Especificação e Implementação de *Templates* GDMO e Tipos ASN.1 para Linguagem C++

Karina May[1]
karina@inf.ufsc.br

Patrícia Ropelato[2]
ropelatt@dcc.unicamo.br

Rivalino Matias Júnior[3]
rivalino@inf.ufsc.br

Elizabeth Sueli Specialski[4]
beth@inf.ufsc.br

Laboratório de Integração Software e Hardware
Universidade Federal de Santa Catarina
Florianópolis -SC

RESUMO

Este artigo descreve a especificação e implementação de regras para o mapeamento de *templates* GDMO (*Guidelines for the Definition of Managed Objects*) [1], além da implementação dos tipos ASN.1, em linguagem de programação C++ [2].

Estas regras serão utilizadas no mapeamento *template* - GDMO para classes (**class**) em C++, linguagem utilizada em todo o projeto da Plataforma de Gerência da Rede Local UFSC, a qual segue o modelo de gerenciamento de redes OSI.

ABSTRACT

This work describes specification and implementation of rules for the mapping of GDMO templates (Guidelines for the Definition of Managed Objects) [1], in addition of implementation the ASN.1 types, using the programming language C++[2].

*These rules will be used in the mapping templates - GDMO for class (**class**) in C++, language used in UFSC Local Network Management Platform, based in the OSI network management model.*

PALAVRAS CHAVE

Gerenciamento de Redes, Plataforma de Gerenciamento OSI, Modelo de Gerência OSI, ASN.1, GDMO.

---

[1] Aluna do Curso de Graduação em Ciências da Computação da UFSC
[2] Aluna do Curso de Pós-Graduação em Ciências da Computação da UNICAMP
[3] Aluno do Curso de Pós-Graduação em Ciências da Computação da UFSC
[4] Professora do Departamento de Informática e de Estatística da UFSC - Mestre em Ciências da Computação (UFRGS - Porto Alegre - 1981)

1. Introdução

Por menor e mais simples que seja, uma rede de computadores, esta necessita de um processo de gerenciamento. Este gerenciamento pode ser realizado de forma manual ou automatizado, dependendo do tamanho e complexidade do ambiente de rede.

A atividade de gerenciamento de redes heterogêneas constitui uma tarefa complexa, dado o número e diversidade de equipamentos existentes. Para garantir a interoperabilidade entre diferentes sistemas de gerenciamento de redes, tais sistemas precisam ter uma visão comum da informação de gerenciamento. Para possibilitar tal visão, a estratégia consiste em definir plataformas de gerenciamento que sejam independentes de fornecedores e que apresentem uma funcionalidade adequada para o controle dos elementos da rede.

O escopo deste trabalho está relacionado com uma parte do projeto para especificação e implementação de uma Plataforma de Gerência de Redes, a qual segue o modelo de gerenciamento OSI, sendo esta desenvolvida por diversos pesquisadores do Grupo de Redes do Laboratório de Integração Software e Hardware (LISHA) da UFSC.

Este trabalho tem como principal objetivo a implementação do GDMO (*Guidelines for the Definition of Managed Objects*) [1], que define uma forma padronizada para a modelagem de objetos gerenciados em uma plataforma de gerência de redes tendo como fundamento a AOO (Abordagem Orientada a Objetos), de acordo com o modelo de gerenciamento definido pela ISO (*International Organization for Standardization*).

Além disto foram também implementados os tipos simples e os tipos construtores da Notação de Sintaxe Abstrata ASN.1 (*Abstract Syntax Notacion One*) [3], sendo estes essenciais na implantação dos *templates* GDMO.

2. Especificação e Implementação de Regras de Mapeamento GDMO-C++

As Regras para Definição de Objetos Gerenciados (GDMO), especificam como as informações de gerenciamento serão definidas e quais ferramentas notacionais serão usadas em tais definições de classe de objeto gerenciado. Incluído no GDMO, estão os *templates* para definições de informações de gerenciamento. Estes *templates* são formatos padrões para a documentação de *name bindings*, definições de classe de objeto gerenciados e seus componentes, tais como pacotes, atributos, grupos de atributos, parâmetros, comportamentos, ações e notificações [1].

O GDMO é usado por projetistas de aplicações de gerenciamento que irão definir classes de objetos gerenciados para aplicações em uma plataforma de gerência de redes baseada no modelo de Gerência OSI.

A especificação e implementação das regras do GDMO, realizadas neste trabalho, estão voltadas para um desenvolvimento futuro de um compilador GDMO - C ++. O usuário fornecerá os dados para a definição da classe, a qual poderá ser especificada através de uma interface gráfica, e o compilador gerará o código correspondente, como ilustra a figura 2.1.

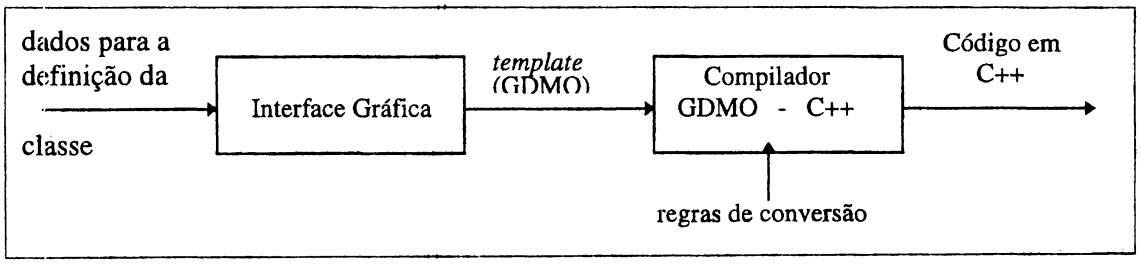

Figura 2.1 Compilador GDMO - C++.

3. Especificação e Implementação dos *Templates*

No contexto deste trabalho, foram implementados processos de mapeamento para os templates de classes, pacotes, atributos, ações, notificações e comportamentos. Os demais *templates*, isto é, dos parâmetros, grupos de atributos e *name bindings* deverão ser implementados em trabalhos futuros.

3.1. *Template* da Classe de Objeto Gerenciado

O *template* da classe de objeto gerenciado forma a base da definição formal de um objeto gerenciado, como mostrado na figura 3.1.

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

Figura 3.1 *Template* da Classe de Objeto Gerenciado.

Cada classe de objeto gerenciado define a(s) superclasse(s) da qual ela foi derivada. A subclasse herda todas as características da(s) sua(s) superclasse(s). A definição de uma subclasse pode adicionar características (especialização) mas não pode remover qualquer característica da superclasse. No *template* acima, esta herança é definida pelo construtor **DERIVED FROM** seguido do nome(s) da(s) classe(s).

A definição da classe de objeto gerenciado possui pacotes obrigatórios e/ou condicionais que incluem definições de comportamento, atributos, grupos de atributos, ações, notificações e parâmetros. O que diferencia pacotes obrigatórios de condicionais, é que este último possui uma condição especificada que determinará se o pacote estará presente ou não na instância da classe. No *template* da classe, os pacotes obrigatórios são definidos por **CHARACTERIZED BY** e os condicionais por **CONDITIONAL PACKAGES**, ambos seguidos do(s) nome(s) do(s) pacote(s). No caso do pacote condicional, a condição é dada por **PRESENT IF**.

3.2. *Template* do Pacote

Este *template* é utilizado na definição de um pacote. Um pacote consiste da combinação de definições de comportamento, atributos, grupos de atributos, operações, notificações e parâmetros a serem definidos pela inserção subsequente dentro do *template* da classe de objeto gerenciado, abaixo das construções **CHARACTERIZED BY** ou **CONDITIONAL PACKAGES**. O *template* do pacote é apresentado na figura 3.2.

```
<package-label> PACKAGE
   [ BEHAVIOUR      <behaviour-definition-label>   [,<behaviour-definition-label>]*;
   ]
   [ ATTRIBUTE      <attribute-label> propertylist  [<parameter-label>]*
                    [,<attribute-label> propertylist [,<parameter-label>]*]*;
   ]
   [ ATTRIBUTE 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];
supporting productions
propertylist     →    [REPLACE-WITH-DEFAULT]
                      [DEFAULT VALUE            value-specifier]
                      [INITIAL VALUE            value-specifier]
                      [PERMITTED VALUES         type-reference]
                      [REQUIRED VALUES type-reference]
                      [get-replace]
                      [add-replace]
value-specifier  →    value-reference | DERIVATION RULE <behaviour-definition-label>
get-replace      →    GET | REPLACE | GET-REPLACE
add-remove       →    ADD | REMOVE | ADD-REMOVE
```

Figura 3.2. *Template* de Package

3.3. *Template* do Atributo

O *template* do atributo apresenta a seguinte estrutura:

```
<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] ;
```

Figura 3.3 *Template* de Atributo

3.3.1. Implementação dos Atributos

Os atributos foram implementados na linguagem C++ como classes (**class**) e estão organizados em hierarquia de herança.
A classe **Attr_Types**, localizada no topo da árvore, é a classe base para todos os tipos de atributos, portanto não é instanciável, ou seja, é uma classe abstrata. Segue abaixo a descrição da classe :

```
# include "global.h"
#ifndef _Attr_Typ_
#define _Attr_Typ_

class Attr_Types{
        private:
          void *value;
          CS *cs;
        protected:
          char *M_Rules;
          void set_value(void *val){value=val;};
          void get_value(void **val){*val=value;};
          virtual int_buid_cs( CS *_cs_ptr){;};
          virtual int_buid_if( CS *_cs_ptr, void *_if_ptr){;};
          virtual int_copyFrom_cs( CS *_cs_ptr){;};
          virtual int_copyTo_cs( CS *_cs_ptr){;};
        public:
          int buid_cs (CS *cs_ptr)        {return (_buid_cs(cs_ptr)); }
          int buid_if (CS *cs_ptr,void*if_ptr){return (_buid_if(cs_ptr,if_ptr)); }
          int getCS (CS *cs_ptr){return )_copyFrom_cs(cs_ptr)); }
          virtual int get          (void **val){;};
          virtual int add          (void *val){;};
          virtual int remove       (void *val){;};
          virtual int replace      (void *val){;};
          virtual int Get          (void **val){;};
          virtual int Add                (void *val){;};
          virtual int Remove       (void *val){;};
          virtual int Replace      (void *val){;};
          virtual int Replace      DFLT(void){;};
          virtual int match(Matching_Rules ruleM, void *val,Substring_Rules ruleS=0) {;};
          virtual Boolean IsPermitted(void *val)=0;
          virtual Boolean IsRequired(void *val)=0;
};
# endif
```

Na variável **value** será armazenado o valor do atributo propriamente dito. A variável **CS** é um ponteiro para uma estrutura que representa a sintaxe concreta [3]. **M_Rules** é um ponteiro para caracteres numéricos '0's e '1's onde cada posição [1-7], indica a presença (caracter '1') ou não (caracter '0') da regra, onde esta **string** é definida pelo usuário quando da definição do atributo. As sete posições deste **string** estão baseadas na variável **Matching_Rules** que possui as sete regras de comparação definidos em [4].

A função **set_value** atribui um valor para **value** e a função **get_value** retorna o endereço de value.

As funções **_build_cs**, que converte o **value** em sintaxe concreta, **_build_if**, que converte a sintaxe concreta na forma interna, **copyFrom_cs** e **copyTo_cs** são funções virtuais e portanto serão implementadas pelos tipos de atributos. As funções **_build_cs** (que converte o **value** em sintaxe

concreta), **_build_if** (que converte a sintaxe concreta na forma interna), **copyFrom_cs** e **copyTo_cs** são funções virtuais e, portanto, são implementadas pelos tipos de atributos.

As funções **get, add, remove** e **replace** são funções virtuais e são implementadas pelos tipos de dados ASN.1 derivados diretamente da classe **Attr_Types**. As funções **Get, Add, Remove** e **ReplaceDFLT** também são funções virtuais, mas somente serão implementadas pelo atributo que o usuário definir.

A função **match** é implementada pelos tipos de dados ASN.1, sendo que cada tipo de dado possui um **match** com regras de comparação que podem ser aplicadas sobre este tipo e, no caso de ser permitido a comparação **SubString**, os três tipos de ordens - **Initial, Any** e **Final** - também serão implementadas.

As funções virtuais **IsPermitted** e **IsRequired** e o conjunto de valores permitidos e requeridos são implementados pelos tipos de dados ASN.1. Estas funções recebem um valor, consultam o conjunto de valores permitidos e requeridos e retornam TRUE ou FALSE caso a consulta tenha sucesso ou não.

3.3.2. Os Tipos ASN.1 Implementados

Todos os tipos de dados ASN.1, dos quais serão derivados os atributos definidos pelo usuário, são derivados, por sua vez, diretamente da classe base **Attr_Types**.

Cada tipo implementado corresponde a uma classe em C++ como é mostrado na tabela 3.1.

TIPO ASN.1	DESCRIÇÃO	C++
INTEGER	Intervalo dependente da máquina	class INTEGER
REAL	Intervalo dependente da máquina	class REAL
NULL	nulo	class _NULL
BOOLEAN	TRUE ou FALSE	class BOOLEAN
NUMERICSTRING	caracteres de 0 a 9 e branco	class NUMERICSTRING
OCTECTSTRING	qualquer caracter	class OCTECTSTRING
PRINTABLESTRING	caracteres : 0 - 9 A-Z a-z branco , ( ) + - . / : =	class PRINTABLESTRING
SEQUENCE	define uma lista ordenada de zero ou mais elementos e cada um é definido por um tipo ASN.1	class SEQUENCE
SEQUENCE OF	define uma lista ordenada de zero ou mais elementos de mesmo tipo ASN.1	class SEQUENCE_OF
SET	define uma lista não-ordenada de zero ou mais elementos e cada um é definido por um tipo ASN.1	class SET
SET_OF_	define uma lista não-ordenada de zero ou mais elementos de mesmo tipo ASN.1	class SET_OF

Tabela 3.1 - Mapeamento dos tipos ASN.1 em C++.

Cada tipo de dado **ASN.1** é mapeado internamente para um tipo de dado C++. Isto é feito no construtor da classe quando se declara uma variável ponteiro do tipo interno (**int** , **double** etc) correspondente, como mostra a tabela 3.2, e se aloca espaço de memória para ela.

TIPO DE DADO ASN.1	**FORMA INTERNA C++**
INTEGER	**int ***
REAL	**double ***

_NULL	**NULL**
BOOLEAN	**enum** Boolean(TRUE=1, FALSE =0) *
NUMERICSTRING	**char ***
PRINTABLESTRING	**char ***
OCTECTSTRING	**unsigned char ***
SEQUENCE	**ListASN1 ***
SEQUENCE_OF	**ListASN1 ***
SET	**ListASN1 ***
SET_OF	**ListASN1 ***

Tabela 3.2 - Mapeamento Interno ASN.1 - C ++

O mapeamento da estrutura **ListASN1** para a linguagem de programação C++ é dado por:

```
struct ListASN1{
        ListASN1 *prev;
       ListASN1 *next;
        ListASN1 *filho;
        Attr_Types *elem;
};
```

A estrutura **ListASN1** é composta por quatro apontadores, sendo que os dois primeiros (**prev** e **next**) servem para controlar o seu sequenciamento. O apontador **filho** é utilizado quando da ocorrência de elementos complexos, isto é, ocorrências de novos tipos construtores. O apontador **elem** aponta para classe **Attr_Types**. Isso significa que, o campo **elem** aponta para um dado objeto (instância de um dado tipo) e, através de seus métodos disponíveis, armazena o valor do dado propriamente dito.

Todos os tipos possuem construtores e destrutores implementados.

Nos tipos simples, existem construtores sem valor inicial e com valor inicial. Ambos inicializam as variáveis **n_req_value** e **n_perm_value** com valor 0 (zero). Estas variáveis representam, respectivamente, o número de valores requeridos e número de valores permitidos. Os construtores alocam espaço de memória para a variável **value**, a qual armazena um valor de acordo com o tipo de atributo, através da função **set_value**. O construtor com valor inicial deve existir, pois na definição de uma classe de objeto gerenciado, o usuário pode querer definir um atributo para esta classe com um valor inicial, assim o compilador saberá que quando existir um valor inicial para um atributo, este valor será colocado como parâmetro do construtor para que quando esta classe for instanciada este valor será armazenado.

Nos tipos construtores ASN.1, há somente o construtor com valor inicial. Neste, também serão inicializadas as variáveis **n_req_value** e **n_perm_value**. O usuário irá definir através dos parâmetros do construtor, o tipo e os valores de cada elemento da classe instanciada.

O destrutor acessa o endereço de **value** através da função **get_value** e desaloca o espaço de memória armazenado para esta variável. Assim também são desalocados espaços para as variáveis **_required_values** e **_permitted_values**, se existirem.

Nos tipos construtores, a variável **value** é um ponteiro para o início da estrutura armazenada em memória.. Sendo assim, , o destrutor acessa o endereço de **value** através da função **get_value** e desaloca cada elemento da estrutura armazenada internamente. Assim também são desalocados espaços para as as variáveis **_required_values** e **_permitted_values**, se existirem.

As funções **set_permitted_values** e **set_required_values** armazenam os valores permitidos e requeridos respectivamente. As variáveis **_required_values** e **_permitted_values** são vetores cujas posições são apontadores para os valores requeridos e permitidos, respectivamente. Este conjunto de valores requeridos e permitidos é fornecido pelo usuário quando da definição de um atributo. Um valor requerido deve ser sempre um valor permitido. No caso dos tipos construtores, as variáveis

_required_values e **_permitted_values** são vetores para listas, onde cada posição do vetor irá apontar para uma lista de valores requeridos e permitidos, respectivamente.

As funções **IsPermitted** e **IsRequired**, retornam um valor *booleano*. Elas verificam se determinado valor pertence ao conjunto de valores permitidos ou requeridos, respectivamente, retornando TRUE se pertencer e FALSE caso contrário. No caso dos tipos construtores, estas funções verificam se uma determinada lista (de mesmo tipo da armazenada internamente) pertence ao conjunto de listas permitidas ou requeridas, respectivamente.

As possíveis operações e regras de comparação que podem ser realizadas sobre um tipo de atributo ASN.1 estão descritas na tabela 3.3. Isso significa que o usuário, quando definir um atributo, deverá determinar quais operações e regras de comparação poderão ser realizadas sobre o atributo, não podendo estar fora do conjunto definido estaticamente para cada tipo.

TIPO ASN.1	OPERAÇÃO	REGRAS DE COMPARAÇÃO
INTEGER	get, replace e replace-with-default	Equality,GreaterOrEqual e LessOrEqual
REAL	get, replace e replace-with-default	Equality,GreaterOrEqual e LessOrEqual
NULL	get, replace e replace-with-default	Equality
BOOLEAN	get, replace e replace-with-default	Equality
NUMERICSTRING	get, replace e replace-with-default	Equality,GreaterOrEqual , LessOrEqual e Substring (Initial, Any e Final)
OCTECTSTRING	get, replace e replace-with-default	Equality,GreaterOrEqual, LessOrEqual e Substring (Initial, Any e Final)
PRINTABLESTRING	get, replace e replace-with-default	Equality,GreaterOrEqual, LessOrEqual e Substring (Initial, Any e Final)
SEQUENCE	get, replace e replace-with-default	Equality
SEQUENCE OF	get, replace e replace-with-default	Equality
SET	get, replace e replace-with-default	Equality
SET OF	get,replace,replace-with-default, add,remove	Equality

Tabela 3.3 - Regras e Operações dos tipos de atributos ASN.1.

A operação Get (CMIP) é mapeada para a função get em C++. Esta operação utiliza a função get_value para obter o endereço de value . A operação get possui um parâmetro, o qual é um ponteiro de ponteiro para void. Este parâmetro irá retornar o valor armazenado em value .

A operação **Replace (Set - CMIP)** é mapeada para a função **replace** em C++. Esta operação utiliza a função **get_value** para obter o endereço de **value**. A variável **value** irá receber o conteúdo do parâmetro da operação **replace**.

No caso dos tipos construtores, as operações **get** e **replace** também são implementadas da mesma forma. A diferença é que, nestes tipos, são chamados procedimentos que tratam da cópia de elemento por elemento da estrutura armazenada internamente, (na função **get**), ou da variável passada no parâmetro (no caso da função replace).

Além destas, são definidas as operações **Add** e **Remove** para a classe SET_OF.

A operação **Add**, mapeada para a função **add** em C++, consiste em adicionar um elemento de tipo ASN.1 na última posição da estrutura armazenada internamente. Esta operação tem como parâmetro o dado a ser inserido. Utiliza a função **get_value** para obter o endereço da estrutura armazenada e a percorre até chegar à última posição. Então a última posição apontará para o elemento a ser inserido.

A operação **Remove**, mapeada para a função **remove** em C++, retira um determinado elemento da estrutura apontada por **value**. Utiliza a função **get_value** para obter o endereço da estrutura armazenada e retira desta estrutura o elemento indicado no parâmetro.

Não existe a função **replace-with-default** implementada pelo tipo de dado ASN.1, pois esta operação é implementada na definição do atributo que chama a função **replace** com o valor default.

3.4. *Template* da Ação

O *template* **ACTION** é usado para definir o comportamento e sintaxe associada com o tipo de ação particular. Tipos de ações definidos por meio deste *template* pode ser carregado pelo serviço M-ACTION [5]. O *template* da ação é mostrado na figura 3.5.

```
<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 3.5 *Template* de ACTION

Ações assim como atributos, também são mapeadas para classes (**class**) em C++. Existe uma classe base de onde derivam todas as ações definidas pelo usuário.

A classe ACTION não é instanciável, portanto é uma classe abstrata. Nesta classe, também existem as funções **_build_if**, **_build_cs**, **_copyFrom_cs** e **_copyTo_cs**.

De acordo com o *template* da ação, se MODE CONFIRMED estiver presente, a ação operará no modo confirmado. Se estiver ausente, a ação pode ser confirmada ou não confirmada, ficando esta decisão a cargo do gerente. Assim, a variável **Mode** do tipo booleano receberá o valor de **mode** que vem como parâmetro do construtor ACTION. Isso significa que, toda ação definida pelo usuário terá um construtor onde o parâmetro é o **mode** com valor default FALSE, e que chamará o construtor ACTION com este parâmetro. Então se **Mode** receber o valor TRUE a ação deve ser confirmada, se receber FALSE a ação pode ser confirmada ou não confirmada.

3.5. *Template* de Notificação

- template de Notificação é usado para definir o comportamento e sintaxe associados com notificações de um objeto. Os tipos de notificações definidos por meio deste *template* podem ser carregados em relatórios de eventos pelo serviço M-EVENT-REPORT [5]. Na figura 3.6 é apresentado o *template* da notificação.

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

Figura 3.6 Template de Notificação

As notificações também são mapeadas para classes (**class**) em C++. Existe uma classe base chamada **NOTIFICATION**, da qual derivam todas as notificações definidas do usuário. É uma classe abstrata, portanto não instanciável.

De acordo com o *template* da notificação, se **WITH INFORMATION SYNTAX** estiver presente, este construtor identifica o tipo de dado ASN.1 que descreve a estrutura da informação de notificação que é carregada no protocolo de gerenciamento, e permite a associação de identificadores de atributo com campos nomeados na sintaxe abstrata. Se ausente, não existe informação específica de notificação associada com a invocação da notificação. Se a opção **AND ATTRIBUTES IDS** estiver presente, o field-name será um identificador definido dentro da sintaxe abstrata referenciada pelo type-reference que aparece na construção. O tipo de dado que é identificado pelo field-name é usado para carregar valores do atributo referenciado pelo attribute-label. O tipo de dado ASN.1 do atributo será o mesmo do tipo de dado referenciado por field-name.

Assim, foi definida a estrutura **Attr_IDS** composta pelos campos **field-id** e **AID**, representando o field-name e o attribute-label, respectivamente. Ou seja, quando existir esta estrutura, a cada campo da estrutura da notificação estará associado um atributo, que conterá o valor do deste atributo.

São declaradas duas variáveis: um ponteiro **Attr_ids** para a estrutura **Attr_IDS** e **n_element** do tipo inteiro, que determina quantas estruturas deste tipo irão existir, relativas ao número de atributos da estrutura que serão associados a atributos este objeto.

Existem dois tipos de construtores para a classe **NOTIFICATION**. Um com o parâmetro Attr_ids recebendo o valor **NULL**, significando que a notificação não usa os atributos para preencher os campos da estrutura **Attr_IDS**. O outro construtor possui dois parâmetros, um que é o ponteiro para **Attr_IDS** e o outro n do tipo inteiro, definindo quantas estruturas do tipo **Attr_IDS** deverão ser instanciadas, quando da instânciação de uma notificação.

O destrutor da classe liberará espaço de memória ocupada pelo ponteiro Attr_ids, se exixtir, ou seja, se foi alocado espaço para alguma estrutura **Attr_IDS**.

A função virtual SendNotification será implementada pelo usuário e possui como parâmetros a sintaxe de informação enviada pelo objeto e a sintaxe de resposta. Esta sintaxe de resposta, determinada por **WITH REPLY SYNTAX** do *template* da notificação, é usada onde a notificação é enviada, usando o modo confirmado do serviço **M-EVENT-REPORT** do **CMISE**. Confirmações de eventos não retornam para o objeto para o objeto gerenciado. A decisão de enviar uma notificação no modo confirmado ou não, é problema que o agente dever determinar, baseado nas políticas nas políticas associadas com o gerente. Quando a construção **WITH REPLY SYNTAX** é omitida da definição da notificação, a notificação é enviada no modo confirmado. Neste caso, a confirmação não

incluirá informação de resposta. Tudo isso é descrito no comportamento da notificação, representada pela construção **BEHAVIOUR** deste *template*.

3.6. *Template* de Comportamento

O *template* BEHAVIOUR é usado para definir aspectos comportamentais de classes de objetos gerenciados, *name bindings*, parâmetros e atributos, tipos de ações e notificações. Este *template* é entendido permitir previsões de extensão, mas especificações de comportamento não mudarão a semântica da informação definida previamente. Se a informação não é definida, a definição de comportamento será explícito sobre o que não é definido. Segue abaixo o template do comportamento:

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

Figura 3.7 *Template* do BEHAVIOUR

Como o comportamento é apenas uma descrição textual, mas muito importante para a definição dos componentes da classe de objeto gerenciado, que aparece nos templates de pacotes, atributos, ações, notificações já vistos anteriormente, e também naqueles não implementados (parâmetros, name bindings), ele é mapeado na forma de texto em C++.

4. Conclusão

O projeto global de um Sistema de Gerência de Redes, abrangendo todas as áreas funcionais definidas no modelo de gerência OSI / ISO, está sendo desenvolvido por diversos componentes do Grupo de Pesquisa em Redes de Computadores do LISHA.

As normas da ISO, para Gerência de Redes em Sistemas Abertos apresentam um enorme obstáculo quando tenta-se aplicá-las a um ambiente real.

Este trabalho assume um papel importante ao tentar reduzir a distância existente entre as interpretações dos padrões internacionais para uma implementação real. Baseado principalmente na norma que especifica as Regras para a Definição de Objetos Gerenciados (GDMO), o trabalho fornece suporte para que possam ser construídas aplicações de gerência. Conceitos e regras que compõem a definição de uma classe de objeto gerenciado foram implementados na linguagem C++.

5. BIBLIOGRAFIA

[1].ISO/IEC 10165-4 I CCITT Rec. X.722, Information Technology - Open Systems Interconnection - Structure of Management Information - Part 4: *Guidelines for the Definition of Managed Objects*, International Organization fo Standardization/Internacional Eletrotechnical Commission, março de 1991.

[2].Booch, G. "*Object - Oriented Analysis and Design with Applications - Second Edition*", The Benjamin / Cummings Publishing Company, 1994.

[3].CCITT Rec. X.680, Information Technology - Open Systems Interconnection - *Abstract Syntax Notation One* (ASN.1): *Specification of Basic Notation*, março de 1993.

[4].ISO/IEC 10165-1 | CCITT Rec. X.720, Information Technology - Open Systems Interconnection - Management Information Services - Structure of Management Information - Part 1: *Management Information Model*, International Organization for Standarization/International Electrotechical Commission, janeiro 1992.

[5].ISO/IEC 9595 | CCITT Rec. X.710 - Information Technology - Open Systems Interconnection - Common Management Information Service Definition , março de 91.