfirmed*, *Opt* e *Default*, conforme a *template* sendo transformada, e estes podem também ser considerados durante o processo a fim de evitar a ambigüidade durante a conversão.

Mapeamento de Identificadores e Referências ASN.1

Em ASN.1, nomes de referências, identificadores, referências de valores e referências de módulos consistem de uma seqüência arbitrária de uma ou mais letras, dígitos e hífens. As letras e dígitos em nomes ASN.1 são mapeados diretamente permanecendo a sensibilidade a maiúsculas e minúsculas. O hífen ASN.1 ('-') é mapeado para sublinha IDL ('_').

5.2.3 Mapeamento de Primitivas de Elementos ASN.1

Mapeamento de Comentários

Um comentário ASN.1 começa com um par de hífens ("--") e termina com o próximo par de hífens ou o final da linha, sempre o que vier primeiro. Em IDL, comentários são delimitados como em C++, usando-se `/*` e `*/` ou `//` indo até o final da linha.

Mapeamento de Tipos Primitivos ASN.1

A tabela abaixo descreve o mapeamento *default* de tipos ASN.1 básicos para tipos IDL. A idéia básica do algoritmo para mapeamento de tipos é converter todos os tipos para elementos primitivos e em seguida mapear cada tipo primitivo de acordo com esta tabela.

Tipo ASN.1	Tipo IDL
BOOLEAN	<code>typedef boolean ASN1_BOOLEAN;</code>
INTEGER	<code>typedef long ASN1_INTEGER;</code>
REAL	<code>typedef double ASN1_REAL;</code>
ENUMERATED	<code>enum <enumName> {<elem1>,, <elemn>;</code>
BIT STRING	<code>sequence <octet> // conforme codificação BER</code>
OCTET STRING	<code>typedef sequence <octet> ASN1_OctetString;</code>
STRING	<code>sequence <octet> ou string</code>
OBJECT IDENTIFIER	<code>typedef string ASN1_ObjectIdentifier</code>

ANY ANY DEFINED BY the open type	any
Tagged	Como untagged type
EXTERNAL InstanceOf	ASN1_External

Tabela 11 - Mapeamento de tipos primitivos.

Tipo Boolean

O tipo ASN.1 **BOOLEAN** é mapeado diretamente para o tipo IDL **boolean**, entretanto, é fornecida em IDL a definição de tipo (*typedef*) para o tipo **ASN1_BOOLEAN**. Isto é utilizado na tradução para dar melhor correspondência textual ao original ASN.1. Por exemplo:

ASN.1	IDL
Married ::= BOOLEAN maritalStatus Married ::= TRUE	<code>typedef ASN1_BOOLEAN MarriedType;</code> <code>const MarriedType maritalStatus = TRUE;</code>

Tabela 12 - Mapeamento do tipo BOOLEAN.

Tipo Inteiro

O tipo ASN.1 **INTEGER** é mapeado para o tipo IDL **long**, sendo fornecida uma definição de tipo (*typedef*) para o tipo **ASN1_INTEGER**, que é usada na tradução para IDL.

Para um número declarado com um nome, na forma <nome> (<valor>) para um tipo inteiro <tipo> deve ser criada uma declaração de constante em IDL:

```
const <tipo>Type[n] <nome> = <valor>;
```

ASN.1	IDL
T0 ::= INTEGER	<code>typedef ASN1_INTEGER T0Type;</code>
a INTEGER ::= 1	<code>const ASN1_INTEGER a = 1;</code>
T1 ::= INTEGER (a(2))	<code>typedef ASN1_INTEGER T1Type;</code> <code>const T1Type a = 2;</code>
ax INTEGER ::= 1 aX INTEGER ::= 2 T2 ::= INTEGER { a(3) b(aX) } c T2 ::= b d T2 ::= a	<code>const ASN1_INTEGER ax = 1;</code> <code>const ASN1_INTEGER aX__1 = 2;</code> <code>typedef ASN1_INTEGER T2Type;</code> <code>const T2Type a = 3;</code> <code>const T2Type b = aX__1;</code> <code>const T2Type c = b;</code> <code>const T2Type d = a;</code>

Tabela 13 - Mapeamento do tipo INTEGER.

Tipo Real

O tipo ASN.1 **REAL** é mapeado para o tipo IDL **double**, sendo fornecida uma definição de tipo (*typedef*) para o tipo **ASN1_REAL**, que é usada na tradução para IDL. A razão para a seleção do tipo IDL **double** é para fornecer máxima precisão, uma vez que não há meio de se determinar a precisão desejada.

Constantes ASN.1 `PLUS-INFINITY` e `MINUS-INFINITY` são mapeadas para IDL como constantes de ponto flutuante de dupla precisão `PLUS_INFINITY` e `MINUS_INFINITY` respectivamente. Em ASN.1, um valor de ponto flutuante pode ser definido como uma tripla composta pela mantissa, base e expoente. Neste caso, o valor da constante é avaliado e é usada a constante de ponto flutuante decimal de dupla precisão.

ASN.1	IDL
<code>AngleInRadians ::= REAL</code>	<code>typedef ASN1_REAL AngleInRadiansType;</code>
<code>pi REAL ::= {mantissa 3141592653897, base 10, exponent -13}</code>	<code>const ASN1_REAL pi = 3.141592653897;</code>

Tabela 14 - Mapeamento do tipo REAL.

Tipos Enumerados

O tipo ASN.1 `ENUMERATED` é mapeado para o tipo IDL `enum`, conforme mostrado na Tabela 15.

ASN.1	IDL
<code>Message ::= ENUMERATED { basic (0), Extended(1) }</code>	<code>enum MessageType { basic, extended };</code>
<code>DAYOfTheWeek ::= ENUMERATED { sunday(0), monday(1), tuesday(2), wednesday(3), thursday(4), friday(5), saturday(7) }</code> <code>first DAYOfTheWeek ::= Sunday</code>	<code>enum DayOfTheWeekType { sunday, monday, tuesday, wednesday, thursday, friday, saturday };</code> <code>const DayOfTheWeekType first = sunday;</code>
<code>MaritalStatus ::= ENUMERATED { single(0), married, widowed(1) }</code>	<code>enum MaritalStatusType { single, married, widowed };</code>

Tabela 15 - Mapeamento do tipo ENUMERATED.

BIT STRING

O tipo ASN.1 `BIT STRING` é mapeado para o tipo IDL `ASN1_BitString` definido como `sequence<octet>` com um escopo global dentro do arquivo IDL `ASN1Types.idl`. O arquivo `ASN1Types.idl` foi definido pelo X/Open, e se encontra listando nos anexos. O tipo `BitString` é definido semelhante à codificação BER de `BIT STRING`.

Cada bit nomeado é mapeado como uma constante IDL `unsigned long` com o valor igual ao deslocamento dentro do *bit string*. O nome da constante é um nome sem ambigüidade dado pelas regras usuais:

```
const unsigned long <bitname> = <offset>;
```

Note que o deslocamento pode ser definido por outra constante.

Para facilitar a manipulação de valores *BitString*, é proposta a seguinte interface PIDL (Pseudo IDL). Esta interface define essencialmente a biblioteca de funções utilitárias de acesso a *BitString* para padronizar e simplificar construções de aplicações. [OPS95]

```
interface ASN1_BitStringHandler          // PIDL
{
typedef short BitValue;

ASN1_BitString create_bit_string (in unsigned long number_of_bits);

BitValue get_bit_value (in ASN1_BitString bit_string, in unsigned long
index);

void set_bit_value (in ASN1_BitString bit_string, in unsigned long
index, in BitValue new_bit_value);

long bit_string_length (in ASN1_BitString bit_string);

string get_bit_string (in ASN1_BitString bit_string);
// produz uma string com valores binários ("1001011B")

void set_bit_string (in ASN1_BitString bit_string, in string
string_value);
};
```

ASN.1	IDL
MessageFlag ::= BIT STRING { posResp (0), negResp (1), doNotForward (2) }	typedef ASN1_BitString MessageFlagType; const unsigned long posResp = 0; const unsigned long negResp = 1; const unsigned long doNotForward = 2;
T0 ::= BIT STRING a INTEGER ::= 1 T1 ::= INTEGER { a(2) } T2 ::= BIT STRING { a(3) b(a) }	typedef ASN1_BitString T0Type; const ASN1_INTEGER a = 1; typedef ASN1_INTEGER T1Type; const T1Type a__1 = 2; typedef ASN1_BitString T2Type; const unsigned long a__2 = 3; const unsigned long b = a;
G3FacsimilePage ::= BIT STRING -- uma sequência de bits de acordo -- com Recomendação T.4 image G3FacsimilePage ::= '100110100100001110110U' trailer BIT STRING ::= '0123456789ABCDEFU'	typedef ASN1_BitString G3FacsimilePageType; interface ConstValues { G3FacsimilePageType image(); // " '100110100100001110110' B"; BitString trailer(); // " '0123456789ABCDEF' H"; }

Tabela 16 - Mapeamento do tipo BIT STRING.

Tipo Octeto de String

O tipo ASN.1 OCTET STRING é mapeado para o tipo IDL **ASN1_OctetString** que é definido como **sequence<octet>**.

Literais OCTET STRING podem ser *strings* binárias ou hexadecimais. O valor é expandido da esquerda para direita como uma sequência de bits com octetos ordenados da esquerda para a direita. O octeto final é preenchido com bits 0 se necessário.

ASN.1	IDL
G4FacsimilePage ::= OCTET STRING -- uma sequência de octetos conforme -- as recomendações T.5 e T.6 image G4FacsimilePage ::= '3FE2EABAD471005'H	<pre>typedef ASN1_OctetString G4FacsimilePageType; interface ConstValues { G4FacsimilePageType Image(); // ``3FE2EABADA471005'H''; };</pre>

Tabela 17 - Mapeamento do tipo OCTET STRING.

STRING

O tipo ASN.1 STRING é definido com uma variedade de opções. Este tipo pode ser dividido em duas categorias: aqueles tipos que podem conter o valor X'00, e aqueles que não podem. Além disso, estas duas categorias podem estar divididas dentro daquelas que são caracteres de byte únicos, mapeados como **sequence<octet>** e aqueles que suportam caracteres extensos, mapeados como **sequence** de tipos extensos. Os tipos *string* que não podem incluir X'00 são simplesmente mapeados como *string*. O arquivo IDL ASN1Types.idl contém as seguintes definições que podem ser usadas diretamente na tradução IDL.

```
// Estas podem conter X'00'
typedef sequence<octet> ASN1_GeneralString;
typedef sequence<octet> ASN1_GraphicString;
typedef sequence<octet> ASN1_ISO646String;
typedef sequence<octet> ASN1_IA5String;
// Estas devem suportar caracteres extensos
typedef sequence<long> ASN1_UniversalString;
typedef sequence<unsigned short> ASN1_BMPString;
// Estas não devem suportar
typedef string ASN1_NumericString;
typedef string ASN1_PrintableString;
typedef string ASN1_TeletexString;
typedef string ASN1_T61String;
typedef string ASN1_ViedeotexString;
typedef string ASN1_VisibleString;
```

ASN.1 fornece um conjunto de sub-tipos de *strings* úteis. A principal diferença entre estes tipos e seus tipos básicos é que eles têm rótulos bem definidos. Uma vez que ignoramos a informação rótulo na Tradução de Especificações, estes tipos são definidos pelas seguintes definições de tipo em IDL :

```
typedef ASN1_VisibleString ASN1_GeneralizedTime;
typedef ASN1_VisibleString ASN1_UTCTime;
typedef ASN1_GraphicString ASN1_ObjectDescriptor;
```

As definições de tipo em IDL *GeneralizedTime* e *UTCTime*, são representações de valores de tempo formatadas em *string*. Como *strings*, estes valores não são fáceis de se manipular, como por exemplo quando são necessárias comparações. Por esta razão são fornecidas funções PIDL para manipular estes valores convertê-los para uma forma mais útil, fazendo com que correspondam a estrutura POSIX *timeval*.

Estas funções fornecem rotinas de conversão entre *string* e as estruturas *Timeval* e também uma rotina de comparação. Esta rotina obtém dois valores, *t1* e *t2*, e retorna -1, 0 ou 1 dependendo se *t1* < *t2*, *t1* = *t2* ou *t1* > *t2* respectivamente.

```
#include "ASN1Types.idl"
struct Timeval {
    long seconds;
    long usecs;
    int tzp;
};

interface ASN1_GeneralizedTimeHandler // PIDL
{
    Timeval string_to_struct(in ASN1_GeneralizedTime t);
    ASN1_GeneralizedTime struct_to_string(in Timeval t);
    long compare(in ASN1_GeneralizedTime t1, in ASN1_GeneralizedTime
t2);
};

interface ASN1_UTCTimeHandler // PIDL
{
    Timeval String_to_struct(in ASN1_UTCTime t);
    ASN1_UTCTime struct_to_string(in Timeval t);
    long compare(in ASN1_UTCTime t1, in ASN1_UTCTime t2);
};
```

IDENTIFICADOR DE OBJETO

O tipo ASN.1 OBJECT IDENTIFIER é mapeado para o tipo IDL **ASN1_ObjectIdentifier** que é definido como um **string**:

```
typedef string ASN1_ObjectIdentifier;
```

As constantes do tipo `OBJECT IDENTIFIER` são representadas em IDL como *string* literais.

ASN.1	IDL
Attributeld ::= OBJECT IDENTIFIER	<code>typedef ASN1_ObjectIdentifier AttributeIdType</code>
smi2MObjectClass OBJECT IDENTIFIER ::= { joint-iso-ccitt ms(9) smi(3) part2(2) managedObjectClass(3) }	<code>const ASN1_ObjectIdentifier smi2MObjectClass = "2.9.3.2.3";</code>

Tabela 18 - Mapeamento do tipo `OBJECT IDENTIFIER`.

Tipo ANY

Ambos os tipos ASN.1 `ANY` e `ANY DEFINED BY` são mapeados para o tipo IDL **any**. Isto porque não há meio de capturar a informação definida que está disponível em `ANY DEFINED BY`.

ASN.1	IDL
Attribute ::= ANY DEFINED BY attributeld;	<code>typedef any AttributeType; // definido por attributeld</code>

Tabela 19 - Mapeamento do tipo `ANY`.

Tipos Rotulados

A informação de rótulo (*tag*) é ignorada durante a Tradução de Especificações e é mapeada do mesmo modo que um tipo sem rótulo. O mapeamento da informação de rótulo é uma questão para a Tradução Interativa.

Tipos Externos

O tipo ASN.1 `EXTERNAL` é mapeado para a seguinte estrutura IDL:

```
struct ASN1_External {
    ASN1_ObjectIdentifier abstractSyntaxName;
    any data_value;
}
```

O mapeamento requer que o *gateway*, solucione totalmente a forma de codificação do valor *External*, que ele recebe, e recodificando-o usando o tipo CORBA **any**. Isto quer dizer que nenhum dos objetos CORBA necessita suportar BER (*Basic Encoding Rules*) para manipular este tipo.

5.2.4. Mapeamento de Valores não Representáveis em Constantes IDL

A IDL CORBA permite somente constantes de tipos primitivos. Desta forma, ela é incapaz de representar adequadamente valores constantes de tipos mais complexos. É importante uma passagem segura para as constantes usadas em GDMO para algo equivalente a valores

default. Para todos os valores constantes que não podem ser representados em IDL CORBA, é definido uma interface PIDL especial de valor constante, no mesmo módulo que contém a declaração *const*. A implementação desta interface de valor constante é responsável por garantir que aqueles valores retornados correspondam a declaração ASN.1/GDMO, mas estas rotinas podem ser automaticamente geradas como parte do processo de tradução. Em geral, a declaração de constante `<ConstType>` com o identificador `<constName>` deve resultar na geração da operação:

```
<ConstType> <ConstName>();  
// retorna "<texto da definição do valor constante>"
```

5.2.5 Mapeamento de Tipos Construtores ASN.1

Os tipos compostos são mapeados de acordo com a tabela abaixo, mas existe uma complexidade adicional que deve ser considerada. Tipos anônimos devem ser mapeados para um tipo nomeado, que é então utilizado. Alguns construtores ASN.1 devem ser resolvidos para pares nome-tipo e então traduzidos.

ASN.1	IDL	Comentários
CHOICE	union/switch	Um <i>enum</i> é para ser definido como <i>switch case-constant</i> .
SELECTION	mapeado como um tipo do elemento selecionado	
SET, SEQUENCE	struct	membros de <i>struct</i> são declarados na mesma ordem que são declarados no tipo ASN.1.
OPTIONAL	union <type>Opt switch (boolean) {case TRUE: <type> value;};	
SET-OF type, SEQUENCE-OF type	sequence<type>	

Tabela 20 - Mapeamento de tipos construtores ASN.1

Tipos Anônimos

Em ASN.1 são permitidos dois mecanismos para se especificar os tipos membros dos tipos construtores SEQUENCE, SET, SEQUENCE OF, SET OF e CHOICE:

1. o uso de referências de tipo para tipos definidos em outro local ou para tipos primitivos.
2. a definição de tipos anônimos.

Um tipo anônimo é definido diretamente dentro do escopo do seu construtor de tipo incluso, entretanto ele não possui um nome. A IDL CORBA não suporta construtores similares, entretanto o mapeamento precisa solucionar estas questões.

Um membro sem um identificador de membro explícito é mapeado para um membro com um identificador, como por exemplo, `item<n>`, onde `n` especifica a posição dentro do construtor.

Cada definição de um tipo anônimo é convertida para a definição padrão do tipo IDL CORBA.

ASN.1	IDL
<pre>Bar ::= SEQUENCE { -- definição de um tipo anônimo paff SEQUENCE { a INTEGER, b VisibleString }, -- referência ao tipo primitivo, -- sem identificador de membro INTEGER }</pre>	<pre>struct BarPaffType { ASN1_INTEGER a; ASN1_VisibleString b; }; struct BarType { BarPaffType paff; ASN1_INTEGER item2; };</pre>

Tabela 21 - Mapeamento do tipo SEQUENCE.

Mapeamento do Tipo CHOICE

O tipo ASN.1 **CHOICE** é mapeado para o tipo IDL **union**. Uma vez que uniões IDL são discriminadas, um tipo IDL **enum** será definido para o tipo do discriminador. Para se fazer isto, deve-se primeiro transformar os tipos anônimos ASN.1 existentes nos tipos IDL correspondentes.

O identificador IDL para o tipo do discriminador é formado pela adição de um sufixo “Choice” para o identificador do tipo CHOICE traduzido. Para cada tipo substituto, haverá um identificador para o tipo discriminante, que é o discriminador do tipo substituto sufixado por “Choice” e não ambíguo, dentro de seu escopo em IDL.

Um tipo **união** (*union*) é então construído, definindo-se um caso para cada substituto, com o label de acordo com o tipo enumerado, e o tipo e identificador de acordo com o substituto. O resultado em IDL será como mostrado abaixo:

```
enum <choicetype>Choice {
  <identifier>Choice,
  .....// um para cada alternado
}

union <choicetype> switch (<choicetype>Choice) {
  case <identifier>Choice: <type> <identifier>;
  .....// um para cada alternado
}
```


Mapeamento de SELECTION

O construtor ASN.1 SELECTION fornece um mecanismo onde um tipo pode ser definido pela referência a um membro nomeado de CHOICE. Quando o tipo SELECTION é usado para identificar um tipo, ele é resolvido em IDL somente para o tipo do elemento selecionado.

ASN.1	IDL
Attribute ::= CHOICE { number INTEGER name VisibleString }	<pre>// mapeia o tipo Attribute enum AttributeTypeChoice { numberChoice, nameChoice, }; union AttributeType switch (AttributeTypeChoice) { case numberChoice:ASN1_INTEGER number; case nameChoice: ASN1_VisibleString name; };</pre>

Tabela 22 - Mapeamento do tipo CHOICE

Mapeamento de SEQUENCE e SET

Em ASN.1 um tipo SEQUENCE é uma coleção ordenada de componentes que são tipados e nomeados. Os tipos SET em ASN.1 são idênticos em IDL CORBA, exceto que seus membros não estão ordenados. Uma vez que em IDL CORBA não existe o tipo registro ordenado, ambos os tipos são manipulados de forma idêntica e são mapeados para uma estrutura IDL CORBA.

Cláusula COMPONENTS OF TYPE

Quando um tipo seqüência ou conjunto aparece como <type> em COMPONENTS OF <type>, aquele tipo seqüência é resolvido para sua lista de membros antes da tradução para IDL.

Componentes OPCIONAIS

Se um membro de um tipo SEQUENCE ou SET é declarado como OPTIONAL, ele pode ou não estar presente. Isto é mapeado para uma **union** IDL CORBA com um *switch* booleano que somente será verdadeiro se possuir um valor que possibilita o elemento opcional. O novo tipo é nomeado pela junção do tipo do elemento opcional com “*Opt*”. O tradutor deve seguir os passos para evitar duplicação destes tipos. Este tipo opcional, deve ser gerado mais de uma vez por módulo.

Para a cláusula ASN.1 :

```
SEQUENCE { <identifier> <type> OPTIONAL }
```

o resultado IDL deve ser :

```
union <type>Opt switch (boolean) {case True: <type> value;};
```

Componentes DEFAULT

Em ASN.1 ocorre um DEFAULT, quando a omissão do valor de um tipo é exatamente equivalente à inserção de um valor *default*. Não há um tipo equivalente direto em IDL, uma vez que se fornece somente acesso ao valor *default*, ou como uma constante ou via uma função PIDL que retorne o valor, que deve ser usada quando um programador deseja usar o valor *default* em domínio CORBA. Se um valor DEFAULT ASN.1 está presente para um membro de conjunto ou para uma seqüência, uma constante IDL será gerada, nomeada pelo anexo “Default” para o identificador do membro e definida como um valor *default* dado.

```
const <type> member-name>Default = <value>;
```

Tradução para IDL

Uma vez que todas as cláusulas COMPONENTS OF, OPTIONAL, DEFAULT, e SELECT são resolvidas, uma **struct** IDL é gerada. Cada componente é mapeado como um membro da estrutura IDL, na ordem em que eles aparecem na lista de membros resolvidos. Se o membro não possuir identificador, então o identificador “Itemn” é usado, onde n é inicialmente 1 e incrementado posteriormente para cada membro sem nome da estrutura.

Exemplos:

ASN.1	IDL
T ::= SEQUENCE{ a Ta, b Tb, c Tc } E ::= SEQUENCE{f1 E1, f2 E2, f3 E3}	<pre>struct TType{TaType a;TbType b;TType c;}; struct Etype {E1Type f1;TType f2;E3Type f3;};</pre>
T ::= SEQUENCE{ a Ta, b SEQUENCE {b1 T1, b2 T2, b2 T3} c Tc } W ::= SEQUENCE {x Wx, COMPONENTS OF T, y, Wy}	<pre>struct T_bType { T1Type b1;T2Type b2; T3Type b3;}; struct Ttype { TaType a; T_bType b; TcType c;}; struct Wtype { WxType x; TaType a; T_bType b; TcType c; WyType y; };</pre>
UserName ::= SET { -- conjunto tratado como uma SEQÜÊNCIA personalName VisibleString, countryName VisibleString OPTIONAL }	<pre>union ASN1_VisibleStringOpt switch (boolean) { case True:ASN1_VisibleString value; }; struct UserName {</pre>

	<pre>ASN1_VisibleString personalName, ASN1_VisibleStringOpt countryName, };</pre>
<pre>A ::= SET { a ENUMERATED {a, b}, b ENUMERATED {a, b}, }</pre>	<pre>enum A_aType {a, b}; enum A_bType {a__1, b__1}; struct aType { A_aType a; A_bType b; };</pre>
<pre>Data ::= SEQUENCE { ReplaceWithDefault BOOLEAN DEFAULT FALSE, defaultValue ValueSpecifier, keyword SEQUENCE { type-reference DefinedType, field Identifier } OPTIONAL, createModifier BIT STRING { withRefObject (0),withAutoNaming(1) } }</pre>	<pre>const ASN1_BOOLEAN replaceWithDefaultDefault = FALSE struct DataKeywordType { DefinedTypeType type_reference, IdentifierType field, }; union DataKeywordTypeOpt switch (boolean) { case True: DataKeywordType value; } typedef BitString DataCreateModifierType; const unsigned long withRefObject = 0; const unsigned long with AutoNaming = 1; struct Data { ASN1_BOOLEAN replaceWithDefault; ValueSpecifierType defaultValue; DataKeywordTypeOpt keyword; DataCreateModifierType createModifier; };</pre>

Tabela 23 - Mapeamento do tipo SEQUENCE.

Mapeamento de SEQUENCE OF e SET OF

Em ASN.1 um tipo SEQUENCE é uma lista ordenada de elementos do mesmo tipo. SET OF é similar exceto que a lista é não ordenada. Ambos são mapeados sobre o construtor IDL `sequence`.

Exemplos

ASN.1	IDL
RDNSequence ::= SEQUENCE OF RDN	<pre>typedef sequence<RDNTYPE>RDNSequenceType ;</pre>
Status ::= SEQUENCE OF INTEGER <pre>{ initializationRequired (0), notInitialized (1), initializing (2), reporting (3), terminating (4) }</pre> -- tradução procede via o novo -- tipo nomeado: StatusItem	<pre>typedef ASN1_INTEGER StatusTypeItem; const StatusTypeItem InitializationRequired = 0; const StatusTypeItem notinitialized = 1; const StatusTypeItem Initializing = 2; const StatusTypeItem reporting = 3; const StatusTypeItem terminationong = 4; typedef sequence<StatusTypeItem> StatusType;</pre>

A ::= SEQUENCE OF SEQUENCE OF INTEGER	typedef sequence<ASN1_INTEGER> AitemType; typedef sequence<AitemType> Atype;
--	---

Tabela 24 - Mapeamento do tipo SEQUENCE OF.

5.2.6 Mapeamento de Definições de Módulos ASN.1 para IDL

Este mapeamento é feito de acordo com as regras de produção da tabela abaixo, explicadas a seguir:

```

<ModuleDefinition> ::=
  <ModuleIdentifier> DEFINITIONS <TagDefault> ::= BEGIN
    <ModuleBody>
  END

<ModuleIdentifier> ::=
  <modulereference>
  <DefinitiveIdentifier>

<TagDefault> ::= EXPLICIT TAGS | IMPLICIT TAGS |
  AUTOMATIC TAGS | empty

<ModuleBody> ::=
  <Exports> <Imports> <AssignmentList> | empty

<Exports> ::=
  EXPORTS <SymbolsList> ; | empty

<Imports> ::=
  IMPORTS <SymbolsImported> ; | empty

<SymbolsImported> ::=
  <SymbolsFromModuleList> | empty

<SymbolsFromModuleList> ::=
  <SymbolsFromModule> | <SymbolsFromModuleList> <SymbolsFromModule>

<SymbolsFromModule> ::=
  <SymbolList> FROM <GlobalModuleReference>

<GlobalModuleReference> ::=
  <modulereference> AssignedIdentifier>

<SymbolList> ::=
  <Symbol> | <SymbolList>, <Symbol>

AssignmentList ::=
  Assignment | AssignmentList Assignment

```

Um identificador de módulo (*ModuleIdentifier*) é mapeado para o mesmo nome do módulo IDL correspondente.

Cláusulas `IMPORTS` são mapeadas como uma lista de diretivas *#include* para o arquivo correspondente ao módulo importado. Este arquivo é nomeado de acordo com o apelido do módulo. Estas referências relevantes habilitam o módulo a não possuir ambigüidade, quando seu apelido é identificado. Isto também deve ser notado, para que o tradutor reconheça qualquer documento importado, a fim de aplicar as regras de não ambigüidade aos identificadores coincidentes.

A produção de *SymbolsFromModule* com valor:

```
<symbol1>, ..., <symboln> FROM <documentIdentifier>::<moduleRefence>
```

é mapeada da seguinte forma:

```
typedef <moduleNickname>::<symbol1> <symbol1>
typedef <moduleNickName>::<symboln> <symboln>
```

O termo `<documentIdentifier>` é usado somente para identificar o apelido único do módulo, e não aparece em IDL. Quando identificadores de módulos externos são referenciados, eles devem ser mapeados como identificadores com um escopo em IDL, onde o escopo é *moduleNickname* (apelido do módulo), uma vez que estes são únicos no ambiente de tradução, nenhum escopo a mais é desejado. Em particular o apelido do documento não é necessário, e o identificador do documento é necessário somente para identificar o apelido correto do módulo.

5.2.7 Mapeamento de Sub-Tipos

Devido ao fato da ASN.1 ser mais poderosa que IDL, subtipos, em particular, não são tão fáceis de se mapear em IDL. A verificação de outras faixas de valores e outros detalhes, está além da preocupação da conversão de uma implementação.

Mapeamento do Tipo *Constrained*

Em ASN.1 é possível definir um tamanho limitado para as construções `SET OF` e `SEQUENCE OF`. Quando um tamanho limitado é usado para estas construções são mapeadas para IDL como um **sequence** restrito, onde o limite superior da sequência IDL é determinado pelo limite superior sobre os valores inteiros permitidos. O resultado da definição é como segue abaixo:

```
typedef sequence <<ItemType>, <upper-bound>> <type>;
```

onde `<ItemType>` é o tipo do item `SEQUENCE OF/SET OF` e `<type>` é o nome traduzido do tipo componente. Por exemplo:

```
T1 ::= SEQUENCE (SIZE(0..10)) OF ObjectInstance
T2 ::= SEQUENCE (SIZE(1|3|5)) OF INTEGER
```

é mapeado para :

```
typedef sequence<ObjectInstanceType, 10> T1Type;
```

```
// T1Type SIZE(0..10)
typedef sequence<ASN1_INTEGER, 5> T2Type;
// T2Type SIZE(1 | 3 | 5)
```

Mapeamento de Valores Simples

Ainda que um tipo seja limitado a um ou mais valores simples, ele ainda será um tipo e será traduzido com as regras definidas para este tipo. O conjunto de valores permitidos é traduzido como um comentário.

```
A ::= INTEGER (1|3|5|7)
```

deverá ser traduzido para

```
typedef ASN1_INTEGER Atype;
// Atype (1|3|5|7)
```

5.3 Tradução de Classes e Módulos GDMO para Interfaces IDL

Neste item descrevemos o processo de tradução de classes e módulos GDMO para interfaces e módulos IDL/CORBA. Apresentando principalmente o mecanismo criado para possibilitar, ao implementador dos processos de gerência, uma ferramenta capaz de ser utilizada para facilitar o processo da implantação da tradução interativa.

Entende-se por tradução interativa ao processo em que as interações de um domínio são mapeadas para uma ou mais interações em outro domínio. Este processo de transformação pode ser representado por um *gateway* ou por um agente-*proxy* [GEN96][BBAN97][SOU95c], que é o elemento responsável pela tradução interativa. Em nosso mecanismo de tradução, que cobre apenas a tradução das especificações dos domínios, existe a possibilidade de o implementador construir uma estrutura específica para o *gateway*, através do emprego de uma linguagem *script*. A estrutura do *gateway* poderá ser desenvolvida conforme a necessidade do implementador, e principalmente conforme a especificação que ele escolher, seja o modelo definido pelo X/Open [OPS95], Soukouti [SOU95c] ou Ferrasa [FER98].

O processo de conversão é implementado através de um tradutor que opera sobre um conjunto de arquivos de entrada e resulta em alguns arquivos de saída. Como as definições IDL/CORBA são processadas em termos de arquivos que determinam sua granularidade e reusabilidade, é necessário especificar que definições serão geradas e que arquivos serão definidos. Além disso, GDMO coloca algumas complexidades, como nomes completos de textos. E como os nomes são utilizados para especificar partes importantes, eles devem ser transformados para acessar os arquivos que contenham essas especificações. É também

necessário ser capaz de associar os arquivos resultantes em IDL/CORBA com os originais em GDMO, permitindo consultas e reúso de especificações. Isto pode é feito pelo uso de um banco de dados de apelidos (*nicknames*), que mapeiam um único nome registrado de um documento GDMO para um apelido curto apropriado para uso como um nome de arquivo base. O apelido pode ser utilizado para encontrar arquivos importantes e controlar os nomes de arquivos IDL/CORBA gerados.

Como os apelidos podem ser usados como base para nomear os arquivos IDL/CORBA gerados, estes podem somente ser reutilizados em ambientes onde o mesmo banco de apelidos está disponível. É desejável que o banco de apelidos seja o mais padronizado possível, ou seja, tenha apelidos específicos para todos os documentos GDMO registrados, e também para os módulos ASN.1 relacionados. Em nosso caso adotamos as três primeira letras da especificação em GDMO como sendo o apelido do módulo sendo transformado.

A principal abordagem do tradutor de GDMO é a representação intermediária das *templates* analisadas do arquivo de entrada. Esta representação, exemplificada na Figura 37, é mostrada como uma classe³, para possibilitar a associação entre as informações obtidas das *templates* GDMO, conforme o relacionamento mostrado na Figura 24. Nesta classe, que é a principal para o tradutor, observamos a declaração dos relacionamentos obtidos das *templates* existentes no arquivo GDMO de entrada para o tradutor. Estes relacionamentos são tratados, internamente ao tradutor, como uma lista encadeada de outros objetos.

```

CLASS GDMO
  this                OBJECT
  ManagedObjectList   LIST OF OBJECT
  PackageList         LIST OF OBJECT
  ConditionalPackageList LIST OF OBJECT
  AttributeList       LIST OF OBJECT
  ActionList          LIST OF OBJECT
  NotificationList    LIST OF OBJECT
  ParameterList       LIST OF OBJECT
  NamebindingsList    LIST OF OBJECT
  AttributeGroupList  LIST OF OBJECT
  BehaviourList       LIST OF OBJECT
END

```

Figura 37 - Representação de dados interna do tradutor GDMO.

A declaração de `ManagedObjectList` é apresentada como uma lista de objetos, ou seja, uma lista de elementos da classe `GDMOClass`. Nesta lista estão contidos todas as declarações de *templates* MANAGED OBJECT CLASS existentes no arquivo em GDMO dado como entrada ao processo de transformação.

³ Para o compilador, esta classe é internamente representada como um registro, ou uma *struct* em C/C++.

Outros elementos relacionados nesta classe são: a lista de *templates* PACKAGES, a lista de PACKAGES condicionais, a lista das *templates* de atributos, a lista das *templates* ACTION, as *templates* declaradas para as notificações, a lista das *templates* PARAMETER, as *templates* referentes ao NAME BINDING, as *templates* dos grupos de atributos e as listas das declarações BEHAVIOUR.

Os elementos são declarados como LIST OF, pois fazem referência à outras classes, que possuem as informações referentes aos elementos realmente declarados, como `PackageList`, que possui a relação das *templates* PACKAGE declaradas no arquivo de entrada em GDMO.

Para dar ao tradutor acesso as informações dos módulos é utilizada uma linguagem *script*, conforme definida no capítulo 6. A linguagem é utilizada para servir de ferramenta de acesso as informações definidas nas estruturas de dados do tradutor, estas estruturas estão relacionadas no anexo 2. A linguagem possui alguns comandos de iteração, como WHILE e FOREACH, comando de decisão, como IF, possibilita também a declaração de variáveis. É através desta linguagem que o implementador do mecanismo da tradução interativa, poderá recuperar as informações dos módulos em GDMO para desenvolver o *gateway*. Maiores detalhes de como a linguagem poderá ser utilizada será fornecido no capítulo 6.

A entrada para a tradução da especificação é um arquivo que contenha um documento GDMO para ser transformado e um banco de apelidos. A primeira saída é um conjunto de arquivos que contém as definições IDL CORBA que resultarão da transformação. A Figura 38 ilustra o processo de tradução e os seus arquivos de saída.

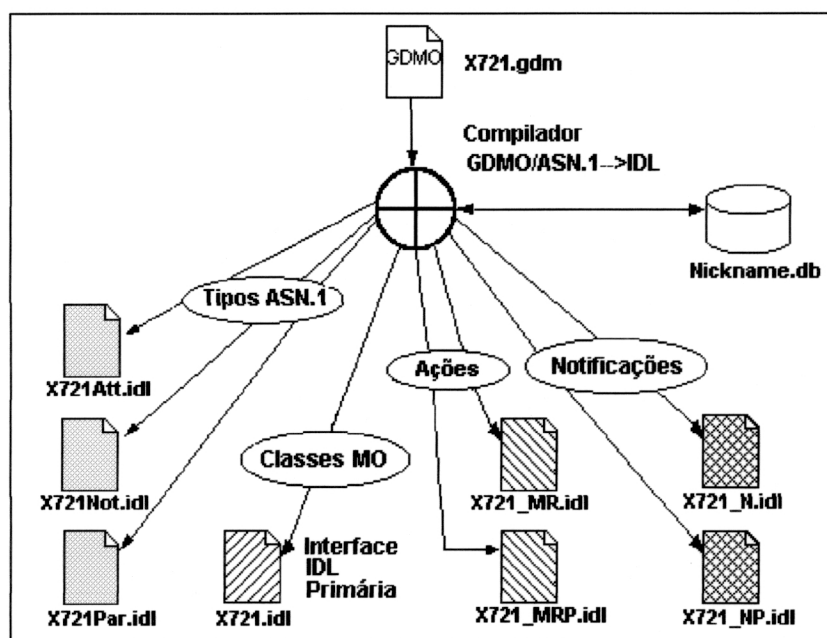


Figura 38 - Organograma do processo de transformação.

5.3.1 Especificação do Algoritmo de Transformação

1. Gerar um arquivo IDL/CORBA para cada módulo ASN.1 que está contido no documento GDMO, e nomeá-lo com um apelido que foi atribuído ao módulo;
2. Mapear cada módulo ASN.1 que está contido no dado documento GDMO para um módulo IDL/CORBA, conforme visto no item 5.2.1;
3. Gerar um arquivo IDL/CORBA para cada documento GDMO, e atribuir a ele o apelido destinado ao documento;
4. Gerar dois arquivos IDL/CORBA que contenham interfaces para manipulação de respostas múltiplas através de modelos *push* e/ou *pull*, conforme se torne necessário;
5. Gerar dois arquivos IDL/CORBA que contenham interfaces para manipulação de notificações através de modelos *push* e/ou *pull*, conforme se torne necessário.

5.3.2 Nomes de Arquivos e Módulos IDL CORBA

A saída de uma transformação é um conjunto de módulos e interfaces IDL CORBA. Estes devem ser organizados dentro de arquivos de forma a facilitar o reuso e a efetiva geração de código pelo tradutor IDL CORBA. Durante o processo de transformação, as regras abaixo determinam o número e o conteúdo de cada arquivo:

1. Cada arquivo começará com um comentário identificando o arquivo GDMO do qual ele foi gerado.
2. Definições contidas em arquivos IDL CORBA gerados devem estar dentro de diretivas **#ifndef ... #endif** que representam os mapeamentos das definições de moldes (*templates*) GDMO, evitando assim redundância de nomes.
3. Para cada documento GDMO/ASN.1, gerar um arquivo IDL nomeado **<apelido>.idl** contendo um módulo IDL CORBA chamado **<apelido>**, dois arquivos chamados **<apelido>_MR.idl** e **<apelido>_MRP.idl** que contenham as interface que suportam respostas múltiplas e dois arquivos chamados **<apelido>_N.idl** e **<apelido>_NR.idl** que contenham as interfaces para notificações.
4. Para cada módulo ASN.1 que contenha um documento ASN.1/GDMO, gerar um arquivo IDL chamado **<módulo_apelido>.idl**, contendo um módulo IDL CORBA chamado **<módulo_apelido>**.
5. Para cada módulo ASN.1 que tenha símbolos importados, adicionar: **#include “<módulo_apelido>.idl”**, para o arquivo IDL CORBA correspondente.

Um ponto negativo desta aplicação é a grande quantidade de arquivos que serão criados, entretanto, os arquivos <_MR>, <_MRP>, <_N> e <_NP> são criados conforme a necessidade da transformação, pois nem todos os módulos exigem aplicações com respostas múltiplas ou emissão de notificações. Este tipo de transformação entretanto pode se beneficiar do compilador IDL/CORBA, devido a granularidade das aplicações que serão geradas, mantendo apenas os módulos necessários para uma compilação relacionados quando necessário. Um outro efeito indesejado, será a colisão de nomes de módulos em ASN.1, pois estes são muitas vezes referenciados apenas como uma sequência de caracteres bem longa. Para evitar isto é feita a utilização de um banco de dados com os apelidos (*nicknames*) dos módulos já transformados, conforme ilustrado na Figura 38.

Pela especificação do X/Open [OPS95] é necessária a existência de alguns arquivos que serão utilizados pela especificação convertida. Estes arquivos contêm as definições dos tipos de dados básicos, utilizados pelo processo de transformação. Os arquivos, listados nos anexos, são:

ASN1Types.idl	Contêm os tipos bases para transformação dos tipos ASN.1 para IDL;
CMISError.idl	Contêm a relação dos erros CMIP e ROSE;
Notification.idl	Contêm as definições para suporte aos mecanismos de notificações;
MultipleReply.idl	Contêm as definições para suporte aos mecanismos de múltiplas respostas;

Para toda classe de objeto gerenciado em GDMO, uma interface IDL/CORBA é definida para suportar as operações definidas na classe. Adicionalmente duas outras interfaces podem ser definidas, para suporte à notificações e para suporte à múltiplas respostas. Um detalhe definido em [OPS95] é como será feita a definição da herança da primeira interface IDL/CORBA, definida através dos módulos GDMO. Seja a hierarquia de classes definida na Figura 39, a interface *ManagedObject* será definida conforme a necessidade da implementação, pois esta é uma classe de uso em *run-time*, ou seja, na tradução interativa, e é possível implementá-la com os métodos necessários para os mecanismos de filtragem CMIP [OPS95]. A classes base de todas as classes de objetos gerenciados é *ManagedObject*, que herda da classe *Object*, como todas as outras interfaces IDL.

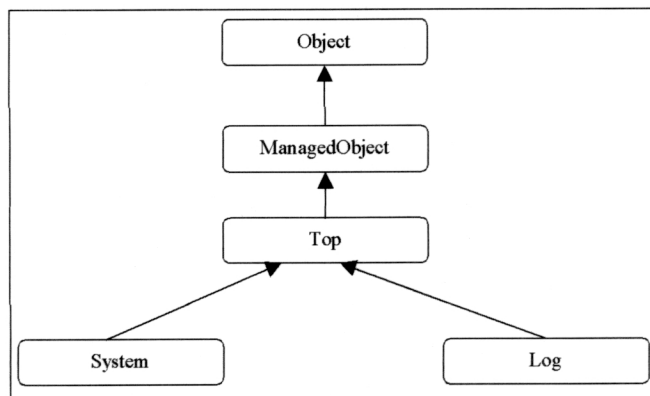


Figura 39 - Hierarquia de Herança em IDL/CORBA.

5.3.3 Mapeamento de MANAGED OBJECT CLASS para IDL/CORBA

A *template* MANAGED OBJECT CLASS, definida como mostra a Figura 40, é mapeada para IDL/CORBA conforme a Tabela 25. Neste processo de mapeamento, as informações mais importantes são o nome da classe e os nomes das superclasses desta classe. O mapeamento consiste na transformação do nome da classe GDMO em um nome de interface em IDL/CORBA, além de ser necessário uma verificação do nome para não ocorrer um conflito com outras interfaces já transformadas. Isto é feito através da utilização dos nomes existentes no arquivo de apelidos (*nicknames*), definido conforme item 5.3.

```

<class-label> MANAGED OBJECT CLASS
  DERIVED FROM <class-label> [, <class-label>]*;
  CHARACTERIZED BY <package-label> [, <package-label>]* ;
  CONDITIONAL PACKAGES <package-label>
    PRESENT IF condition-definition
    [, <package-label> PRESENT IF condition-definition ]*;
  REGISTERED AS object-identifier;
  
```

Figura 40 - Estrutura da *template* MANAGED OBJECT CLASS

A Tabela 25 apresenta as informações a serem tratadas da *template* MANAGED OBJECT CLASS. O elemento especificado em **REGISTERED AS** representa uma informação necessária apenas ao implementador da tradução interativa, quando da referência à uma classe de objeto específica, instanciada através de um OID determinado por esta cláusula.

Elemento	Comentário
MANAGED OBJECT CLASS	Define o nome da interface a ser gerada.
DERIVED FROM	Define as superclasses da interface a ser gerada.
CHARACTERIZED BY	Relação das diversas operações definidas para a interface, traduzidas conforme o mapeamento da <i>template</i> PACKAGE.
CONDITIONAL PACKAGES	Relação das diversas operações definidas para a interface, em caráter condicional, traduzidas conforme o mapeamento da <i>template</i> PACKAGE.
REGISTERED AS	Utilizado para a tradução interativa.

Tabela 25 - Tradução da *template* MANAGED OBJECT CLASS.

A *template* MANAGED OBJECT CLASS é representada internamente pelo tradutor através da estrutura definida na Figura 41. Esta estrutura, apresentada como uma classe (CLASS), é utilizada pelo tradutor para representar as informações obtidas ao analisar as *templates* contidas dentro do módulo em GDMO. Esta estrutura representa as informações conforme descrito na Tabela 26.

CLASS GDMOClass	
this	OBJECT
MOName	STRING
ClassName	STRING
IDLFileName	STRING
ParentList	LIST OF OBJECT
AttributeList	LIST OF OBJECT
AttributeGroupList	LIST OF OBJECT
NamebindingsList	LIST OF OBJECT
ActionList	LIST OF OBJECT
NotificationList	LIST OF OBJECT
Oid	STRING
External	INTEGER
END	

Figura 41 - Representação interna da *template* MANAGED OBJECT CLASS

Esta forma de representação foi utilizada para possibilitar ao implementador da tradução interativa, através da linguagem *script* definida no capítulo 6, o acesso a todas as informações das *templates* analisadas pelo tradutor. Com esta flexibilidade, o implementador poderá definir que tipo de informação adicional podem ser exportada para a implementação da tradução interativa. As informações, conforme definido em [OPS95] e [SOU95c], necessárias para o trabalho da tradução interativa podem ser exportadas de acordo com a especificação do implementador, por exemplo um arquivo contendo as referências aos OID dos objetos instanciados.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
MOName	<i>String</i>	Nome da classe, <class-label>
ClassName	<i>String</i>	Nome transformado para IDL, normalmente igual ao MOName.
IDLFileName	<i>String</i>	Nome do módulo no qual está declarado
ParentList	Lista de objetos	Lista da hierarquia de classes (DERIVED FROM)
AttributeList	Lista de objetos	Lista de atributos
AttributeGroupList	Lista de objetos	Lista de grupo de atributos
NameBindingsList	Lista de objetos	Lista da hierarquia de nomes
ActionList	Lista de objetos	Lista das <i>actions</i> declaradas
NotificationList	Lista de objetos	Lista das notificações declaradas
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
External	Inteiro	Define se a classe está presente no módulo atual ou se é uma classe existente em outro módulo. (1) Externa, (0) presente no módulo.

Tabela 26 - Definição da estrutura interna da *template* MANAGED OBJECT CLASS.

Um exemplo da realização da tradução da especificação da *template* MANAGED OBJECT CLASS em GDMO para IDL/CORBA é apresentado na Figura 42. Para esta *template* é somente necessário transformar o nome da classe GDMO em uma interface IDL/CORBA e relacionar a sua herança.

**equipment MANAGED OBJECT CLASS
DERIVED FROM top;**

REGISTERED AS { m3100ObjectClass 2 };

TRADUÇÃO

```
interface equipment : X0720::top { };
```

Figura 42 - Exemplo de *template* MANAGED OBJECT CLASS.

O mecanismo de tradução, apresentado na Figura 43, é mostrado de maneira simples, pois todo o trabalho de consistência do módulo, como a restrição de nomes, já foi realizado no momento da interpretação do módulo GDMO, restando apenas ao tradutor especificar como o código IDL será criado.

```
PRINT( "interface ", MOName )
parentCount = 0
FOREACH parent IN ParentList {
    IF( parentCount == 0 )          PRINT( " : " )
    IF( parent.IDLFileName != "" ) PRINT( "::" )
    PRINT( parent.ClassName )
    parentCount = parentCount + 1
    IF( parentCount != LISTCOUNT(ParentList) ) PRINT( ", " )
}
PRINT( " { };\n" )
```

Figura 43 - Implementação da tradução da *template* MANAGED OBJECT CLASS.

5.3.4 Mapeamento de PACKAGE para IDL/CORBA

A *template* PACKAGE, definida como mostra a Figura 44, é mapeada para IDL/CORBA conforme a Tabela 27. Neste processo de mapeamento, as informações referentes aos atributos, as ações e as notificações são mapeadas como operações nas interfaces IDL/CORBA correspondentes.

```
<package-label> PACKAGE
[BEHAVIOUR <behaviour-definition-label>[,<behaviour-definition-label>]*;]
[ATTRIBUTES <attribute-label> propertylist [<parameter-label>]*
    [,<attribute-label> propertylist [<parameter-label>]*;]
[ATTRIBUTES GROUPS <group-label> [<attribute-label>]*
    [,<group-label> [<attribute-label>]*]*;]
[ACTIONS <action-label> [<parameter-label>]*
    [,<action-label> [<parameter-label>]*]*;]
[NOTIFICATIONS <notification-label> [<parameter-label>]*
    [,<notification-label> [<parameter-label>]*]*;]
[REGISTERED AS object-identifier];

propertylist -> [REPLACE-WITH-DEFAULT]
                [DEFAULT VALUE      value-specifier]
                [INITIAL VALUE      value-specifier]
                [PERMITTED VALUES  type-reference]
                [REQUIRED VALUES  type-reference]
                [get-replace]
                [add-remove]
```

value-specifier ->	value-reference DERIVATION RULE <behaviour-definition-label>
get-replace ->	GET REPLACE GET-REPLACE
add-remove ->	ADD REMOVE ADD-REMOVE

Figura 44 - Estrutura da *template* PACKAGE

Algumas informações contidas na *template* PACKAGE, como as definições de DEFAULT VALUE, INITIAL VALUE, PERMITTED VALUES e REQUIRED VALUE, são utilizadas no processo de implementação da tradução interativa. Estas informações são utilizadas pelo implementador do mecanismo de tradução interativa, através da linguagem *script* definida no capítulo 6, exportando as informações que julgar necessárias no processo, pois o acesso as estas informações estão declarados na classe dos atributos relacionados com esta *template*.

Elemento	Informação Complementar	Comentário
BEHAVIOUR		Traduzido como comentário
ATTRIBUTES	GET	<tipo> <nome> _get() raises (CMIP_ATTRIBUTE_ERRORS);
	REPLACE	void <nome> _set(in <tipo> valor) raises (CMIP_ATTRIBUTE_ERRORS);
	GET-REPLACE	<tipo> <nome> _get() raises (CMIP_ATTRIBUTE_ERRORS); void <nome> _set(in <tipo> valor) raises (CMIP_ATTRIBUTE_ERRORS);
	ADD	void <nome> _add(in <tipo> elemento) raises (CMIP_ATTRIBUTE_ERRORS);
	REMOVE	void <nome> _remove(in <tipo> elemento) raises (CMIP_ATTRIBUTE_ERRORS);
	ADD-REMOVE	void <nome> _add(in <tipo> elemento) raises (CMIP_ATTRIBUTE_ERRORS); void <nome> _remove(in <tipo> elemento) raises (CMIP_ATTRIBUTE_ERRORS);
	REPLACE-WITH-DEFAULT	void <nome> _replaceWithDefault(in <tipo> elemento) raises (CMIP_ATTRIBUTE_ERRORS);
	DEFAULT VALUE	Valor utilizado nas operações REPLACE-WITH-DEFAULT
	INITIAL VALUE	Valor inicial utilizado na instanciação do atributo da PACKAGE
	PERMITTED VALUE	Utilizado como um definição de valores permitidos, referenciados em módulos ASN.1
	REQUIRED VALUE	Utilizado como um definição de valores necessários, referenciados em módulos ASN.1
ACTIONS	Nome da ação	Relação das diversas operações definidas em uma interface adicional.
NOTIFICATIONS	Nome da notificação	Relação das diversas operações definidas em uma interface adicional.
PARAMETERS	Nomes dos parâmetros	Traduzido em definição de tipos utilizados pela interface.
REGISTERED AS	Identificador do objeto	Utilizado para a tradução interativa.

Tabela 27 - Tradução da *template* PACKAGE.

Internamente ao tradutor a estrutura *template* PACKAGE é representada pela definição da classe relacionada na Figura 45.

CLASS	GDMOPackage	
	this	OBJECT
	PackageName	STRING
	AttributeList	LIST OF OBJECT
	ActionList	LIST OF OBJECT
	NotificationList	LIST OF OBJECT
	AttributeGroupList	LIST OF OBJECT
	Conditional	INTEGER
	Oid	STRING
	BehaviourReferenceList	LIST OF OBJECT
END		

Figura 45 - Representação interna da *template* PACKAGE.

A Tabela 28 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* PACKAGE.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
PackageName	<i>String</i>	Nome da <i>package</i> , <package-label>
AttributeList	Lista de objetos	Lista de atributos contidos na <i>template</i>
ActionList	Lista de objetos	Lista de <i>actions</i>
NotificationList	Lista de objetos	Lista das notificações
AttributeGroupList	Lista de objetos	Lista de grupo de atributos
Conditional	Inteiro	Representa uma PACKAGE condicional (1) ou não (0)
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento da PACKAGE

Tabela 28 - Definição da estrutura interna da *template* PACKAGE.

Um exemplo da realização da tradução da especificação da *template* PACKAGE para IDL/CORBA é apresentado na Figura 46. Esta *template* atua como um agrupamento de atributos, ações e notificações, esta transformação trabalha apenas com as propriedades dos atributos definidos, relacionandos como operações na interface IDL/CORBA correspondente a sua *template* MANAGED OBJECT CLASS.

equipment MANAGED OBJECT CLASS DERIVED FROM top; CHARACTERIZED BY equipmentPackage PACKAGE ATTRIBUTES equipmentId GET, replaceable GET; :: REGISTERED AS { m3100ObjectClass 2 }; TRADUÇÃO interface equipment : X0721::top { M3100ASN::NameTypeType equipmentId_get() raises (CMIP-ATTRIBUTE-ERRORS); }

```

M3100ASN::ReplaceableType replaceable_get()
    raises ( CMIP-ATTRIBUTE-ERRORS );

};

```

Figura 46 - Exemplo de *template* PACKAGE.

O mecanismo de tradução, apresentado na Figura 47, é mostrado de maneira simples, pois o trabalho de consistência do módulo, como a transformação dos nomes, já foi realizado no momento da interpretação da *template* PACKAGE pelo tradutor, restando apenas ao tradutor especificar como o código IDL será criado.

[illegible]

Figura 47 - Implementação da tradução da *template* PACKAGE.

5.3.5 Mapeamento de ATTRIBUTE para IDL/CORBA

A *template* ATTRIBUTE, definida como mostra a Figura 48, é mapeada para IDL/CORBA conforme a Tabela 29. Neste processo, a sintaxe de tipo do atributo é mapeado como uma definição de tipo nas interfaces IDL/CORBA correspondentes, conforme já definido no item 5.2.1.

```
<attribute-label> ATTRIBUTE
    derived-or-with-syntax-choice;
    [MATCHES FOR qualifier [, qualifier]*;
    ]
    [BEHAVIOUR <behaviour-definition-label>
```



```

                                [, <behaviour-definition-label>]*;
    ]
    [PARAMETERS <parameter-label>
                                [, <parameter-label>]*;
    ]
    [REGISTERED AS object-identifier];

    qualifier -> EQUALITY | ORDERING | SUBSTRINGS |
                SET-COMPARISON | SET-INTERSECTION

    derived-or-with-syntax-choice -> DERIVED FROM <attribute-label> |
                                    WITH ATTRIBUTE SYNTAX type-reference

```

Figura 48 - Estrutura da *template* ATTRIBUTE

As informações referentes as cláusulas MATCHES FOR são tratadas de forma específica para a tradução interativa, pois são referentes a mecanismos de filtragem e pesquisa do CMISE, sendo possível a sua definição na interface IDL/CORBA ManagedObject.[OPS95]

Elemento	Comentário
DERIVED FROM	Define a sintaxe do atributo, definido em módulos ASN.1, ou em uma definição de atributo em GDMO.
WITH ATTRIBUTE SYNTAX	
MATCHES FOR	Definição de filtros de pesquisa CMISE, a ser implementado pela tradução interativa.
BEHAVIOUR	Transformado em comentários na interface IDL/CORBA.
PARAMETERS	São transformados em declaração de tipos.
REGISTERED AS	Utilizado para a tradução interativa.

Tabela 29 - Tradução da *template* ATTRIBUTE

A estrutura interna referente a *template* ATTRIBUTE é apresenta na Figura 49.

CLASS GDMOAttribute	
this	OBJECT
AttributeName	STRING
Syntax	STRING
Derived	STRING
MatchesFor	INTEGER
ReplaceWithDefault	INTEGER
DefaultValue	STRING
InitialValue	STRING
PermittedValue	STRING
RequiredValue	STRING
GetReplace	INTEGER
AddRemove	INTEGER
Oid	STRING
ClassName	STRING
IDLFileName	STRING
BehaviourReferenceList	LIST OF OBJECT
END	

Figura 49 - Representação interna da *template* ATTRIBUTE.

A Tabela 30 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* ATTRIBUTE.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
AttributeName	<i>String</i>	Nome do atributo, <attribute-label>
Syntax	<i>String</i>	Referência a valor definido em ASN.1

Derived	<i>String</i>	Referência a valor definido em ASN.1
MatchesFor	Inteiro	(1) Equality (2) Ordering (4) SubStrings (8) Set-Comparison (16) Set-Intersection
ReplaceWithDefault	Inteiro	(1) Sim (0) Não
DefaultValue	<i>String</i>	Referência a valor definido em ASN.1
InitialValue	<i>String</i>	Referência a valor definido em ASN.1
PermittedValue	<i>String</i>	Referência a tipo definido em ASN.1
RequiredValue	<i>String</i>	Referência a tipo definido em ASN.1
GetReplace	Inteiro	(1) Get (2) Replace (3) Get-Replace
AddRemove	Inteiro	(1) Add (2) Remove (3) Add-Remove
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
ClassName	<i>String</i>	Nome transformado para IDL, normalmente igual ao AttributeName.
IDLFileName	<i>String</i>	Nome do módulo no qual está declarado
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento do ATTRIBUTE

Tabela 30 - Definição da estrutura interna da *template* ATTRIBUTE.

O mecanismo de transformação das informações dos atributos já foi descrito em detalhes junto a *template* PACKAGE, as informações contidas nesta representação são específicas para o implementador do mecanismo de tradução interativa, pois terá acesso as informações adicionais do atributo, como as específicas para manipulação dos filtros CMIS, representadas pelos valores constantes no elemento `MachesFor`, entre outras.

5.3.6 Mapeamento de ATTRIBUTE GROUP para IDL/CORBA

A *template* ATTRIBUTE GROUP, definida como mostra a Figura 50, seu mapeamento deve ser especificado pelo implementador, pois somente a definição de como ter acesso aos grupos de atributos é específico do mecanismo implementado.

```

<group-label> ATTRIBUTE GROUP
    [GROUP ELEMENTS <attribute-label>
        [, <attribute-label>]*;]
    [FIXED ;]
    [DESCRIPTION delimited-string;]
    REGISTERED AS object-identifier;

```

Figura 50 - Estrutura da *template* ATTRIBUTE GROUP

A estrutura interna referente a *template* ATTRIBUTE GROUP é apresenta na Figura 51.

```
CLASS GDMOAttributeGroup
```

this	OBJECT
AttributeGroupName	STRING
Elements	LIST OF STRING
Extensions	LIST OF STRING
Fixed	INTEGER
Oid	STRING
END	

Figura 51 - Representação interna da *template* ATTRIBUTE GROUP.

A Tabela 31 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* ATTRIBUTE GROUP.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
AttributeGroupName	<i>String</i>	Nome do grupo de atributos, <group-label>
Elements	Lista de <i>string</i>	Relação de atributos definidos por este agrupamento
Extensions	Lista de <i>string</i>	Definição constante na cláusula DESCRIPTION
Fixed	Inteiro	(1) Sim (0) Não
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS

Tabela 31 - Definição da estrutura interna da *template* ATTRIBUTE GROUP.

Como descrito para a *template* ATTRIBUTE, as informações contidas nesta classe são específicas para o implementador do mecanismo de tradução interativa, pois terá acesso a todas as informações do agrupamento de atributos. Este acesso permitirá definir que tipo de operações serão definidas na tradução interativa. Em [OPS95] o acesso aos grupos de atributos são definidos através da classe *ManagedObject*, que poderá conter as seguintes operações:

```
sequence <any> GetAttributeGroup();
void SetAttributeGroupToDefault();
```

5.3.7 Mapeamento de PARAMETER para IDL/CORBA

A *template* PARAMETER, definida como mostra a Figura 52, é mapeada para IDL/CORBA conforme mostra a Tabela 32. Neste processo de mapeamento os parâmetros são transformados em declarações de tipos nas interfaces IDL/CORBA correspondentes.

<pre> <parameter-label> PARAMETER CONTEXT context-type; syntax-or-attribute-choice; [BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*;] [REGISTERED AS object-identifier]; context-type -> context-keyword ACTION-INFO ACTION-REPLY EVENT-INFO EVENT-REPLY SPECIFIC-ERROR context-keyword -> type-reference.<identifier> syntax-or-attribute-choice -> WITH SYNTAX type-reference ATTRIBUTE <attribute-label> </pre>	
--	--

Figura 52 - Estrutura da *template* PARAMETER

A Tabela 32 apresenta as informações a serem tratadas da *template* PARAMETER.

Esta *template* é transformada como uma definição de tipo na interface IDL/CORBA na qual ela está relacionada.

Elemento	Comentário
PARAMETER	Transformados no nome do tipo definido pela declaração WITH ... SYNTAX
WITH ATTRIBUTE SYNTAX	Define a sintaxe do atributo, definido em módulos ASN.1.
CONTEXT	São traduzidos como comentários junto a declaração de tipo correspondente.
BEHAVIOUR	Transformado em comentários na interface IDL/CORBA.
REGISTERED AS	Utilizado para a tradução interativa.

Tabela 32 - Tradução da *template* PARAMETER

A *template* PARAMETER é representada internamente pelo tradutor através da estrutura definida na Figura 53. Os elementos ParameterName e Syntax determinam como esta *template* será transformada, sendo que ParameterName define o nome do tipo criado a partir de Syntax.

CLASS GDMOParame	
this	OBJECT
ParameterName	STRING
Context	INTEGER
Syntax	STRING
Oid	STRING
BehaviourReferenceList	LIST OF OBJECT
END	

Figura 53 - Representação interna da *template* PARAMETER.

A Tabela 33 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* PARAMETER.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
ParameterName	<i>String</i>	Nome do parâmetro, <parameter-label>
Context	Inteiro	(0) ACTION-INFO (1) ACTION-REPLY (2) EVENT-INFO (4) EVENT-REPLY (8) SPECIFIC-ERROR
Syntax	<i>String</i>	Referência a tipo definido em ASN.1
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento do PARAMETER

Tabela 33 - Definição da estrutura interna da *template* PARAMETER.

5.3.8 Mapeamento de ACTION para IDL/CORBA

A *template* ACTION, definida como mostra a Figura 54, é mapeada para IDL/CORBA conforme mostra a Tabela 34. Neste processo de mapeamento as informações referentes as ações são transformadas em operações nas interfaces IDL/CORBA correspondentes.

```

<action-label> ACTION
    [BEHAVIOUR <behaviour-definition-label>
      [, <behaviour-definition-label>]*;]
    [MODE CONFIRMED ;]
    [PARAMETERS <parameter-label>
      [, <parameter-label>]*;]
    [WITH INFORMATION SYNTAX type-reference ;]
    [WITH REPLY SYNTAX type-reference ;]
    REGISTERED AS object-identifier;

```

Figura 54 - Estrutura da *template* ACTION

A Tabela 34 apresenta as informações a serem tratadas da *template* ACTION.

Elemento	Comentário
ACTION	Transformado no nome da operação na interface IDL correspondente.
BEHAVIOUR	Transformado em comentários na interface IDL/CORBA.
MODE	Determina se a operação pode ser transformada como oneway em IDL/CORBA
WITH ATTRIBUTE SYNTAX	Define a sintaxe do argumento da operação, conforme seu tipo definido em módulo ASN.1
WITH REPLY SYNTAX	Define a sintaxe do valor de retorno da operação correspondente a esta ação, seu tipo é definido em módulo ASN.1
REGISTERED AS	Utilizado para a tradução interativa.

Tabela 34 - Tradução da *template* ACTION

A *template* ACTION é representada internamente pelo tradutor através da estrutura definida na Figura 55.

```

CLASS GDMOAction
    this                OBJECT
    ActionName          STRING
    Mode                INTEGER
    InfoSyntax           STRING
    ReplySyntax         STRING
    Oid                 STRING
    BehaviourReferenceList  LIST OF OBJECT
END

```

Figura 55 - Representação interna da *template* ACTION.

A Tabela 35 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* ACTION.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
ActionName	<i>String</i>	Nome da ação, <action-label>
Mode	Inteiro	(1) CONFIRMED (0) Não confirmada
InfoSyntax	<i>String</i>	Referência a tipo definido em ASN.1

ReplySyntax	<i>String</i>	Referência a tipo definido em ASN.1
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento da ACTION

Tabela 35 - Definição da estrutura interna da *template* ACTION.

5.3.9 Mapeamento de NOTIFICATION para IDL/CORBA

A *template* NOTIFICATION, definida como mostra a Figura 56, é mapeada para IDL/CORBA conforme mostra a Tabela 36. Neste processo de mapeamento as informações referentes aos atributos, as ações e as notificações são mapeadas como operações nas interfaces IDL/CORBA correspondentes.

<pre> <notification-label> NOTIFICATION [BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*;] [PARAMETERS <parameter-label> [, <parameter-label>]*;] [WITH INFORMATION SYNTAX type-reference [AND ATTRIBUTE IDS <field-name> <attribute-label> [, <field-name> <attribute-label>]*] ;] [WITH REPLY SYNTAX type-reference ;] REGISTERED AS object-identifier; </pre>

Figura 56 - Estrutura da *template* NOTIFICATION

A Tabela 36 apresenta as informações a serem tratadas da *template* NOTIFICATION.

Elemento	Comentário
NOTIFICATION	Transformado em nome de operação na interface IDL correspondente.
BEHAVIOUR	Transformado em comentários na interface IDL/CORBA.
PARAMETERS	São transformados em comentários.
WITH ATTRIBUTE SYNTAX	Define a sintaxe do argumento da operação, conforme seu tipo definido em módulo ASN.1
WITH REPLY SYNTAX	Define a sintaxe do retorno da operação correspondente a esta notificação, seu tipo é definido em módulo ASN.1
AND ATTRIBUTES IDS	Não traduzido.
REGISTERED AS	Utilizado para a tradução interativa.

Tabela 36 - Tradução da *template* NOTIFICATION

A *template* NOTIFICATION é representada internamente pelo tradutor através da estrutura definida na Figura 57.

CLASS GDMONotification	
this	OBJECT
NotificationName	STRING
InfoSyntax	STRING
ReplySyntax	STRING
Oid	STRING
BehaviourReferenceList	LIST OF OBJECT
END	

Figura 57 - Representação interna da *template* NOTIFICATION.

A Tabela 37 apresenta uma descrição dos elementos contidos na estrutura interna utilizada para representar as informações obtidas da *template* NOTIFICATION.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
NotificationName	<i>String</i>	Nome da notificação, <notification-label>
InfoSyntax	<i>String</i>	Referência a tipo definido em ASN.1
ReplySyntax	<i>String</i>	Referência a tipo definido em ASN.1
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento da NOTIFICATION.

Tabela 37 - Definição da estrutura interna da *template* NOTIFICATION.

5.3.10 Mapeamento de NAME BINDING para IDL/CORBA

A *template* NAME BINDING, definida como mostra a Figura 58, é apenas analisada e preenche a estrutura de informações específica do tradutor, deixando seu uso a critério do implementador da tradução interativa.

<pre> <name-binding-label> NAME BINDING SUBORDINATE OBJECT CLASS <class-label> [AND SUBCLASSES]; NAMED BY SUPERIOR OBJECT CLASS <class-label> [AND SUBCLASSES]; WITH ATTRIBUTE <attribute-label>; [BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*;] [CREATE [create-modifier [, create-modifier]] [<parameter-label>]*;] [DELETE [delete-modifier] [<parameter-label>]*;] REGISTERED AS object-identifier; create-modifier -> WITH-REFERENCE-OBJECT WITH-AUTOMATIC-INSTANCE-NAMING delete-modifier -> ONLY-IF-NO-CONTAINED-OBJECTS DELETES-CONTAINED-OBJECT </pre>
--

Figura 58 - Estrutura da *template* NAME BINDING

A estrutura interna referente a *template* NAME BINDING é apresentada na Figura 59.

CLASS	GDMONameBinding	
	this	OBJECT
	BindingName	STRING
	Superior	STRING
	Subordinate	STRING
	NamingAttribute	STRING
	Create	INTEGER
	Delete	INTEGER
	Oid	STRING
	BehaviourReferenceList	LIST OF OBJECT
	DeleteAttribute	STRING
	DeleteValue	STRING
END		

Figura 59 - Representação interna da *template* NAME BINDING.

A Tabela 38 apresenta as informações a serem tratadas da *template* NAME BINDING.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
BindingName	<i>String</i>	Nome da <i>name binding</i> , <name-binding-label>
Superior	<i>String</i>	Nome da superclasse
Subordinate	<i>String</i>	Nome da subclasse
NamingAttribute	<i>String</i>	Nome do atributo utilizado pelo <i>name binding</i>
Create	Inteiro	(1) WITH-AUTOMATIC-INSTANCE-NAMING (2) WITH-OBJECT-REFERENCE
Delete	Inteiro	(1) ONLY-IF-NO-CONTAINED-OBJECTS (2) DELETES-CONTAINED-OBJECTS
Oid	<i>String</i>	<i>Object-Identifier</i> obtido de REGISTERED AS
BehaviourReferenceList	Lista de objetos	Lista da declaração de comportamento da NAME BINDING

Tabela 38 - Definição da estrutura interna da *template* NAME BINDING.

5.3.11 Mapeamento de BEHAVIOUR para IDL/CORBA

A *template* BEHAVIOUR, definida como mostra a Figura 60, é mapeada para IDL/CORBA como comentário. As informações definidas em BEHAVIOUR são necessárias apenas para uma descrição da classe em qual ela está definida, existe uma opção de que futuramente esta *template* possa ser realmente utilizada por algum tradutor, mas para isto será necessário que seu conteúdo seja definido formalmente.

```
<behaviour-definition-label> BEHAVIOUR
    DEFINED AS delimited-string;
```

Figura 60 - Estrutura da *template* BEHAVIOUR

A estrutura interna referente a *template* BEHAVIOUR é apresentada na Figura 61.

```
CLASS GDMOBehaviour
    this                OBJECT
    BehaviourName       STRING
    Definition           STRING
END
```

Figura 61 - Representação interna da *template* BEHAVIOUR.

A Tabela 39 apresenta as informações a serem tratadas da *template* NAME BINDING.

Elemento	Tipo	Propósito
this	Objeto	Endereço da classe (como em C++)
BehaviourName	<i>String</i>	Nome do <i>behaviour</i> , <behaviour-label>
Definition	<i>String</i>	Conteúdo definido na cláusula DEFINED AS.

Tabela 39 - Definição da estrutura interna da *template* BEHAVIOUR.

Seu mapeamento consiste apenas na transformação de seu conteúdo, definido no elemento *Definition* da estrutura da Tabela 39, em comentário junto a tradução da *template* para a qual ela foi definida.

5.4 Considerações Finais

Neste capítulo foram apresentados as regras utilizadas para a tradução das especificações ASN.1 e GDMO para IDL/CORBA. Um enfoque especial foi dado ao processo de tradução das especificações GDMO para IDL/CORBA, mostrando em detalhes como a informação obtida do módulo em GDMO é representada no tradutor, e como uma linguagem *script* pode trabalhar sobre estas informações para a criação dos arquivos em IDL.

CAPÍTULO 6

DESENVOLVIMENTO DO TRADUTOR DE GDMO/OSI PARA IDL/CORBA

6.1 Introdução

Atualmente existem vários estudos sobre como realizar, a transformação de GDMO em IDL/CORBA, mas todos realizam apenas a tradução das especificações. Uma das características deste método de tradução é a possibilidade de disponibilizar informações para o processo da tradução interativa, pois este é dependente da implementação. Isto mostrou a necessidade de um mecanismo específico para a exportação das informações obtidas pelo processo de tradução. A ferramenta possui internamente uma linguagem *script* que possibilita ao implementador acesso as estruturas internas do tradutor, definidas como classes. Estas classes internas representam as informações obtidas das *templates* em GDMO, conforme representado na Figura 62.

A implementação do tradutor de GDMO para IDL/CORBA foi realizada em duas etapas distintas, uma para as declarações de tipos e valores em ASN.1, através de um tradutor chamado de *asn1-idl*, e outra para os módulos definidos em GDMO, através de um tradutor chamado de *gdm2idl*. Esta divisão foi utilizada pois as linguagens possuem características próprias, que devem ser interpretadas por compiladores específicos.

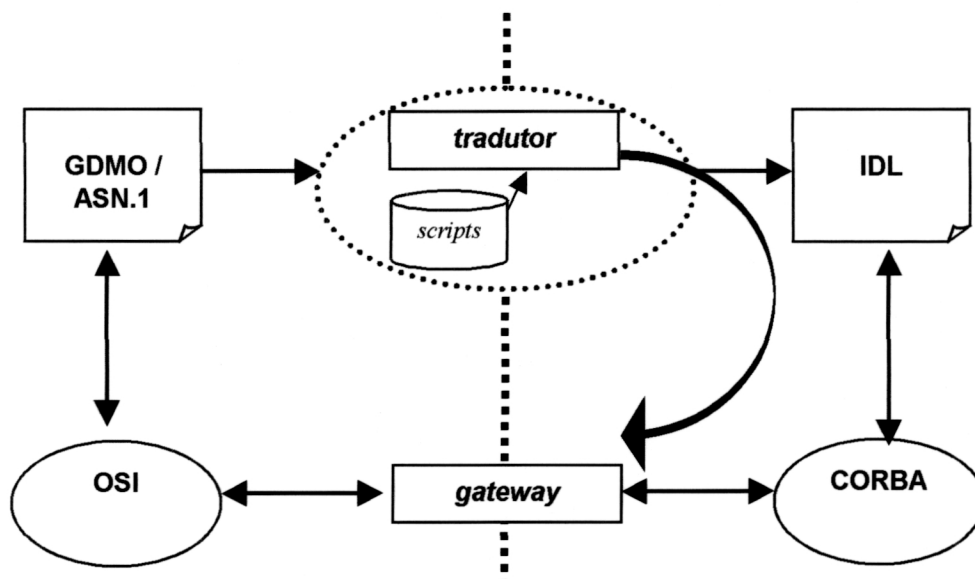


Figura 62 - Mecanismo de tradução

6.2 Tradutor de ASN.1

O principal problema da transformação da linguagem ASN.1 em IDL/CORBA é a complexidade da implementação de ASN.1. Para resolver este problema o tradutor é implementado em diversas fases, fazendo diversas análises sobre o código para conseguir montar a estrutura da árvore de *parsing*. As fases executadas pelo tradutor são:

1. Analisar os módulos ASN.1;
2. Resolver os tipos declarados;
3. Analisar as declarações dos valores nos módulos ASN.1;
4. Resolver as referências aos valores importados;
5. Normalizar os tipos;
6. Checagem dos erros semânticos que possam ocorrer;
7. Gerar o código IDL/CORBA para as definições de tipos ASN.1;
8. Gera o arquivo de atualização para o tradutor GDMO;

A representação da implementação do tradutor é apresentada na Figura 63.

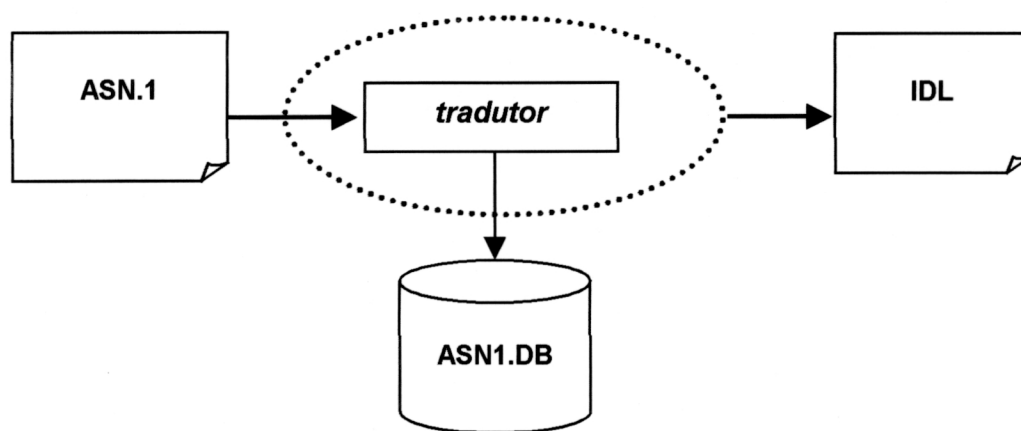


Figura 63 - Tradutor de ASN.1 para IDL/CORBA.

Analise dos módulos ASN.1

Nesta fase um módulo específico é interpretado para uma estrutura interna do tradutor, representada através de uma árvore de *parsing*. Após esta consulta ao módulo ASN.1 ele não é novamente processado. O núcleo principal do tradutor de ASN.1 foi obtido de um compilador de ASN.1 para C/C++, o SNACC [SAM93]. Todas as modificações realizadas no tradutor, foram para compatibilizá-lo com as especificações desenvolvida no capítulo 5. A parte principal da análise é feita através da especificação da linguagem ASN.1 nas ferramentas **Yacc** e **Lex** [AHO85][LEV92][LS85][JOHN85].

Como não há necessidade de processamento das definições de MACRO, elas são ignoradas, pois não existe a necessidade da criação de macros para os módulos em GDMO, por isso, todas as declarações entre BEGIN e END são apenas processadas pelo analisador **Lex** [LEV92][LS85].

Os erros de sintaxe encontrados são mostrados nesta etapa do processo de análise do módulo. A recuperação de um erro ocorre após a ocorrência de uma “;”, possibilitando que apenas a declaração em questão seja relacionada como incorreta.

Resolução dos tipos declarados

Nesta etapa da compilação são resolvidos os tipos declarados no módulo em ASN.1. Isto é feito para cada identificador de módulo, ele é verificado para evitar conflitos com outros módulos. Dois módulos com mesmo nome, mas com OBJECT IDENTIFIER diferentes não são tratados como erro, pode ocorrer apenas uma sobreposição dos arquivos resultantes da tradução.

Algumas verificações quanto a determinadas declarações são realizadas aqui, como por exemplo:

- um tipo SELECTION deve referenciar um campo de um tipo CHOICE;

- um tipo COMPONENTS OF em uma declaração SET deve referenciar um SET;
- um tipo COMPONENTS OF em uma declaração SEQUENCE deve referenciar um SEQUENCE.

Analisar as declarações dos valores nos módulos ASN.1

Neste passo é feita a análise da declaração de qualquer valor declarado dentro de { }. São tratados todas as declarações de INTEGER, REAL e BOOLEAN, bem como as declarações dos OBJECT IDENTIFIERS.

Resolver as referências aos valores importados

Neste passo do tradutor são tratadas todas as referências dos OBJECT IDENTIFIER, isto é realizado para fazer a checagem dos valores declarados dentro do OBJECT IDENTIFIER com os valores permitidos na árvore de registro.

Normalizar os tipos

Nesta passagem são realizadas as normalizações de tipos declarados, ou seja, o alinhamento dos tipos em ASN.1 com os disponível em IDL/CORBA, isto é necessário para tornar a geração do código em IDL/CORBA mais simples. Pois haverá uma perda na tradução, pois a linguagem ASN.1 é uma linguagem criada especificamente para a criação de estruturas de dados, o que não ocorre em IDL/CORBA, que utiliza uma declaração simples de seus atributos.

Checagem dos erros semânticos que possam ocorrer

As principais checagens semânticas realizadas referem-se aos elementos definidos dentro de declarações CHOICE, pois seus elementos devem possuir tipos com *tags* distintas.

Gerar o código IDL/CORBA para as definições de tipos ASN.1

Neste ponto o tradutor ASN.1 já possui toda informação necessária para a transformação das especificações em ASN.1 em declaração de tipos IDL/CORBA.

Gerar o arquivo de atualização para o tradutor de GDMO

Uma das principais tarefas do tradutor de ASN.1 é atualizar todas as informações, já transformadas em IDL/CORBA, para o tradutor de GDMO, isto é necessário pois o tradutor *gdm2idl* precisa saber qual o nome do módulo em qual uma determinada declaração de sintaxe foi definida. Este arquivo, como representado na Figura 64, atuam como um banco de dados de apelidos (*nickname*), possibilitando o relacionameto das informações transformadas com os nomes criados e os módulos em quais se encontram. O arquivo a ser atualizado para o tradutor de GDMO é:

- **ASN1.DB:** utilizado pelo tradutor GDMO para resolver as referências criadas pelas definições de tipos e valores em ASN.1. Este arquivo contém todos os nomes dos atributos e valores já transformados, e os nomes dos arquivos IDL/CORBA no qual eles se encontram.

Nome ASN.1	Nome IDL	Nome do Módulo IDL
ProblemCause	ProblemCauseType	X0721Att
LoggingTime	LoggingTimeType	X0721Att
smi2Parameter	smi2Parameter	X0721Par

Figura 64 - Estrutura do arquivo ASN1.DB

6.3 Tradutor de GDMO

Uma das principais características do tradutor *gdm2idl* é o acesso à suas estruturas de dados internas. A estrutura do tradutor de GDMO foi obtida a partir do compilador do OSIMIS [OSIMIS93], pois sua abordagem atendia as principais necessidades para acesso as informações definidas nas *templates* GDMO. Esta solução permitiu que os algoritmos de transformação das especificações em GDMO para IDL fossem construídos através da sua linguagem *script*, e ainda assim dar ao implementador acesso às mesmas informações, ver Figura 66. O acesso as informações das *templates* em GDMO possibilitam que o implementador possa alterar os *scripts* de transformação, criando os seus próprios *scripts* para transformar os arquivos em IDL, adequando o resultado em IDL às suas necessidades particulares, como por exemplo determinado tipo de construção de interfaces para acesso através de um *gateway*. Em [FER98] vemos que é possível a construção de um *gateway* entre as aplicações OSI e CORBA, e a necessidade de um mecanismo personalizado para a construção das interfaces IDL pode ser útil.

No tradutor, a árvore de *parsing*, após estar completa, é transformada em uma relação de classes. Com estas classes o implementador possui acesso completo a todas as estruturas declaradas nos módulos GDMO. Esta abordagem foi utilizada por ser possível realizar, de maneira personalizada, a transferência de informações para a criação do mecanismo de tradução interativa.

A declaração principal das estruturas internas do tradutor corresponde a classe representada na Figura 65. Nela é feito todo o relacionamento existente entre as *templates* definidas nos módulos GDMO, como por exemplo as *templates* referentes aos atributos são relacionados através do elemento `AttributeList`.

CLASS GDMO	
this	OBJECT
ManagedObjectList	LIST OF OBJECT
PackageList	LIST OF OBJECT
ConditionalPackageList	LIST OF OBJECT

AttributeList	LIST OF OBJECT
ActionList	LIST OF OBJECT
NotificationList	LIST OF OBJECT
ParameterList	LIST OF OBJECT
NamebindingsList	LIST OF OBJECT
AttributeGroupList	LIST OF OBJECT
BehaviourReferenceList	LIST OF OBJECT
END	

Figura 65 - Estrutura de dados inicial utilizada pelo tradutor GDMO.

Na Figura 68 e na Figura 69 é apresentado o funcionamento do mecanismo para tradução das especificações GDMO/ASN.1 para IDL/CORBA. A principal característica do processo de tradução está nos *scripts* de tradução, que são arquivos contendo os procedimentos necessários para realizar o processo de criação dos arquivos em IDL/CORBA. Grande parte destes procedimentos já foram mostrados no capítulo 5, especialmente os procedimentos listados na Figura 43 e na Figura 47, reportando os procedimentos de tradução das *templates* MANAGED OBJECT CLASS e as operações definidas sobre os atributos declarados em uma *template* PACKAGE.

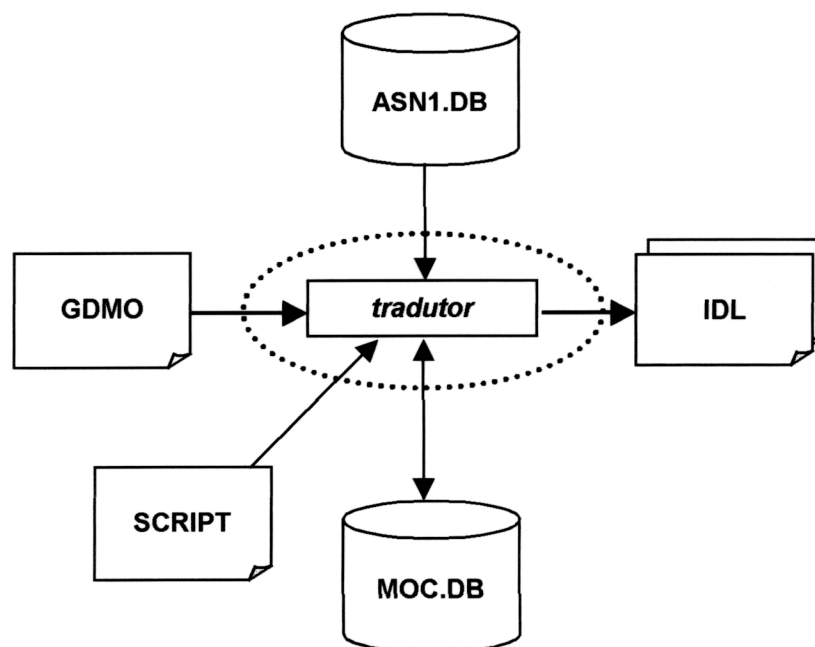


Figura 66 - Tradutor de GDMO para IDL/CORBA.

Além do processo de tradução, o tradutor é responsável por atualizar as informações no arquivo de apelidos (*nickname*). Neste arquivo estão relacionadas todos os módulos em GDMO já transformados para IDL, bem como os nomes das *templates* relacionadas.

- **MOC.DB:** utilizado pelo tradutor GDMO para resolver as referências às *templates*, este arquivo contém os nomes das interfaces e operações IDL já transformados e os nomes dos arquivos IDL/CORBA nos quais eles se encontram.

Nome GDMO	Nome IDL	Nome do Módulo IDL
top	top	X0721
terminationPoint	terminationPoint	M3100
equipment	equipment	M3100

Figura 67 - Estrutura do arquivo MOC.DB

Este arquivo serve como referência para todos os módulos que já foram utilizados pelo tradutor, ele apresenta os nomes das classes transformadas para IDL/CORBA, bem como o nome do arquivo IDL/CORBA que foi gerado pelo tradutor. O nome do arquivo IDL é utilizado para referência em outros módulos, através do uso da condição de `#include <nome_módulo.idl>` em um novo módulo traduzido, que faça referência ao módulo origem.

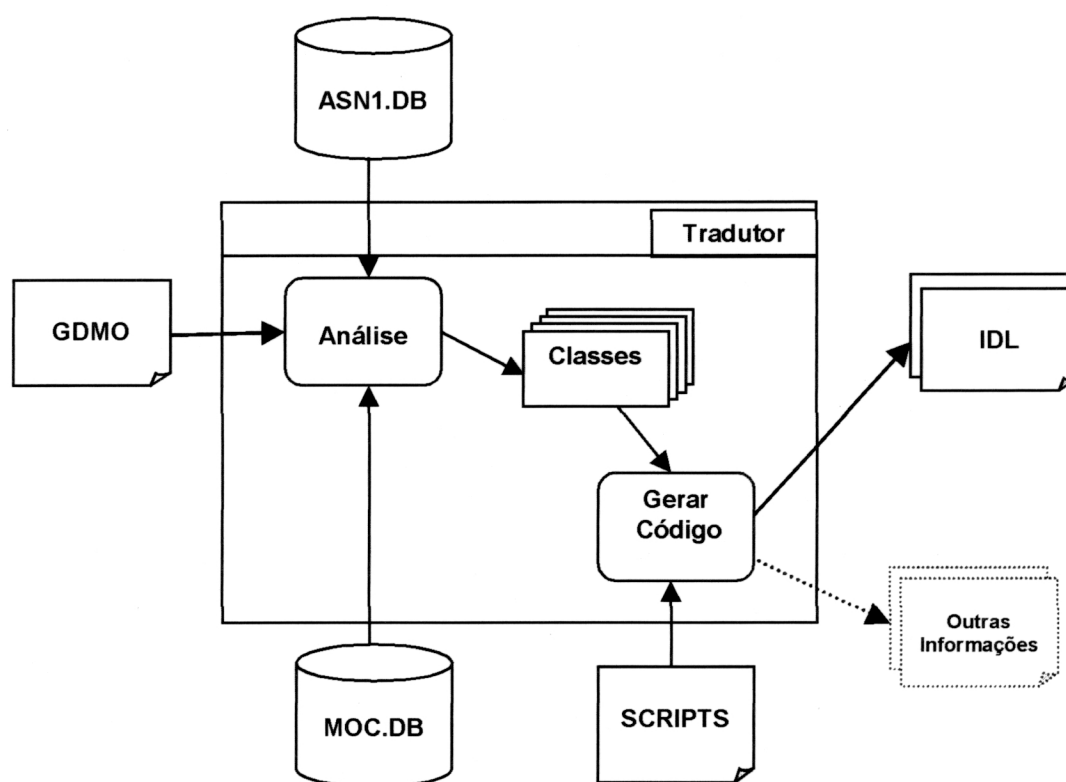


Figura 68 - Tradutor de GDMO para IDL/CORBA.

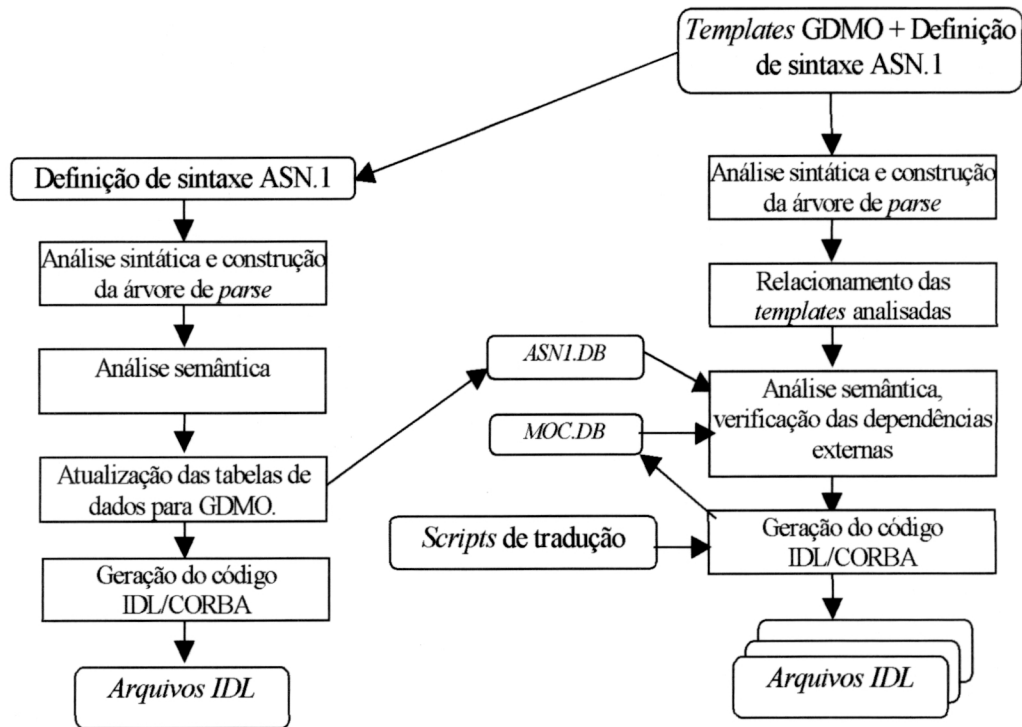


Figura 69 - Processo de tradução de especificações GDMO e ASN.1 para IDL/CORBA.

6.3.1 Linguagem *Script* do Tradutor

A linguagem *script* utilizada pelo tradutor GDMO é de fácil compreensão e implementação, sendo suas principais características:

- ◆ Permitir a declaração de variáveis;
- ◆ Possuir várias estruturas de controle (*if*, *while*, *foreach*);
- ◆ Manipular expressões aritméticas, lógicas e relacionais;
- ◆ Suportar modularidade de *scripts*, mas não modularidade de procedimentos e funções;
- ◆ Possuir diversas funções e procedimentos internos que ajudam na geração do código.

Esta linguagem foi definida para permitir o acesso às informações internas das classes do tradutor GDMO, ela foi implementada com poucas estruturas de controle, somente possui IF, WHILE e FOREACH, possui poucas definições de tipos de variáveis, possui somente OBJECT, STRING e INTEGER. Sua utilização é para percorrer os elementos constantes nas estruturas das *templates* analisadas pelo tradutor e possibilitar a sua transformação em código IDL/CORBA.

Além das características descritas acima, a linguagem possui diversas funções, criadas para dar ao implementador ferramentas auxiliares no processo de produção do código IDL ou informações específicas para o mecanismo de tradução interativa.

Statments

A linguagem *script* providencia diversas estruturas de controle e *statments*:

♦ <i>variável = expressão</i>	Atribuição de valores às variáveis
♦ <i>;</i>	Comando nulo
♦ <i>IF(expressão) comandos</i>	Comando condicional
♦ <i>IF(expressão) comandos ELSE comandos</i>	Comando condicional
♦ <i>WHILE(expressão) comandos</i>	Comando de iteração
♦ <i>FOREACH variável IN variável comandos</i>	Comando de iteração em listas
♦ <i>{ comandos }</i>	Bloco de comandos

Expressões

Os seguintes tipos de expressões são suportados:

♦ <i>variável</i>	Retorna o valor da variável
♦ <i>variávelL.variávelC</i>	Retorna o valor do campo
♦ <i>“texto”</i>	Cadeia de caracter
♦ <i>números inteiros</i>	
♦ <i>expressão aritmética (+, -, /, *)</i>	
♦ <i>expressão relacional (>, >=, <, <=, ==, !=)</i>	
♦ <i>expressão lógica (AND ou &&, OR ou)</i>	
♦ <i>NOT expressão</i>	
♦ <i>(expressão)</i>	

Funções Internas

- ♦ **EXISTFILE(nome_arquivo)** – Verifica a existência de um arquivo em disco.
- ♦ **LISTCOUNT(variável)** – Retorna a quantidade de elementos em uma lista de objetos.

- ◆ EXECUTE(*nome_script*, *nome_arquivo*, *nome_objeto*) – Executa um *script*, que está localizado em disco <*nome_script*>, criando um arquivo de saída <*nome_arquivo*>, e recebendo um objeto específico <*nome_objeto*> como parâmetro.
- ◆ DISPLAY(*expressão*) – Escreve na console o resultado da expressão.
- ◆ PRINT(*expressão*) – Escreve em um arquivo o resultado da expressão.
- ◆ INCLUDE(*nome_arquivo*) – Copia o conteúdo de <*nome_arquivo*> no arquivo de saída atualmente em uso.

Parâmetros do Tradutor

Na Figura 70 é novamente apresentado o processo de tradução dos arquivos GDMO e ASN.1 em arquivos IDL, em específico é mostrado que o tradutor *gdmo2idl* têm dois arquivos específicos de entrada, o INICIO.GEN e TRADUZ.SCT. Estes arquivos são respectivamente sobre a estrutura de dados do tradutor e o mecanismo de tradução das especificações GDMO para IDL.

A estrutura de dados do tradutor, como já mostrado anteriormente, é definida através de um conjunto de classes relacionadas entre si. Este relacionamento entre classes segue o relacionamento mostrado na Figura 24.

Para o implementador do mecanismo de tradução interativa é possível especificar o seu arquivo de *script* para a geração das suas informações. Para isso deve apenas passar para o tradutor, através da linha de comando, um parâmetro informando o nome do arquivo de *script* a ser utilizado no lugar de TRADUZ.SCT:

```
$ gdmo2idl -s script_definido_pelo_implementador arquivo.gdmo
```

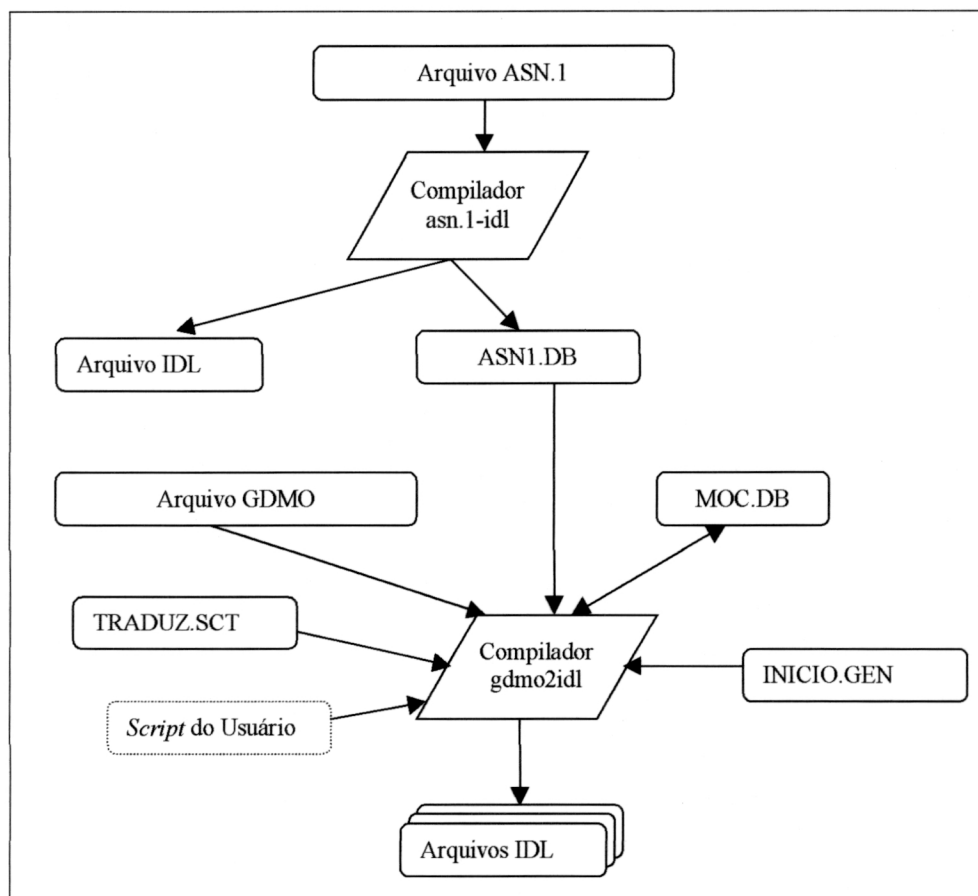


Figura 70 - Arquitetura da tradução GDMO/OSI para IDL/CORBA

6.4 Considerações Finais

Neste capítulo foi apresentada a forma como os tradutores foram implementados, sendo dado especial atenção ao tradutor de GDMO para IDL.

O tradutor de GDMO apresenta para seu funcionamento uma linguagem *script* para acesso às suas estruturas internas. Este acesso garante ao implementador uma ferramenta personalizável, visto que é possível realizar a conversão de GDMO não apenas para IDL, mas possibilita o acesso a diversos tipos de informações do módulo em GDMO.

CAPÍTULO 7

CONCLUSÃO E TRABALHOS FUTUROS

A atividade de gerenciamento de sistemas se apresenta como uma tarefa extremamente complicada, se avaliada a quantidade de especificações, ambientes, protocolos e informações necessárias para seu funcionamento. É desejável que todo o processo de gerenciamento seja desenvolvido em uma arquitetura em particular, aproveitando as informações disponíveis em outras arquiteturas. Este trabalho se concentra nas arquiteturas OSI e CORBA.

Assim vemos que já foi realizado um investimento considerável em ferramentas de notações, por exemplo GDMO, ASN.1 e IDL, o que produziu diversos verificadores de sintaxe, e geradores de estrutura de dados. Alinhar todas estas ferramentas é impraticável, mas acredita-se que seja possível produzir ferramentas automáticas ou semi-automáticas para a tradução.

O objetivo deste trabalho foi desenvolver uma ferramenta semi-automática para a tradução das especificações GDMO/ASN.1 para IDL, na busca de facilitar o emprego destas tecnologias na gerência de redes de computadores. Concentramos o nosso estudo com ênfase nas tecnologias OSI e CORBA.

No segundo capítulo apresentamos os detalhes funcionais da arquitetura CORBA. Sua arquitetura se mostra interessante para o desenvolvimento dos sistemas de gerenciamento de redes, principalmente por sua característica distribuída, onde os dados, no caso os objetos, se encontram distribuídos pela organização. Foi dada mais atenção para a linguagem de definição de interfaces - IDL. Através desta linguagem é possível ser feita a implementação dos programas clientes e servidores em diferentes linguagens, pois os objetos são definidos através de suas interfaces.

No terceiro capítulo foram apresentadas as características do ambiente de gerenciamento OSI, sendo dado especial destaque as ferramentas notacionais, como GDMO e ASN.1. Estas

ferramentas são utilizadas para fazer a definição dos objetos gerenciados, aos quais estão ligados os recursos físicos que devem ser controlados.

No quarto capítulo foi realizado um estudo comparativo entre as duas arquiteturas buscando ressaltar os aspectos positivos e negativos em ambos os ambientes. E identificamos, que apesar da arquitetura CORBA não ter sido criada para atuar nos aspectos de gerência, sua estrutura funcional permite que seja realizada com ela esta tarefa. Principalmente, como no nosso caso, que ela seja a figura principal destinada ao processo de gerência, ou seja, as aplicações são construídas para serem executadas em seu contexto.

A partir do quinto capítulo foram apresentados os detalhes da implementação das ferramentas de conversão. Neste capítulo são relacionados os detalhes da transformação das especificações em ASN.1 para IDL e de GDMO para IDL. São observados alguns pontos onde poderá ocorrer certa perda de informação, pois a notação ASN.1 é extremamente poderosa quanto aos tipos de dados e estruturas que pode manipular. Em particular foi dada ênfase a maneira como a qual o tradutor de GDMO têm acesso as informações obtidas das *templates*, principalmente o que a pessoa do implementador possa fazer com ela.

No sexto capítulo foram apresentados a maneira pela qual os tradutores foram implementados. Foram omitidos alguns detalhes de implementação por motivo de clareza, mas de forma geral o processo de tradução conseguiu atingir os objetivos iniciais para os quais havia sido proposto. Principalmente pelo fato das estruturas das *templates* em GDMO terem seu acesso facilitado pelo implementador de um possível mecanismo interativo de tradução, relacionado muitas vezes como tradução interativa neste documento.

A implementação dos tradutores de ASN.1 para IDL/CORBA e GDMO para IDL/CORBA, mostram que ainda é necessário definir outras estruturas de informação que possibilitem a interoperabilidade completa entre os ambientes a serem gerenciados. Principalmente um mecanismo que possibilite a integração de forma dinâmica entre estes ambientes.

Neste sentido verificamos que os mecanismos para mediação em tempo de execução, em geral, são dependentes de ferramentas de tradução de especificação. É provável que para funcionar adequadamente o *gateway* ou agente-*proxy* necessite conhecer a notação básica no formato original e no formato traduzido. Com este conhecimento, o mediador pode traduzir dinamicamente as mensagens e os serviços de invocação de uma forma para outra. A transformação de mensagens de forma dinâmica pode funcionar em ambas as direções, embora

uma notação tenha sido escolhida como base e a outra tenha sido derivada através do processo de tradução.

Nesta dissertação o principal enfoque, além da tradução de GDMO/ASN.1 para IDL/CORBA, era possibilitar ao implementador do mecanismo de tradução interativa uma ferramenta que pudesse ser tão genérica quanto possível, deixando ao implementador a possibilidade de alterar, e principalmente criar as informações necessárias para realizar a implementação do mecanismo interativo.

A ferramenta necessita de uma interface gráfica que auxilie o implementador no processo de *browsing* das informações das *templates* GDMO e das definições de tipos em ASN.1, facilitando o processo de criação de suas informações em particular. A implementação da linguagem *script* no tradutor de ASN.1 seria de grande valia, pois verifica-se que tal característica pode auxiliar na tradução das especificações em SNMP para IDL, como descrito com detalhes em [OPS95] e [SOU95c]. O desenvolvimento de *scripts* para a tradução de outras especificações IDL, como por exemplo ANSAware e DCE. E principalmente a atualização da linguagem *script* para suporte à procedimentos e funções, possibilitando a criação de uma biblioteca de suporte ao processo de tradução.

Um dos pontos a serem destacados nesta implementação, e que ficou comprovado por [GEN96], é a possibilidade de fazer com que as informações obtidas pelo processo de tradução, seja de GDMO ou ASN.1, poderiam ser armazenadas em uma base de dados comum, e acessíveis através da linguagem *script* aqui apresentada, ou através de uma API de funções bem definidas. Esta abordagem é prática para o mecanismo de tradução interativa, principalmente pelo coberto por [FER98], onde uma integração entre ambas as ferramentas seria possível.

ANEXOS

ANEXO 1 - GRAMÁTICA IDL CORBA

```

<specification> ::= <definition>+
<definition> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <interface> ";"
| <module> ";"
<module> ::= "module" <identifier> "{" <definition>+ "}"
<interface> ::= <interface_dcl>
| <forward_dcl>
<interface_dcl> ::= <interface_header> "{" <interface_body> "}"
<forward_dcl> ::= "interface" <identifier>
<interface_header> ::= "interface" <identifier> [ <inheritance_spec> ]
<interface_body> ::= <export>+
<export> ::= <type_dcl> ";"
| <const_dcl> ";"
| <except_dcl> ";"
| <attr_dcl> ";"
| <op_dcl> ";"
<inheritance_spec> ::= ":" <scoped_name> { "," <scoped_name> }+
<scoped_name> ::= <identifier>
| ":" <identifier>
| <scoped_name> ":" <identifier>
<const_dcl> ::= "const" <const_type> <identifier> "=" <const_exp>
<const_type> ::= <integer_type>
| <char_type>
| <boolean_type>
| <floating_pt_type>
| <string_type>
| <scoped_name>
<const_exp> ::= <or_expr>
<or_expr> ::= <xor_expr>
| <or_expr> "|" <xor_expr>
<xor_expr> ::= <and_expr>
| <xor_expr> "^" <and_expr>
<and_expr> ::= <shift_expr>
| <and_expr> "&" <shift_expr>
<shift_expr> ::= <add_expr>
| <shift_expr> ">>" <add_expr>
| <shift_expr> "<<" <add_expr>
<add_expr> ::= <mult_expr>
| <add_expr> "+" <mult_expr>
| <add_expr> "-" <mult_expr>
<mult_expr> ::= <unary_expr>
| <mult_expr> "*" <unary_expr>
| <mult_expr> "/" <unary_expr>
| <mult_expr> "%" <unary_expr>
<unary_expr> ::= <unary_operator> <primary_expr>
| <primary_expr>
<unary_operator> ::= "-"
| "+"
| "~"
<primary_expr> ::= <scoped_name>
| <literal>
| "(" <const_exp> ")"
<literal> ::= <integer_literal>
| <string_literal>
| <character_literal>
| <floating_pt_literal>
| <boolean_literal>
<boolean_literal> ::= "TRUE"
| "FALSE"
<positive_int_const> ::= <const_exp>
<type_dcl> ::= "typedef" <type_declarator>
| <struct_type>

```

		<union_type>
		<enum_type>
<type_declarator>	::=	<type_spec> <declarators>
<type_spec>	::=	<simple_type_spec>
		<constr_type_spec>
<simple_type_spec>	::=	<base_type_spec>
		<template_type_spec>
		<scoped_name>
<base_type_spec>	::=	<floating_pt_type>
		<integer_type>
		<char_type>
		<boolean_type>
		<octet_type>
		<any_type>
<template_type_spec>	::+=	<sequence_type>
		<string_type>
<constr_type_spec>	::=	<struct_type>
		<union_type>
		<enum_type>
<declarators>	::=	<declarator> { ",", <declarator> }*
<declarator>	::=	<simple_declarator>
		<complex_declarator>
<simple_declarator>	::=	<identifier>
<complex_declarator>	::=	<array_declarator>
<floating_pt_type>	::=	"float"
		"double"
<integer_type>	::=	<signed_int>
		<unsigned_int>
<signed_int>	::=	<signed_long_int>
		<signed_short_int>
<signed_long_int>	::=	"long"
<signed_short_int>	::=	"short"
<unsigned_int>	::=	<unsigned_long_int>
		<unsigned_short_int>
<unsigned_long_int>	::=	"unsigned" "long"
<unsigned_short_int>	::=	"unsigned" "short"
<char_type>	::=	"char"
<boolean_type>	::=	"boolean"
<octet_type>	::=	"octet"
<any_type>	::=	"any"
<struct_type>	::=	"struct" <identifier> "{" <member_list> "}"
<member_list>	::=	<member>+
<member>	::=	<type_spec> <declarators> ";"
<union_type>	::=	"union" <identifier> "switch" "(" <switch_type_spec> ")"
		"{" <switch_body> "}"
<switch_type_spec>	::=	<integer_type>
		<char_type>
		<boolean_type>
		<enum_type>
		<scoped_name>
<switch_body>	::=	<case>+
<case>	::=	<case_label>+ <element_spec> ";"
<case_label>	::=	"case" <const_exp> ":"
		"default" ":"
<element_spec>	::=	<type_spec> <declarator>
<enum_type>	::=	"enum" <identifier> "{" <enumerator> { ",", <enumerator> }* "}"
<enumerator>	::=	<identifier>
<sequence_type>	::=	"sequence" "<" <simple_type_spec> ">" <positive_int_const> ">"
		"sequence" "<" <simple_type_spec> ">"
<string_type>	::=	"string" "<" <positive_int_const> ">"
		"string"
<array_declarator>	::=	<identifier> <fixed_array_size>+
<fixed_array_size>	::=	"[" <positive_int_const> "]"
<attr_dcl>	::=	["readonly"] "attribute" <simple_type_spec> <declarators>
<except_dcl>	::=	"exception" <identifier> "{" <member> "}"
<op_dcl>	::=	[<op_attribute>] <op_type_spec> <identifier>
		<parameter_dcls> [<raises_expr>] [<context_expr>]
<op_attribute>	::=	"oneway"

```

<op_type_spec>      ::= <simple_type_spec>
                       |
                       "void"
<parameter_dcls>    ::= "(" <param_dcl> { "," <param_dcl> }* ")"
                       |
                       "(" ")"
<param_dcl>         ::= <param_attribute> <simple_type_spec> <declarator>
<param_attribute>   ::= "in"
                       |
                       "out"
                       |
                       "inout"
<raises_expr>       ::= "raises" "(" <scoped_name> { "," <scoped_name> }* ")"
<context_expr>      ::= "context" "(" <string_literal> { "," <string_literal> }* ")"

```


ANEXO 2 - DEFINIÇÕES DE DADOS DO TRADUTOR DE GDMO

A estrutura interna do tradutor GDMO representa todas as *templates* analisadas como se fossem classes de objetos. Isto possibilita que o implementador tenha um acesso completo as definições criadas pelos módulos definidos em GDMO.

```
CLASS GDMO
  this                OBJECT
  ManagedObjectList   LIST OF OBJECT
  PackageList         LIST OF OBJECT
  ConditionalPackageList LIST OF OBJECT
  AttributeList       LIST OF OBJECT
  ActionList          LIST OF OBJECT
  NotificationList    LIST OF OBJECT
  ParameterList       LIST OF OBJECT
  NamebindingsList    LIST OF OBJECT
  AttributeGroupList  LIST OF OBJECT
  BehaviourList       LIST OF OBJECT
END

CLASS GDMOClass
  this                OBJECT
  MOname              STRING
  ClassName           STRING
  IDLFileName         STRING
  ParentList          LIST OF OBJECT
  AttributeList       LIST OF OBJECT
  AttributeGroupList  LIST OF OBJECT
  NamebindingsList    LIST OF OBJECT
  ActionList          LIST OF OBJECT
  NotificationList    LIST OF OBJECT
  Oid                 STRING
  External            INTEGER
END

CLASS GDMOAttribute
  this                OBJECT
  AttributeName       STRING
  Syntax              STRING
  Derived             STRING
  MatchesFor          INTEGER
  ReplaceWithDefault  INTEGER
  DefaultValue        STRING
  InitialValue        STRING
  PermittedValue      STRING
  RequiredValue       STRING
  GetReplace          INTEGER
  AddRemove           INTEGER
  Oid                 STRING
  ClassName           STRING
  IDLFileName         STRING
  BehaviourReferenceList LIST OF OBJECT
END

CLASS GDMOPackage
  this                OBJECT
  PackageName         STRING
  AttributeList       LIST OF OBJECT
  ActionList          LIST OF OBJECT
  NotificationList    LIST OF OBJECT
  AttributeGroupList  LIST OF OBJECT
  Conditional         INTEGER
  Oid                 STRING
  BehaviourReferenceList LIST OF OBJECT
END
```



```

CLASS GDMOAttributeGroup
    this                OBJECT
    AttributeGroupName  STRING
    Elements             LIST OF STRING
    Extensions          LIST OF STRING
    Fixed               INTEGER
    Oid                 STRING
    BehaviourReferenceList LIST OF OBJECT
END

CLASS GDMOParameter
    this                OBJECT
    ParameterName       STRING
    Context             INTEGER
    Syntax              OBJECT
    Oid                 STRING
    BehaviourReferenceList LIST OF OBJECT
END

CLASS GDMONotification
    this                OBJECT
    NotificationName    STRING
    InfoSyntax          STRING
    ReplySyntax         STRING
    Oid                 STRING
    BehaviourReferenceList LIST OF OBJECT
END

CLASS GDMOAction
    this                OBJECT
    ActionName          STRING
    Mode                INTEGER
    InfoSyntax          STRING
    ReplySyntax         STRING
    Oid                 STRING
    BehaviourReferenceList LIST OF OBJECT
END

CLASS GDMONameBinding
    this                OBJECT
    BindingName         STRING
    Superior            STRING
    Subordinate         STRING
    NamingAttribute     STRING
    Create              INTEGER
    Delete              INTEGER
    Oid                 STRING
    BehaviourReferenceList LIST OF OBJECT
    IndexLabels         LIST OF STRING
    EntryLabel          LIST OF STRING
END

CLASS GDMOBehaviour
    this                OBJECT
    BehaviourName        STRING
    Definition            STRING
END

```

ANEXO 3 - TEMPLATES GDMO

```
<class-label> MANAGED OBJECT CLASS
  [DERIVED FROM <class-label>
    [, <class-label>]* ;
  ]
  [CHARACTERIZED BY <package-label>
    [, <package-label>]* ;
  ]
  [CONDITIONAL PACKAGES <package-label>
    PRESENT IF condition-definition
    [, <package-label> PRESENT IF condition-definition ]*;
  ]
REGISTERED AS object-identifier;

condition-definition -> delimited-string

<package-label> PACKAGE
  [BEHAVIOUR <behaviour-definition-label>
    [, <behaviour-definition-label>]*;
  ]
  [ATTRIBUTES <attribute-label> propertylist [<parameter-label>]*
    [, <attribute-label> propertylist [<parameter-label>]*];
  ]
  [ATTRIBUTES GROUPS <group-label> [<attribute-label>]*
    [, <group-label> [<attribute-label>]*]*;
  ]
  [ACTIONS <action-label> [<parameter-label>]*
    [, <action-label> [<parameter-label>]*]*;
  ]
  [NOTIFICATIONS <notification-label> [<parameter-label>]*
    [, <notification-label> [<parameter-label>]*]*;
  ]
[REGISTERED AS object-identifier];

propertylist -> [REPLACE-WITH-DEFAULT]
  [DEFAULT VALUE value-specifier]
  [INITIAL VALUE value-specifier]
  [PERMITTED VALUES type-reference]
  [REQUIRED VALUES type-reference]
  [get-replace]
  [add-remove]

value-specifier -> value-reference | DERIVATION RULE
  <behaviour-definition-label>

get-replace -> GET | REPLACE | GET-REPLACE
add-remove -> ADD | REMOVE | ADD-REMOVE

<parameter-label> PARAMETER
  CONTEXT context-type;
  syntax-or-attribute-choice;
  [BEHAVIOUR <behaviour-definition-label>
    [, <behaviour-definition-label>]*;
  ]
[REGISTERED AS object-identifier];

context-type -> context-keyword | ACTION-INFO | ACTION-REPLY |
  EVENT-INFO | EVENT-REPLY | SPECIFIC-ERROR

context-keyword -> type-reference.<identifier>

syntax-or-attribute-choice -> WITH SYNTAX type-reference |
  ATTRIBUTE <attribute-label>
```

```

<name-binding-label> NAME BINDING
  SUBORDINATE OBJECT CLASS <class-label> [AND SUBCLASSES];
  NAMED BY SUPERIOR OBJECT CLASS <class-label>
                                     [AND SUBCLASSES];

  WITH ATTRIBUTE <attribute-label>;
  [BEHAVIOUR <behaviour-definition-label>
    [, <behaviour-definition-label>]*;
  ]
  [CREATE    [create-modifier [, create-modifier ] ]
    [<parameter-label>]*;
  ]
  [DELETE    [delete-modifier]
    [<parameter-label>]*;
  ]
REGISTERED AS object-identifier;

create-modifier -> WITH-REFERENCE-OBJECT |
                  WITH-AUTOMATIC-INSTANCE-NAMING

delete-modifier -> ONLY-IF-NO-CONTAINED-OBJECTS |
                  DELETES-CONTAINED-OBJECT

<attribute-label> ATTRIBUTE
  derived-or-with-syntax-choice;
  [MATCHES FOR qualifier [, qualifier]*;
  ]
  [BEHAVIOUR <behaviour-definition-label>
    [, <behaviour-definition-label>]*;
  ]
  [PARAMETERS <parameter-label>
    [, <parameter-label>]*;
  ]
[REGISTERED AS object-identifier];

qualifier -> EQUALITY | ORDERING | SUBSTRINGS | SET-COMPARISON |
            SET-INTERSECTION

derived-or-with-syntax-choice -> DERIVED FROM <attribute-label> |
                                WITH ATTRIBUTE SYNTAX type-reference

<group-label> ATTRIBUTE GROUP
  [GROUP ELEMENTS <attribute-label>
    [, <attribute-label>]*;
  ]
  [FIXED ;
  ]
  [DESCRIPTION delimited-string;
  ]
REGISTERED AS object-identifier;

<behaviour-definition-label> BEHAVIOUR
  DEFINED AS delimited-string;

<action-label> ACTION
  [BEHAVIOUR <behaviour-defition-label>
    [, <behaviour-defition-label>]*;
  ]
  [MODE CONFIRMED ;
  ]
  [PARAMETERS <parameter-label>
    [, <parameter-label>]*;
  ]
  [WITH INFORMATION SYNTAX type-reference ;
  ]
  [WITH REPLY SYNTAX type-reference ;
  ]
REGISTERED AS object-identifier;

```

```

<notification-label> NOTIFICATION
    [BEHAVIOUR <behaviour-definition-label>
      [, <behaviour-definition-label>]*;
    ]
    [PARAMETERS <parameter-label>
      [, <parameter-label>]*;
    ]
    [WITH INFORMATION SYNTAX type-reference
      [AND ATTRIBUTE IDS <field-name> <attribute-label>
        [, <field-name> <attribute-label>]*
      ] ;
    ]
    [WITH REPLY SYNTAX type-reference ;
    ]
REGISTERED AS object-identifier;

type-reference      ->    <module-name>.<type-name>
value-reference     ->    <module-name>.<value-name>

```


ANEXO 4 - NOTAÇÃO ASN.1

```

ModuleDefinition ::= ModuleIdentifier DEFINITIONS TagDefault "::~="
    BEGIN
        ModuleBody
    END
TagDefault ::= EXPLICIT TAGS | IMPLICIT TAGS | empty
ModuleIdentifier ::= modulereference AssignedIdentifier
AssignedIdentifier ::= ObjectIdentifier Value | empty
ModuleBody ::= Exports Imports AssignmentList | empty
Exports ::= EXPORTS SymbolsExported; | empty
SymbolsExported ::= SymbolList | empty
Imports ::= IMPORTS SymbolsImported; | empty
SymbolsImported ::= SymbolsFromModuleList | empty
SymbolsFromModuleList ::= SymbolsFromModule SymbolsFromModuleList |
    SymbolsFromModule
SymbolsFromModule ::= SymbolList FROM ModuleIdentifier
SymbolList ::= Symbol, SymbolList | Symbol
Symbol ::= typereference | valuereference
AssignmentList ::= Assignment AssignmentList | Assignment
Assignment ::= Typeassignment | Valueassignment
Externaltypereference ::= modulereference "." typereference
Externalvaluereference ::= modulereference "." valuereference
DefinedType ::= Externaltypereference | typereference
DefinedValue ::= Externalvaluereference | valuereference
Typeassignment ::= typereference "::~=" Type
Valueassignment ::= valuereference Type "::~=" Value
Type ::= BuiltinType | DefinedType | Subtype
BuiltinType ::= BooleanType | IntegerType | BitStringType | OctetStringType | NullType
    | SequenceType | SequenceOfType | SetType | SetOfType | ChoiceType |
    SelectionType | TaggedType | AnyType | ObjectIdentifierType |
    CharacterStringType | UsefulType | EnumeratedType | RealType |
NamedType ::= identifier Type | Type | SelectionType
Value ::= BuiltinValue | DefinedValue
BuiltinValue ::= BooleanValue | IntegerValue | BitStringValue |
    OctetStringValue | NullValue | SequenceValue |
    SequenceOfValue | SetValue | SetOfValue | ChoiceValue |
    SelectionValue | TaggedValue | AnyValue |
    ObjectIdentifierValue | CharacterStringValue |
    EnumeratedValue | RealValue
NamedValue ::= identifier Value | Value
BooleanType ::= BOOLEAN
BooleanValue ::= TRUE | FALSE
IntegerType ::= INTEGER | INTEGER {NamedNumberList}
NamedNumberList ::= NamedNumber | NamedNumberList, NamedNumber
NamedNumber ::= identifier(SignedNumber) | identifier(DefinedValue)
SignedNumber ::= number | -number
IntegerValue ::= SignedNumber | identifier
EnumeratedType ::= ENUMERATED {Enumeration}
Enumeration ::= NamedNumber | NamedNumber, Enumeration
EnumeratedValue ::= identifier
RealType ::= REAL
RealValue ::= NumericRealValue | SpecialRealValue
NumericRealValue ::= {Mantissa, Base, Exponent} | 0
Mantissa ::= SignedNumber
Base ::= 2 | 10
Exponent ::= SignedNumber
SpecialRealValue ::= PLUS-INFINITY | MINUS-INFINITY
BitStringType ::= BIT STRING | BIT STRING {NamedBitList}
NamedBitList ::= NamedBit | NamedBitList, NamedBit
NamedBit ::= identifier(number) | identifier(DefinedValue)
BitStringValue ::= bstring | hstring | {IdentifierList} | { }
IdentifierList ::= identifier | IdentifierList, identifier
OctetStringType ::= OCTET STRING
OctetStringValue ::= bstring | hstring
NullType ::= NULL

```

```

NullValue ::= NULL
SequenceType ::= SEQUENCE {ElementTypeList} | SEQUENCE { }
ElementTypeList ::= ElementType | ElementTypeList, ElementType
ElementType ::= NamedType | NamedType OPTIONAL | NamedType DEFAULT Value |
    COMPONENTS OF Type
SequenceValue ::= {ElementValueList} | { }
ElementValueList ::= NamedValue | ElementValueList, NamedValue
SequenceOfValue ::= SEQUENCE OF Type | SEQUENCE
SequenceOfValue ::= {ValueList} | { }
ValueList ::= Value | ValueList, Value
SetType ::= SET{ElementTypeList} | SET { }
SetValue ::= {ElementValueList} | { }
SetOfValue ::= SET OF Type | SET
SetOfValue ::= {ValueList} | { }
ChoiceType ::= CHOICE{AlternativeTypeList}
AlternativeTypeList ::= NamedType | AlternativeTypeList, NamedType
ChoiceValue ::= NamedValue
SelectionType ::= identifier < Type
SelectionValue ::= NamedValue
TaggedType ::= Tag Type | Tag IMPLICIT Type | Tag EXPLICIT Type
Tag ::= [Class ClassNumber]
ClassNumber ::= number | DefinedValue
Class ::= UNIVERSAL | APPLICATION | PRIVATE | empty
TaggedValue ::= Value
AnyType ::= ANY | ANY DEFINED BY identifier
AnyValue ::= Type Value
ObjectIdentifierType ::= OBJECT IDENTIFIER
ObjectIdentifierValue ::= {ObjIdComponentList} | {DefinedValue ObjIdComponentList}
ObjIdComponentList ::= ObjIdComponent | ObjIdComponent ObjIdComponentList
ObjIdComponent ::= NameForm | NumberForm | NameAndNumberForm
NameForm ::= identifier
NumberForm ::= number | DefinedValue
NameAndNumberForm ::= identifier(NumberForm)
CharacterStringType ::= typereference
CharacterStringValue ::= cstring
UsefulType ::= typereference
Subtype ::= ParentType SubtypeSpec | SET SizeConstraint OF Type | SEQUENCE
    SizeConstraint OF Type
ParentType ::= Type
SubtypeSpec ::= (SubtypeValueSet SubtypeValueSetList)
SubtypeValueSetList ::= "|" SubtypeValueSet SubtypeValueSetList | empty
SubtypeValueSet ::= SingleValue | ContainedSubtype | ValueRange | PermittedAlphabet |
    SizeConstraint | InnerTypeConstraint
SingleValue ::= Value
ContainedSubtype ::= INCLUDES Type
ValueRange ::= LowerEndPoint . . UpperEndPoint
LowerEndPoint ::= LowerEndValue | LowerEndValue <
UpperEndPoint ::= UpperEndValue | <UpperEndValue
LowerEndValue ::= Value | MIN
UpperEndValue ::= Value | MAX
SizeConstraint ::= SIZE SubtypeSpec
PermittedAlphabet ::= FROM SubtypeSpec
InnerTypeConstraints ::= WITH COMPONENT SingleTypeConstraint | WITH COMPONENTS
    MultipleTypeConstraints
SingleTypeConstraint ::= SubtypeSpec
MultipleTypeConstraints ::= FullSpecification | PartialSpecification
FullSpecification ::= {TypeConstraints}
PartialSpecification ::= { . . . , TypeConstraints}
TypeConstraints ::= NamedConstraint | NamedConstraint, TypeConstraints
NamedConstraint ::= identifier Constraint | Constraint
Constraint ::= ValueConstraint PresenceConstraint
ValueConstraint ::= SubtypeSpec | empty
PresenceConstraint ::= PRESENT | ABSENT | empty | OPTIONAL

```

ANEXO 5 - EXEMPLO DE TRADUÇÃO GDMO PARA IDL

```

/*****\
|
|   equipment.idl gerado a partir de traduz.ccg
|
| *****/

// Conforme X/Open Forum

#include "ASN1Types.idl"
#include "CMISError.idl"
#include "Notification.idl"
#include "MultipleReply.idl"

/*
**  Interfaces
*/

interface equipment : X0721::top {

    M3100ASN::NameTypeType    equipmentId_get()
        raises ( CMIP-ATTRIBUTE-ERRORS );

    M3100ASN::ReplaceableType replaceable_get()
        raises ( CMIP-ATTRIBUTE-ERRORS );

};

/*
**  Fim da geracao de codigo
*/

/*****\
|
|   network.idl gerado a partir de traduz.ccg
|
| *****/

// Conforme X/Open Forum

#include "ASN1Types.idl"
#include "CMISError.idl"
#include "Notification.idl"
#include "MultipleReply.idl"

/*
**  Interfaces
*/

interface network : X0721::top {

    M3100ASN::NameTypeType    networkId_get()
        raises ( CMIP-ATTRIBUTE-ERRORS );

};

/*
**  Fim da geracao de codigo
*/

/*****\
```



```

|
| connection.idl gerado a partir de traduz.ccg
|
\*****/

// Conforme X/Open Forum

#include "ASN1Types.idl"
#include "CMISError.idl"
#include "Notification.idl"
#include "MultipleReply.idl"

/*
** Interfaces
*/

interface connection : connectivity.h::connectivity {

    M3100ASN::NameTypeType    connectionId_get()
        raises ( CMIP-ATTRIBUTE-ERRORS );

};

/*
** Fim da geracao de codigo
*/

```

ANEXO 6 - SCRIPT DE TRADUÇÃO: TRADUZ.SCT

(parcial)

```

/*****
|
| $PRINT(MOName, ".cc")$ gerado a partir de traduz.sct
|
| *****/
// Conforme X/Open Forum

#include "ASN1Types.idl"
#include "CMISError.idl"
#include "Notification.idl"
#include "MultipleReply.idl"

/*
** Interfaces
*/

$

// Script GLOBALS
GLOBAL INTEGER parentCount
GLOBAL OBJECT parent
GLOBAL OBJECT attribute
GLOBAL OBJECT action

OBJECT element

PRINT( "interface ", MOName )

IF( MOName == "top" )
    PRINT( " : ManagedObject" )

    // heranca...
    parentCount = 0
    FOREACH parent IN ParentList
    {
        IF( parentCount == 0 )
            PRINT( " : " )

        PRINT( parent.HeaderFileName, ":", parent.MOName )
        parentCount = parentCount + 1

        IF( parentCount != LISTCOUNT(ParentList) )
            PRINT( ", " )
    }
    PRINT( " { \n\n" )
    //
    // Colocar parametros e retorno... -- verificar tipos padroes !!! --
    //
    FOREACH attribute IN AttributeList
    {
        IF( attribute.GetReplace != 0 || attribute.ReplaceWithDefault != 0 ) // GET ou GET-
REPLACE ou REPLACE-WITH-DEFAULT
        {
            IF( attribute.GetReplace == 1 || attribute.GetReplace == 3 ) // GET ou GET-
REPLACE
            {
                PRINT( "\t", attribute.HeaderFileName, ":", attribute.Syntax, "Type" )
            }
            ELSE IF( attribute.GetReplace == 2 || attribute.ReplaceWithDefault == 1 )
            {
                PRINT( "\tvoid" )
            }
            PRINT( "\t", attribute.AttributeName )
            IF( attribute.GetReplace == 1 || attribute.GetReplace == 3 ) // GET ou GET-
REPLACE
            {
                PRINT( " get()\n" )
            }
        }
    }
}

```


ANEXO 7 - ARQUIVOS ESPECIFICADOS PELO X/OPEN

```

/*****
ASN1Types.idl
*****/

#ifndef ASN1Type_idl
#define ASN1Type_idl

typedef any                ASN1_ANY;
typedef octet              ASN1_Null;
typedef long               ASN1_INTEGER;
typedef long[2]            ASN1_Integer64;
typedef sequence<octet>    ASN1_BitString;
typedef sequence<octet>    ASN1_OctetString;
typedef string             ASN1_ObjectIdentifier;
typedef double             ASN1_REAL;
typedef sequence<unsigned short> ASN1_BMPString;
typedef sequence<octet>    ASN1_GeneralString;
typedef sequence<octet>    ASN1_GraphicString;
typedef sequence<octet>    ASN1_ISO64String;
typedef sequence<long>     ASN1_UniversalString;
typedef sequence<octet>    ASN1_IA5String;
typedef string             ASN1_NumericString;
typedef string             ASN1_PrintableString;
typedef string             ASN1_TeletexString;
typedef string             ASN1_T61String;
typedef string             ASN1_VideotexString;
typedef string             ASN1_VisibleString;
typedef ASN1_VisibleString ASN1_GeneralizedTime;
typedef ASN1_VisibleString ASN1_UTCTime;
typedef ASN1_GraphicString ASN1_ObjectDescriptor;

struct ASN1_External {
    ASN1_ObjectIdentifier syntax;
    any                    data_value;
}

const ASN1_NullValue = 0;

#endif /* ASN1Type_idl */

/*****
CMISError.idl
*****/

// ROSE exceptions

exception ROSEDuplicateInvocation      { DuplicateInvocationType  error_info; };
exception ROSEMistypedArgument         { MistypedArgumentType    error_info; };
exception ROSEResourceLimitation       { ResouceLimitationType  error_info; };
exception ROSEUnrecognizedOperation    { UnrecognizedOperationType error_info; };

// CMIP exceptions

exception CMIPaccessDenied              { AccessDeniedType      error_info; };
exception CMIPclassInstanceConflict     { ClassInstanceConflictType error_info; };
exception CMIPcomplexityLimitation      { ComplexityLimitationType error_info; };
exception CMIPgetListError              { GetListErrorType      error_info; };
exception CMIPsetListError              { SetListErrorType      error_info; };
exception CMIPinvalidArgumentValue      { InvalidArgumentValueType error_info; };
exception CMIPinvalidAttributeValue     { InvalidAttributeValueType error_info; };
exception CMIPinvalidScope              { InvalidScope          error_info; };
exception CMIPinvalidFilter             { InvalidFilter          error_info; };

```

```

/*****
MultipleReply.idl
*****/

exception UsingMR { long request_id;    CosTypeEventChannelAdmin admin; };

struct MultipleReply {
    long      request_id;
    object src_object;
    any       value;
};

interface CMIPMultipleReply {
    void      endReply( in long request_id );
};

/*****
dummy.idl
*****/

#ifndef dummy_idl
#define dummy_idl

    typedef long      dummy;

#endif /* dummy_idl */

#define CMIPError_idl

interface ManagedObject {

    exception CMIP_ATTRIBUTE_ERRORS { };
    exception CMIP_ACTION_ERRORS { };
    exception CMIP_NOTIFICATION_ERRORS { };

    sequence<any>      GetAttributeGroup();
    void              SetAttributeGroupToDefault();

};

```

GLOSSÁRIO

Atributos

Um *atributo* é um valor mantido pelos objetos numa classe. Cada atributo tem um valor para cada instância de objeto. Diferentes instâncias de objetos podem possuir os mesmos ou diferentes valores para um dado atributo. Cada nome de atributo é único dentro da classe.

Abstração

O conceito de *abstração* significa focalizar o essencial, isto é, os aspectos inerentes de uma entidade e ignorando suas propriedades casuais. O uso da abstração durante a análise significa trabalhar somente com conceitos do domínio da aplicação, não fazendo decisões sobre o projeto e implementação, antes do problema ser entendido.

Classes

Uma *classe* de objetos é um grupo de objetos que possuem propriedades (atributos), comportamento (operações), e relacionamentos com outros objetos com características semelhantes. A abreviação *classe* é muitas vezes usada ao invés de *classe de objeto*.

Os objetos pertencentes a uma classe, compartilham uma semântica comum, além dos requisitos de atributo, comportamento e relacionamento comuns. As definições comuns como o nome da classe e nome de atributos, são atribuídas por classe, ao invés de por instância.

Compartilhamento

As técnicas de orientação a objeto promovem compartilhamento em diversos níveis diferentes, herdando de ambos, estrutura de dados e procedimentos, permitindo estruturas comuns para serem compartilhadas entre diversas subclasses semelhantes sem redundância. O compartilhamento de código utilizando herança é um das principais vantagens das linguagens orientadas à objetos.

Comportamento

Uma mudança de estado dentro de um objeto.

Concorrência na Interface

A propriedade que define a capacidade de uma interface em aceitar elementos de protocolo a qualquer momento.

Encapsulamento

Conhecido também como ocultamento de informações, consiste na separação de aspectos externos de um objeto, que são acessíveis a outros objetos, dos detalhes da implementação interna, que são ocultados para outros objetos. Graças ao encapsulamento, implementação de um objeto, pode ser modificada sem afetar as aplicações que a utilizam. Encapsulamento é então a habilidade de combinar estruturas de dados e procedimentos em uma entidade simples.

Estado

Uma condição estável interna de um objeto.

Exceção

Uma condição que interrompe uma execução normal. Devendo ser tratada, pois pode representar um erro.

Encapsulamento

A propriedade de um objeto, segundo a qual o seu estado pode somente ser acessado por meio de sinais pré-definidos e que cruzam o limite do objeto, ou por meio de relacionamentos pré-definidos entre objetos.

Evento

Um comportamento autônomo de um objeto; ele pode ser sinalizado pela emissão de uma notificação.

Herança

As classes definidas também podem apresentar aspectos semelhantes, ou seja, o comportamento descrito por uma classe engloba aspectos já definidos em outra. Neste caso, através do mecanismo de herança, pode-se derivar classes a partir de classes bases. A classe derivada herda os atributos e métodos da classe ou classes que lhe servirem de base, dependendo do mecanismo de herança.

Instanciação de Objeto

O processo pela qual instâncias de objetos são criadas.

Interface Aberta

Uma interface que suporta um protocolo público, onde se pode realizar testes de conformidade.

Invocação

Um sinal, possivelmente parametrizado, requisitando que uma operação seja realizada por um objeto.

Métodos

Um *método* é a implementação de uma operação para uma classe. Quando uma operação é descrita em diversas classes, é importante que todos os métodos tenham a mesma assinatura - número e tipos de argumentos e o tipo do valor resultante.

Notificação

Um sinal, possivelmente parametrizado, indicando que ocorreu um evento.

Objeto

Um objeto é definido como um conceito, abstração, ou coisa com limites bem definidos que tenha significado para o problema que se quer resolver. Quando se define um objeto, promove-se um entendimento do mundo real e o fornecimento de uma base prática para implementação em computadores. A criação de uma estrutura baseada em objetos, depende muito da natureza do problema, sendo que podem haver várias representações válidas para cada objeto.

Todo objeto possui uma *identidade*, isto é, os objetos são distinguidos por sua existência e não pelas propriedades descritivas que eles podem ter. Um objeto pode ser uma simples coisa, ou muitas vezes ele pode se referir a um grupo de coisas similares. Quando é necessário uma referência exata à um determinado objeto, é usado o termo *instância do objeto*.

Operação

Uma *operação* é uma função que pode ser aplicada para, ou por um objeto em uma classe. Todos os objetos de uma classe compartilham as mesmas operações. O comportamento da operação depende da classe para qual é destinada.

Um objeto conhece sua classe e herda dela a implementação da operação. A mesma operação pode ser aplicada para diferentes classes. Uma operação é polimórfica, isto é, a mesma operação leva diferentes formas em diferentes classes.

Protocolo

Um conjunto de regras sintáticas e semânticas para troca de informações. Um protocolo ao qual são adicionadas invocações, notificações e respostas.

Resposta

Um sinal, possivelmente parametrizado, indicando uma resposta para uma invocação ou notificação; respostas são freqüentemente classificadas como normais ou exceções.

Parâmetro

Um valor de dado que obedece a sintaxe de um protocolo determinado.

Qualificador

Um semântica adicional aplicada a um atributo (com respeito à acessibilidade ou valores default).

BIBLIOGRAFIA

- [AHO85] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1985.
- [BBAN97] Bela Ban. *A Generic Management Model For CORBA, CMIP and SNMP*. IBM Zurich Research Laboratory, abril, 1997.
- [BRISA93] BRISA, *Gerência de Redes: Uma abordagem de sistemas abertos*. Makron Books, 1993.
- [BRISA94] BRISA, *Arquitetura de Redes de Computadores*. Makron Books, 1994.
- [CORBA91] OMG. *The Common Object Request Broker: Architecture and Specification*. Revision 1.1, Object Management Group. December 1991.
- [CORBA95] OMG. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, Object Management Group. July 1995.
- [CORBA97] OMG. *The Common Object Request Broker: Architecture and Specification*. Revision 2.1, Object Management Group. August 1997.
- [CORBA98] OMG. *The Common Object Request Broker: Architecture and Specification*. Revision 2.2, Object Management Group. February 1998.
- [COSS95] OMG. *Common Object Services Specification*. Object Management Group. Document Number: 95.3.31, March 1995.
- [FER98] Marcelo Ferrasa. *Desenvolvimento de um Mecanismo para Prover Interoperabilidade entre Domínios CORBA e OSI*. Proposta de Dissertação de Mestrado, submetida para avaliação. CEFET-PR, 1998.
- [FFP95] Marcelo Ferrasa, Dierone C. Foltran Jr. Eliane Pantaleão. *CORBA: Uma Visão Geral*. Anais do 3º Encontro Interuniversitário de Informática, PUC-PR, setembro, 1995.
- [GEN96] Guy H. Genilloud. *Towards a Distributed Architecture for Systems Management*. Tese de Doutorado, École Polytechnique Fédérale de Lausanne. 1996.
- [GEN97] Guy H. Genilloud. *Integrating the OSI and the CORBA Architectures for System Management*. Proceedings of the CORBA Management Workshop at the OMG TC Meeting in Dublin, Dublin, Ireland, 1997.

- [IIOP95] OMG Tc Document. *CORBA 2.0/Interoperability*. Document Number 95.3.xx, March, 1995.
- [JIDL97] OMG Tc Document. *IDL to JAVA Language Mapping*. Document Number 97.03.08, March, 1997.
- [JOHN85] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. Computing Science Technical Report, 1985.
- [LEV92] John R. Levine, Tony Mason, Doug Brown. *Lex & Yacc*. O'Reilly & Associates, Inc. Canadá, 1992.
- [LM95] Luiz O.B. Lento, Edmundo R.M. Madeira. *Um Esquema para Acessar Objetos em Ambientes Distribuídos*. Anais do Simpósio Brasileiro de Redes de Computadores, abril 1995.
- [LS85] Mark E. Lesk, Eric Schmidt. *Lex: A Lexical Analyser Generator*. Computing Science Technical Report, 1985.
- [M.3000] International Telecommunication Union – “Information Technology - Open Systems Interconnection - *Overview of TMN Standards*”, ITU-T, 1993.
- [M.3100] International Telecommunication Union - “Information Technology - Open Systems Interconnection - *Generic Network Information Model*”, ITU-T, 1993.
- [MAZ95] Subrata Mazumdar. *Inter-Domain Management: CORBA, OSI, SNMP*. IBM T. J. Watson Research Center, 1995.
- [OH95] Robert Orfali, Dan Harkey. *Cliente/Servidor com Objetos Distribuídos*. Byte Brasil, maio, 1995.
- [OHE95] Robert Orfali, Dan Harkey, Jeri Edwards. *Computação Cliente/Servidor Intergalática*. Byte Brasil, maio, 1995.
- [OMA92] OMG. *Object Management Architecture Guide*, Revision 2.0, Second Edition, OMG TC Document 92.11.1, September, 1992.
- [OPS95] X/Open Company Limited. *Inter-Domain Management Specifications: Specification Translation*. X/Open Company, Draft, April, 1995.
- [OSI 7498] Information Processing Systems - Open Systems Interconnection - *Basic Reference Model, International Organization for Standardization*, outubro 1984.
- [OSIMIS93] Department of Computer Science. *OSIMIS GDMO Compiler user manual*. University College London, agosto, 1993.
- [POST94] PostModern Computing Technologies, Inc. *ORBeLine: User's Guide*. 1994.
- [RUM91] James Rumbaugh et al. *Object-Oriented Modeling and Design*. General Electric Research and Development Center, Schenectady, New York. Prentice Hall, 1991.
- [SAM93] Michael Sample. *Snacc 1.1: A High Performance ASN.1 to C/C++ Compiler*. University of British Columbia, 1993.
- [SLO94] Morris Sloman. *Network and Distributed Systems Management*. Addison-Wesley, 1994.

- [SOU94a] Nader Soukouti, *Automatic Translation of GDMO Classes to C++ Classes*. IEEE, 1994.
- [SOU94b] Nader Soukouti, *Using CORBA to Manage OSI*. Draft 1.0, 1994.
- [SOU95c] Nader Soukouti, *Contribution aux travaux de l'XOPEN et le NMF sur le problème de l'interfonctionnement entre CORBA, CMIP, SNMP*. Tese de Doutorado, L'Universite Pierre et Marie Curie, Paris, outubro, 1995.
- [X.208] Information Processing Systems - Open Systems Interconnection - *Specification of Abstract Syntax Notation One (ASN.1)*, International Organization for Standardization, 1987.
- [X.209] Information Processing Systems - Open System Interconnection - *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, International Organization for Standardization, 1987.
- [X.700] International Telecommunication Union - "*Management Framework Definition for Open Systems Interconnection (OSI) for CCITT Applications*", ITU-T, 1992.
- [X.710] International Telecommunication Union - "*Common Management Information Protocol Definition*", ITU-T, 1991.
- [X.711] International Telecommunication Union - "*Common Management Information Service Definition*", ITU-T, 1991.
- [X.720] International Telecommunication Union - "Information Technology - Open Systems Interconnection - *Structure of Management Information: Management Information Model*", ITU-T, 1992.
- [X.721] International Telecommunication Union - "Information Technology - Open Systems Interconnection - *Structure of Management Information: Definition of Management Information*", ITU-T, 1992.
- [X.722] International Telecommunication Union - "Information Technology - Open Systems Interconnection - *Structure of Management Information: Guidelines for the Definition of Managed Object*", ITU-T, 1992.



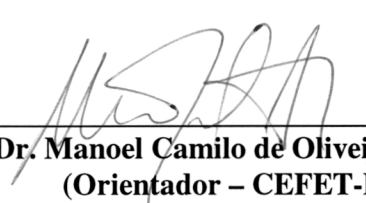
MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO PARANÁ
CURSO DE PÓS-GRADUAÇÃO EM ENG^a ELÉTRICA E INFORMÁTICA INDUSTRIAL

“Desenvolvimento de uma Ferramenta para Tradução das Especificações GDMO / OSI para IDL / CORBA”

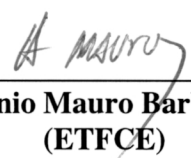
por

Dierone Cesar Foltran Jr.

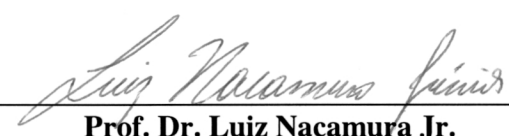
Esta Dissertação foi apresentada no dia 09 de Novembro de 1998, como requisito parcial para a obtenção do título de MESTRE EM CIÊNCIAS - Área de Concentração: Telemática. Aprovada pela Banca Examinadora composta pelos professores:



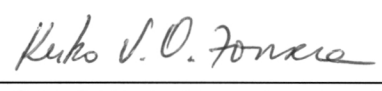
Prof. Dr. Manoel Camilo de Oliveira Penna Neto
(Orientador – CEFET-PR)



Prof. Dr. Antonio Mauro Barbosa de Oliveira
(ETFCE)

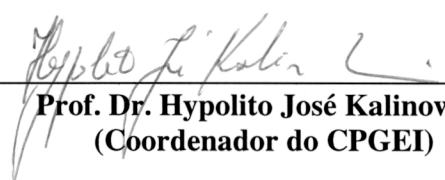


Prof. Dr. Luiz Nacamura Jr.
(CEFET-PR)



Prof^a. Dr^a. Keiko Veronica Ono Fonseca
(Suplente-CEFET-PR)

Visto e aprovado para impressão:



Prof. Dr. Hypolito José Kalinowski
(Coordenador do CPGEI)



MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DO PARANÁ
CURSO DE PÓS-GRADUAÇÃO EM ENG^a ELÉTRICA E INFORMÁTICA INDUSTRIAL

“Desenvolvimento de uma Ferramenta para Tradução das Especificações GDMO / OSI para IDL / CORBA”

por

Dierone Cesar Foltran Jr.

Esta Dissertação foi apresentada no dia 09 de Novembro de 1998, como requisito parcial para a obtenção do título de MESTRE EM CIÊNCIAS - Área de Concentração: Telemática. Aprovada pela Banca Examinadora composta pelos professores:

Prof. Dr. Manoel Camilo de Oliveira Penna Neto
(Orientador – CEFET-PR)

Prof. Dr. Antonio Mauro Barbosa de Oliveira
(ETFCE)

Prof. Dr. Luiz Nacamura Jr.
(CEFET-PR)

Prof^a. Dr^a. Keiko Veronica Ono Fonseca
(Suplente-CEFET-PR)

Visto e aprovado para impressão:

Prof. Dr. Hypolito José Kalinowski
(Coordenador do CPGEI)



originais fornecidos pelo autor

RESUMO:

Atualmente existem diversas tecnologias atuando no serviço de gerência de sistemas, entre elas destacamos o uso do modelo OSI e CORBA. Estas tecnologias podem coexistir dentro de uma organização. Para que esta coexistência seja possível é necessário que as informações entre elas sejam compartilhadas. A proposta deste trabalho é uma ferramenta que possibilite ao gerente destes sistemas, possivelmente seu implementador, a transformação das especificações de gerência do domínio OSI para o domínio CORBA, ou seja, a transformação das especificações em GDMO/OSI para IDL/CORBA.

PALAVRAS-CHAVE

IDL - CORBA / GDMO - OSI / Interoperabilidade

ÁREA / SUB-ÁREA DE CONHECIMENTO

Telemática / Sistemas Distribuídos

1996

Nº: 138

RESUMO:

Atualmente existem diversas tecnologias atuando no serviço de gerência de sistemas, entre elas destacamos o uso do modelo OSI e CORBA. Estas tecnologias podem coexistir dentro de uma organização. Para que esta coexistência seja possível é necessário que as informações entre elas sejam compartilhadas. A proposta deste trabalho é uma ferramenta que possibilite ao gerente destes sistemas, possivelmente seu implementador, a transformação das especificações de gerência do domínio OSI para o domínio CORBA, ou seja, a transformação das especificações em GDMO/OSI para IDL/CORBA.

PALAVRAS-CHAVE

IDL - CORBA / GDMO - OSI / Interoperabilidade

ÁREA / SUB-ÁREA DE CONHECIMENTO

Telemática / Sistemas Distribuídos

1998

Nº: 138