マルチメディア通信と分散処理　55－2
（1992・5．15）

OSI応用層プロトコル用ASN.1ライトウェイト符号化規則のための符号化/復号処理系の実装と評価

堀内浩規　小花貞夫　鈴木健二
国際電信電話株式会社研究所

OSI(開放型システム間相互接続)の応用層におけるデータ要素は、ASN.1(抽象構文記法1)により定義され、このデータ要素を符号化するための符号化規則には、国際標準として基本符号化規則(BER : Basic Encoding Rules)がある。しかし、次世代LANや広帯域ISDN(ATM等)などの高速通信環境では、エンドシステムの処理がボトルネックとなり、特に、ASN.1の符号化/復号処理の高速処理が必要となる。このため、ASN.1の新符号化規則の一つとして、計算機の内部表現に基づき高速な符号化/復号を可能とするライトウェイト符号化規則(LWER : Light Weight Encoding Rules)が注目されている。本稿では、LWERを扱うOSI応用層プログラムを容易に開発可能とするため、抽象構文からLWER用符号化/復号プログラムを自動生成するLWER符号化/復号処理系の実装とその評価結果について報告した。具体的には、抽象構文定義からプログラミング言語(C言語)用型定義ならびに符号化/復号関数を自動生成するための生成規則を明確にし、特に、符号化/復号関数では、従来のBERの場合とは異なり、抽象構文定義に対応するC言語の型を持つ変数値を符号化結果格納領域に直接書き込むことで実現できることを示した。また、実装したLWER符号化/復号処理系を、生成された符号化/復号プログラムの符号化/復号処理時間等の観点より評価した。この結果、符号化/復号処理時間については、特に、応用層プロトコルでよく使用されるINTEGERを複数持つデータ構造の場合にはBERの約20倍以上高速となり実装したLWER処理系が実用できることを示した。さらに、LWERは、符号化データ長がBERに比較して長くなり伝送時間が増加する傾向にあるため、特に、伝送時間よりエンドシステムにおける処理時間が全体の通信時間に影響を及ぼす高速通信環境で有効となることを示した。

Implementation and Evaluation of Automatic Generator of Encoder/Decoder programs for ASN.1 Light Weight Encoding Rules

Hiroki Horiuchi　Sadao Obana　Kenji Suzuki
KDD R&D Labs.
2-1-15, Ohara, Kamifukuoka-shi, Saitama, 356 Japan

Abstract syntaxes of data elements in OSI application layer protocols are defined by using Abstract Syntax Notation One (ASN.1), and for the encoding of these data elements in transfer syntaxes, Basic Encoding Rules (BER) has been standardized as an international standard. BER has been widely applied to various application layer protocols. However, it becomes necessary to realize high-speed ASN.1 encode/decode to avoid the bottleneck at end systems under high-speed communication network environments such as high-speed LAN and B-ISDN(ATM etc.). For the purpose of enabling the high-speed ASN.1 encoding/decoding, standardization of Light Weight Encoding Rules (LWER) is under way as one of the new encoding rules, which is based on the internal data representation in computers. It is expected that LWER will be widely utilized in OSI systems under high-speed communication network environment. In order to implement various OSI application layer protocol programs which handle LWER effectively, we have developed the automatic generator (or compiler) of LWER encoder/decoder programs from abstract syntax definitions by ASN.1. In this paper, we clarified the mechanisms for generation of encoding/decoding programs of LWER and ensured the effectiveness of the compiler through the evaluation of the encoding/decoding time and generated program size. Especially, the encoding/decoding time for the typical data structure was 20 times faster than that in BER.

1. はじめに

OSI(開放型システム間相互接続)の応用層におけるデータ要素は、ASN.1(抽象構文記法1)[1]により定義されるが、このデータ要素を符号化するための符号化規則には、国際標準として広く普及している基本符号化規則[2](BER : Basic Encoding Rules) がある。近年OSIの利用技術が進み、業界ネットワークやLANの構築でOSIが導入され、通信処理の高速化への要求が高まってきた。これに対する解決方法の一つとして、ASN.1符号化規則の改良が注目されている。ISO等で現在検討している圧縮符号化規則(PER : Packed Encoding Rules)[3]、識別符号化規則(DER : Distinguished Encoding Rules)[4]並びにライトウェイト符号化規則(LWER : Light Weight Encoding Rules)[5]はASN.1符号化規則の改良の一例である。ここで、PERおよびDERはBERをベースとしており、PERはBERの符号化データ長を短くして転送時間の短縮、DERは値に対する符号化結果を一意として処理負荷の軽減を計るものである。これに対して、LWERは計算機の内部表現に基づいた符号化方式により大幅な符号化/復号処理時間の短縮を目指し、PERやDERよりも高速処理に向くことが報告されている[6,7,8,9]。

OSI通信の高速化がはかられるにつれ、今後は、エンドシステムの処理がボトルネックとなることを避けるため、ASN.1符号化規則として、LWERを使用する機会が増えることが予想される。このため、LWERを扱うOSI応用層プログラムを容易に開発可能とする必要が生じ、抽象構文からLWER用符号化/復号プログラムを自動生成するLWER符号化/復号処理系(LWER処理系)の開発等が重要となる。これまでにも、BERを対象とした符号化/復号処理系に関する各種研究報告がなされている[10,11,12]が、LWERを対象とした処理系は、従来の処理系に比べて、計算機の内部表現を符号化に利用するため符号化方法が異なるとともに、高速処理に特徴を持つ違いがある。そこで、筆者等はLWER符号化/復号処理系を実装して、性能評価を行ったので、以下に、概要と評価結果を報告する。

2. BERをベースとした符号化規則とLWER

2.1 BERをベースとした符号化規則(PER,DER)の概要

基本符号化規則(BER)では、ASN.1により定義された各データ要素の型を、図1に示すように、識別子(ID)、長さ(LI)、コンテンツ(CO)の3つの要素からなるオクテット列として符号化することとしている(但し、LIが不定長の場合には、さらにコンテンツ終了(EOC)が付加)。

識別子 (ID)	長さ (LI)	コンテンツ (CO)

a)LIが 固定長の場合

図1 基本符号化規則による符号化データの構成

IDは型のクラスと番号等を示す。LIはデータの長さを示す。COは、データ要素の値自身または、他のデータ要素が入れ子として含む構造形からなる。圧縮符号化規則(PER)は、BERのうち予測可能なIDやLIを削除を行うことにより、符号化データ長を圧縮する符号化規則である。識別符号化規則(DER)は、BERにおける符号化方法に制約を加えて、値に対する符号化結果を一意のオクテット列とする符号化規則である。BER, PER, DERは全てオクテット単位の符号化規則である。

2.2 LWERの概要

ライトウェイト符号化規則(LWER)は、計算機の内部表現を意識し、負荷を軽くして高速な符号化/復号を可能とするワード単位の符号化規則(表1)であり、以下の特徴を持つ。

表1 主なライトウェイト符号化規則

型名	符号化方法
BOOLEAN	1ワードで符号化. 値はFALSE(0) / TRUE(0以外).
INTEGER / ENUMERATED	1ワードで符号化. 値はワードサイズにより制限される.
REAL	64bitで符号化. IEEE標準フォーマット.
BitString /CharacterString / OctetString	長さ / ポインタ/値の3組で表現. 値はワードでアラインメントが行われる. 値がない時、長さ / ポインタの値0.
NULL	符号化されない. 但し, SETおよびSEQUENCEの要素でOPTIONALの時ポインタ(存在1, 存在せず0)
SEQUENCE / SET	SEQUENCEとSETの区別無し. 抽象構文の定義順に要素を並べる. OPTIONAL,DEFAULTの要素はポインタ/値で表現. 但し、Extensible(拡張可能)の場合には先頭に固定部分の長さを付加.
SEQUENCE OF / SET OF	要素数を示すカウンタ / ポインタ / 値列の3組で表現.
CHOICE	選択要素の識別のためのインデックス / 選択された型の符号化 の2組で表現. 選択された型の符号化は、最も長い型にそろえてパディング.
OBJECT IDENTIFIER	OctetStringとして符号化.
Tagged	符号化されない.
ANY	長さ / ポインタ/値の3組で表現.

(1) 符号化形式

① 計算機の内部表現を考慮してワードを符号化の単位とする。

② 型識別のための情報は付加しないため、INTEGER型とENUMERATED型、IA5StringやPrintableString等のCharacterString間で符号化上の差は無い。

③ 固定長の型(BOOLEAN/INTEGER /ENUMERTAED/REAL)に対しては値の長さは付加されない。

④ String系の型等の可変長の値を持つ型は、値の長さ / ポインタ(以下ではオフセットと呼ぶ。) / 値の3組により符号化を行う。なお、オフセットは、オフセットが格納されている位置と値の位

置との相対位置を示す。図2に示すように、符号化位置には値の長さとオフセットが配置され、値は符号化データの後部にまとめて配置する。

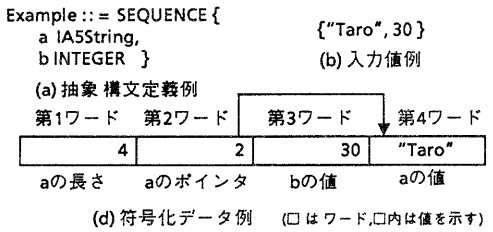
(2) 扱うデータ構造における制約

① 型SEQUENCEとSETの区別を無くし、要素の符号化順は抽象構文定義の順とする。

② INTEGERの値の範囲、CharacterStringの長さ、SET OF / SEQUENCE OFの要素数等は、1ワードで表現できる範囲に限定される。

(3) 転送構文

他の符号化規則が単一の転送構文を持つのに対して、LWERの転送構文はワード長とワードを構成するバイトの順序を組として複数定義される。現在、ワード長が16 / 32 / 64ビットの3種、バイトの順序がBig EndianかLittle Endianかの2種の組合せ計6組の転送構文が定義されている。



注) SEQUENCEの符号化はExtensibleを使用していない。

図2ライトウエイト符号化規則の符号化例

3. LWER符号化/復号処理系の実装

3.1 設計方針

LWER符号化/復号処理系(LWER処理系)を、以下の設計方針により、SUNおよびVAX(OSはVMS)上に実装した。

① ASN.1による抽象構文の定義からC言語による符号化/復号処理プログラムを自動生成する。

② 生成されるC言語の型定義は、LWER処理系の動作する計算機の転送構文に従う。

③ BERと異なり、LWERは計算機のワード長とバイト順序により6種類の転送構文が定義されているため、全ての転送構文に対応する符号化/復号プログラムを生成する。

④ 従来のBER処理系の符号化/復号ではLIの計算、値の符号化形式への変換やIDの付加等が必要であるのに対し、本処理系の符号化/復号は、高速処理のため値の設定された変数を直接符号化結果格納領域(以下ではバッファと呼ぶ)に書き込むことで実現する。

3.2 ソフトウェア構成

実装したLWER処理系のソフトウェア構成を以下に示す(図3)。

(1) プログラム生成部

抽象構文解析部、C言語の型定義生成部、符号化/復号関数生成部からなる。

① 抽象構文解析部

C言語の型定義生成部および符号化/復号関数生成部の処理を容易とするため、抽象構文定義中のASN.1の型定義の構文解析を行い、内部構造に変換する。この内部構造はASN.1の型定義の構造を保った形でメモリ上に生成する。

② C言語の型定義生成部

内部構造から、LWER処理系の動作する計算機と同じワード長とバイトの順序となる転送構文(以下自システムの転送構文と呼ぶ)に対応するC言語の型定義を、3.3節の生成方法に従って生成する。

③ 符号化/復号関数生成部

内部構造から、抽象構文定義中で定義された型定義毎に符号化/復号を行う関数群を3.4節で述べるLWER符号化/復号関数の生成方法に従って生成する。生成する符号化/復号関数は自システムと異なるワード長やバイト順序を持つ計算機との通信も可能とするため、自システムの転送構文に加え、自システムと異なる転送構文(計6種類)にも対応した。また、符号化/復号処理の高速化と転送構文毎の切りだしを容易とするため、転送構文毎に別プログラムを生成する。

(2) 共通関数部

抽象構文の種類に依存せず、生成した符号化/復号関数によって共通的に使用される関数(表2)をあらかじめ用意した。

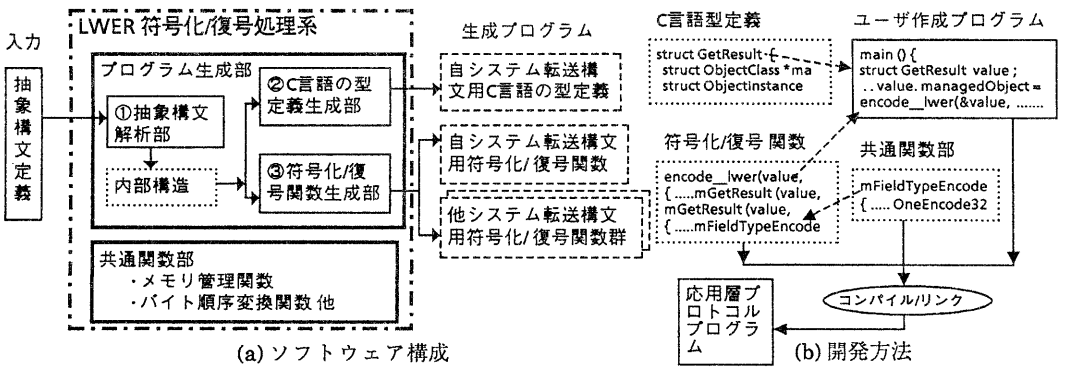


図3 LWER符号化/復号処理系のソフトウェア構成とLWER処理系を用いた応用層プロトコルプログラムの開発方法

表2 共通関数部に含まれる関数例

関数	機能
バイト順序変換	BigEndianとLittleEndian間の変換。
ワードサイズ変換	各ワード長(16/32/64)間の変換。
型SEQUENCE OF/SET OF符号化/復号	要素数を示すカウンタとオフセットの符号化/復号。
String系の型の符号化/復号	CharacterString,OctetString,BitString型の符号化/復号。
メモリ管理	メモリの割り付け/解放を管理する。

LWER処理系を用いた応用層プロトコルプログラムの開発方法は以下となる(図3(b))。

① LWER処理系に、対象となる応用層プロトコルの抽象構文を入力して、C言語の型定義と符号化/復号処理関数を生成させる。

② ユーザ作成プログラムでは、データ要素の符号化は、生成されたC言語の型定義を持つ変数に値を設定し、生成された符号化関数を呼び出す。また、復号は受信データを設定して、生成された復号関数を呼び出す。ユーザ作成プログラムが呼び出す符号化/復号関数の引数は、生成された型定義を持つ変数、転送構文の種別、符号化対象となる抽象構文定義内の型名、符号化/復号データ、エラー情報からなる。

③ ②で作成したユーザプログラムとともに、符号化/復号関数、共通関数部をコンパイル/リンクを行い、応用層プロトコルプログラムを作成する。

3.3 C言語型定義の生成

図3(a)におけるプログラム生成部では、抽象構文中のASN.1の型定義からC言語の型定義の生成は表3に示す生成規則を適用する。例えば、BOOLEAN等

表3 C言語型定義の生成規則

ASN.1の型	生成されるC言語の型
BOOLEAN/INTEGER/ENUMERATED	int
REAL	double
BitString/OctetString/CharacterString/Object Identifier	値の長さ(int length)と値を格納するポインタ(char *field)からなる構造体(struct field)。
NULL	SEQUENCEまたはSETの要素で、存在がオプショナルの時はintを生成。 存在が必須の時は生成させない。
SEQUENCE SET	対応する要素を持つ構造体(struct)で、オプショナルな要素は型定義へのポインタとする。構造体名は型定義名、構造体のメンバー名はASN.1のレファレンスを使用。(拡張可能な場合は、先頭に長さ(int length)が付加される。)
SEQUENCE OF SET OF	要素数を示すカウンタ(int number)と要素の型へのポインタを持つ構造体(struct)。
CHOICE	選択要素を識別するためのインデックス(int index)と対応する要素を持つ共用体(union)。

はint、可変長の値を持つBitString等は値の長さと値を格納するポインタをメンバに持つ構造体、SEQUENCEは対応する要素を持つ構造体を生成する。入力される抽象構文定義例と生成されるC言語の型定義例を図4に示す。各ASN.1型定義に対応する

```
Record ::= SEQUENCE {
        number      INTEGER,
        name        Name,
        dateOfHire      IA5String,
        children    ChildInf };
ChildInf ::= SEQUENCE OF Name ;
Name ::= SET {  first    [1] IA5String,
                last     [2] IA5String  };
```

(a) 入力抽象構文定義例

```
struct field {  int     length;
                char    *field;};
struct Name {struct field first;
             struct field last;};
struct ChildInf {   int num;
                    struct Name *child ;  };
struct  Record {    int number;
                    struct Name    name;
                    struct field   dateOfHire;
                    struct ChildInf children;};
```

(b) 生成されるC言語の型定義例

図4 抽象構文に対応するC言語の型定義生成例

構造体struct Record、struct ChildInfおよびstruct Name、可変長の値を格納するための構造体struct fieldが生成されている。構造体のメンバー名はASN.1のレファレンス(number, name,first, last等)を使用している。なお、SEQUENCEやSETの型における要素の追加が可能か否かは(拡張可能性)は、LWER処理系起動時に指定することとしている。

3.4 符号化/復号関数の生成

符号化(復号)関数の処理は、符号化(復号)を行うため、3.3節で示したC言語の型を持つ変数をバッファに(から)直接書き込む(読み込む)ことを基本とする。自システムと異なる転送構文の符号化/復号関数は、計算機の内部表現が異なるため、さらにワード長やバイト順序の変換が必要となる。以下に自システムの転送構文の場合と異なる場合の符号化/復号関数の生成規則を述べる。

3.4.1 自システムと同一の転送構文の場合

(1) 符号化関数の生成

① C言語の型定義を持つ変数をたどり符号化に必要なワード長を計算する。CharacterString等の場合には、値をワード単位で計算する。

② ①のデータ長に応じた領域を確保する。

③ 表4に示す符号化関数の基本生成規則に従い、抽象構文の型定義毎に符号化/復号関数を生成する。例えば、INTEGER等の固定長の型に対応する要素は、単純に変数をバッファにコピーする。OctetString等の可変長の型に対応する要素(図4(b)のstruct fieldの型を持つ要素)は、値の長さ(int length)は単純にコピーを行い、値については、ポインタ値の代わりにオフセットを書き込み、値自体はバッファの後部に書き込む。構造形のSEQUENCEおよびSETは、各要素の型に応じた符号化を、定義順に従ってバッファに書き込む。この時、オプショナルな要素の場合は、ポインタの値が0の時は単純に

表4 符号化関数の基本生成規則

ASN.1の型	生成する符号化関数の処理
BOOLEAN / INTEGER / ENUMERATED / REAL	値の設定された変数を関数memcpy()を用い、バッファにコピー。
BitString / OctetString / CharacterString /Object Identifier	①値の長さを示すメンバ(int length)は関数memcpy()でコピーを行う。 ②ポインタ値(char *field)の代わりにオフセットを書き込む。 ③値はバッファの後部にコピー。
NULL	変数が在る場合にはコピー。
SEQUENCE SET	要素の型に応じ、定義順に符号化する。オプショナルな要素は、ポインタ値0の時はコピー、非0の場合はオフセット値を書き込み、値を後部に書き込む。ポインタを含まない要素が連続する時は、複数まとめてコピーする。
SEQUENCE OF SET OF	①要素数を示すカウンタ(int number)は単純にコピーを行う。 ②ポインタの代わりにオフセットの値を書き込む。 ③バッファ後部にカウンタの数だけ、要素の型に従いコピーする。。
CHOICE	①選択要素を識別するインデックス(int index)および共用体をコピー。 ②選択要素にポインタが有る時は、バッファ内のポインタ値をオフセットに書き換え、バッファ後部に値を書き込む。

注) 復号関数の生成は、基本的には符号化関数の生成規則の逆の規則となる。

コピーを行い、0でない時はオフセットの値を書き込んだ後、相当する値を後部に書き込む。

図4(a)の抽象構文の入力に対して生成される符号化関数(32ビット/BigEndian)の例を図5に示す。図4(a)の抽象構文定義中の型Record、ChildInf、Nameに対応する符号化関数mERecord32No()、mEChildInf32No()、mEName32No()がそれぞれ生成されている。また、関数mFieldTypeEncode32No(), OneEncode32No()は、共通関数部内の関数であり、それぞれ、String系の符号化、SEQUENCE OFの要素数を示すカウンタの符号化に使用される。

(2) 復号関数の生成

復号関数の生成については、基本的には表4で示した符号化関数の基本生成規則の逆の規則を適用する。つまり、変数からバッファへのコピーをバッファから変数へのコピーに変更する。また、オフセットを使用する場合には、オフセットの値をポインタとして変更してからコピーする。

3.4.2 自システムと異なる転送構文の場合

自システムと異なる転送構文の場合における符号化関数の生成は、3.4.1節で述べた自システムと同一の転送構文の場合における符号化関数の基本生成規則(表4)に加えて、以下の規則を追加する。また、復号関数の生成についても同様な変換規則が追加される。

```
/** 型Recordの符号化関数 (32Bit /BigEndian) **/
mERecord32No(Inp, str1, str2)
    struct Record   *Inp;    /* 入力変数 */
    unsigned char  **str1;  /* 符号化結果の格納位置*/
    unsigned char  **str2;  /* 符号化後部の位置*/
{
    int i;  int leng;  int size;   unsigned char  *ptr;
    int     cnt;    unsigned char  *str3;

    memcpy(*str1,&(Inp->number),4); /*number符号化*/
    (*str1) + = 4;
    if(mEName32No(&(Inp->name), str1, str2) = = FALSE)
        return FALSE;  /* nameの符号化 */
    if(mFieldTypeEncode32No(&(Inp->dateOfHire), str1,
str2, FALSE) = = FALSE)  /* dateOfHire の符号化 **/
        return FALSE;
    if(mEChildInf32No(&(Inp->children), str1, str2) = =
FALSE)      return FALSE;  /* childrenの符号化 */
            return TRUE;
}
/* 型ChildInfの符号化関数 (32Bit /BigEndian) */
mEChildInf32No(Inp, str1, str2)
struct ChildInf  *Inp;    /* 入力変数 */
unsigned char  **str1;  /* 符号化結果の格納位置*/
unsigned char  **str2;  /* 符号化後部の位置*/
{   int     i;  int     leng;  int     size;
    unsigned char  *ptr;  int  cnt;
    unsigned char  *str3;
    OneEncode32No((&(Inp->number)), (*str1));
    if(Inp->number)          /* カウンタの符号化 */
        leng = IsDiffLength((*str1), (*str2), 4);
    else
        leng = 0;
    OneEncode32No((&leng), (*str1));
    str3  = *str2;
    *str2 + = (sizeof(struct Name) * Inp->number);
    ptr = (unsigned char *)Inp->name;
    for(i = size = 0; i < Inp->number; i + +, size + =
sizeof(struct Name)){
        if(mEName32No(&ptr[size], &str3, str2) = = FALSE)
            return FALSE; }  /* Nameの符号化 */
            return TRUE;
 }
/* 型Nameの符号化関数 (32Bit / BigEndian) */
mEName32No(Inp, str1, str2)
    struct Name    *Inp;    /* 入力変数 */
    unsigned char  **str1;  /* 符号化結果の格納位置*/
    unsigned char  **str2;  /* 符号化後部の位置*/
{
    if(mFieldTypeEncode32No(&(Inp->first), str1, str2,
FALSE) = = FALSE)  /* firstの符号化 */
        return FALSE;
    if(mFieldTypeEncode32No(&(Inp->last), str1, str2,
FALSE) = = FALSE)      return FALSE; /* lastの符号化 */
        return TRUE;
}
```

図5 生成された符号化関数例

追加規則

(1)ワード長が異なる転送構文の場合

intの型を持つ要素に対し、ワード長減少時には値があふれないことを検査した後必要なバイトのみを、ワード長増大時には必要なバイトを付加して

バッファへ書き込む。また、OctetString等の値に対しては、アライメントをワード長に合わせる。

(2)バイト順序が異なる転送構文の場合

C言語でバッファ内に書き込む際にバイト順序変換の規則新たに追加する。

4. LWER符号化/復号処理系の評価

4.1 生成される符号化/復号プログラムの処理時間の測定

以下では、LWER処理系が生成する符号化/復号プログラムの処理時間の評価を行うため、典型的なデータ構造と実応用層プロトコルを用いた測定を行った。なお、測定にはSUN 3/260(ワード長32ビット/バイト順序BigEndian)を使用し、符号化/復号処理時間は1000回ループさせた平均をとった。

4.1.1 典型的なデータ構造を用いた場合の測定結果

典型的なデータ構造として、1)符号化が構造体のコピーだけで済む固定長の要素のみの場合、2)可変長の要素を含み、符号化が構造体のコピーに加えて、オフセットの計算と後部への値のコピーが必要となる場合について、6種類の転送構文に対し符号化/復号処理時間測定した。比較のため、文献[10]の処理系を用いたBERの符号化/復号処理時間も測定した。また、BER, PER, LWERの符号化データ長の結果も併記した。

(1) 固定長の要素のみの場合

図6(a)に示すSEQUENCEでINTEGERの要素を複数持つ抽象構文定義(対応するC言語型定義を図6(b)に示す)に対して要素数を変化させた。

図6(c)(d)より測定を行ったSUN 3/260と同一のワード長/バイト順序の転送構文を使用した場合の符号化/復号処理時間は、バイト順序変換とワードサイズの変換が不要であることと、複数の要素をまとめてバッファに書き込む方法によりBERより20倍以上高速となった。また、異なる転送構文を使用した転送構文では、バイト順序変換がワードサイズ変換より処理負荷が大きいため、16ビット/BigEndianが最も高速となり、64ビット/LittleEndianの場合はバイト順序変換とワード長増大によるデータ長の増加により遅くなる。しかしながら、異なる転送構文を使用した場合でもBERよりは高速となる。

(2) 可変長の要素を含む場合

図7(a)に示す可変長のIA5Stringを含む要素の並びとなる抽象構文定義(対応するC言語型定義を図7(b)に示す)に対して要素数を変化させた。

図7(c)(d)よりLWERの符号化/復号時間は、自システムの転送構文の場合には、それぞれBERの約3.5倍/約6.0倍高速となった。また、符号化データ長はBERの約1.8倍、PERの約2.5倍と大きくなった。

4.1.2 実際の応用層プロトコルのデータ要素を使用した場合の測定結果

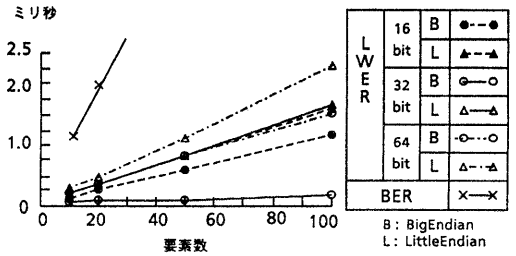
実際のOSI応用層プロトコルでよく使用されるデータ要素の構造は、以下の3種類に大別できる(表5)。

```
ClassA ::= SEQUENCE {
  elm1   INTEGER,
  elm2   INTEGER,
  elm3   INTEGER,
    .
    .
  elmn   INTEGER
}
```

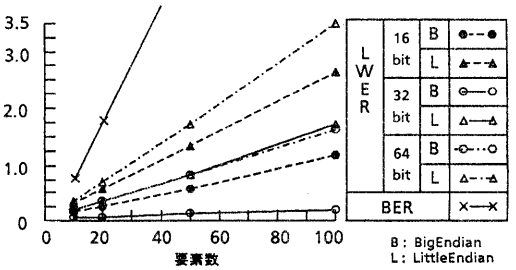
(a) 抽象構文定義

```
struct ClassA {
    int   elm1;
    int   elm2;
    int   elm3;
      .
      .
    elmn ; };
```

(b)対応するC言語の構造体



(c) 符号化処理時間



(d) 復号処理時間

符号化規則 \ 要素数		10	20	50	100
LWER	16bit/word	20	40	100	200
	32bit/word	40	80	200	400
	64bit/word	80	160	400	800
BER		32	62	153	304
PER		11	21	51	101

(e) 符号化データ長

図6 符号化/復号処理時間

[構造a] 一般的に構造形の要素は少なく、しかも基本形の長さが小さい場合。(ディレクトリ、OSI管理、RDAプロトコル等の操作要求を行うデータ要素を想定)

[構造b] 一般に基本形の長さは小さいが、構造形の要素が多い場合。(ディレクトリ、OSI管理、RDAプロトコル等の操作応答が数10個以上の場合やFTAMにおけるシーケンシャルフラットファイルの転送のデータ要素を想定)

[構造c] 特定の基本形の長さの大きな値を持つ場合。(MHSで運ばれるテキスト電文やFTAMにおける無構造ファイルの転送のデータ要素を想定)

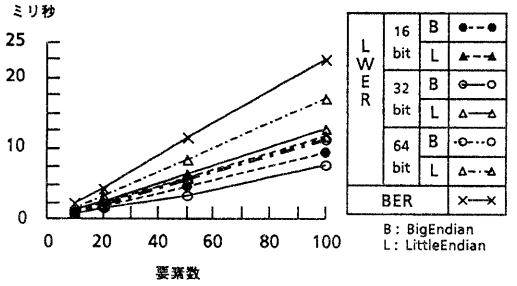
上記の構造の分類に従って、実際の応用層プロトコルのデータ要素における符号化/復号処理時間とデータ長について、LWER処理系により生成される

```
Group :: = SEQUENCE OF
        Person
Person :: = SEQUENCE {
  number   INTEGER,
  nameIA5String }
```

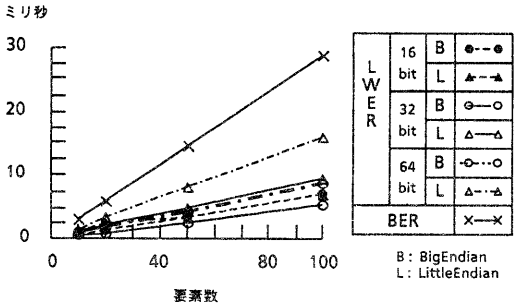
(a) 抽象構文

```
struct Person { int number ;
   struct { int length ;
      char *field; } name ;};
struct Group { int count ;
   struct Person *person ;};
```

(b) 対応するC言語の構造体



(c) 符号化処理時間



(d) 復号処理時間

符号化規則 ＼ 要素数		10	20	50	100
LWER	16bit/ word	124	244	604	1204
	32bit/ word	208	408	1008	2008
	64bit/ word	336	656	1616	3216
BER		122	248	604	1204
PER		82	168	404	804

(e) 符号化データ長

図7　可変長の要素を含む場合の測定結果

符号化/復号プログラムの符号化/復号処理時間を表5に示す。比較のためBERでの測定結果も示す。

表5　実際の応用層プロトコルのデータ要素における符号化/復号処理時間とデータ長

構造	データ要素	符号化規則	符号化時間(ミリ秒)	復号時間(ミリ秒)	データ長(オクテット)
a	m-Get操作要求	LWER	0.86	1.10	216
		BER	3.80	3.92	147
b	m-Get操作結果	LWER	5.24	5.09	1424
		BER	11.20	12.35	658
c	IM-UAPDU	LWER	1.60	1.45	452
		BER	3.12	2.89	370

注) LWERは ワード長32ビット, バイト順序BigEndianの転送構文使用

ここでは、各データ要素に対しては以下の条件を設定した。

(1) 構造aの場合

CMIPのm-Get操作要求の相対識別名を5個持ち、同期のモードおよびスコープの指定をして、全属性の読み出し要求をした場合。

(2) 構造bの場合

CMIPのm-Get操作結果は、長さ8オクテット以下の50個の属性値の検索結果を示す場合。

(3) 構造cの場合

'84年版MHSのIM-UAPDUで、200オクテットのボディパートを持つ場合。

4.2 生成される符号化/復号プログラムの規模

LWER処理系を使用した場合のFTAMとCMIPの符号化/復号プログラムの規模を表6に示す。この符号化/復号関数のステップ数はワード長が32ビットで、2つの転送構文(BigEndian/ LittleEndian)をサポートした場合である。

表6 符号化/復号プログラム規模 (C言語ステップ数)

抽象構文	C言語型定義等	符号化/復号関数	共通関数	計
FTAM	0.7 k	19.4k (57.4k)*	1.8k	21.9k (59.9.k)*
CMIP	0.6 k	11.8k (35.9 k)*		14.2k (38.3.k)*

*( ) 内の数字は6種類の転送構文の場合のステップ数

LWER処理系により生成されるC言語の型定義と符号化/復号関数のステップ数は、FTAMで21.9k、CMIPで14.2kステップとなった。また、総ステップ数はFTAMで21.9kステップ、CMIPで14.2kステップとなる。

5. 考察

5.1 LWER符号化/復号処理系について

(1) 符号化/復号プログラムの自動生成

ASN.1による抽象構文定義からC言語型定義ならびに符号化/復号関数を含む符号化/復号プログラムを自動生成するための生成規則を明確にし、この規則に基づいたLWER処理系を作成し、その動作を確認した。生成する符号化/復号関数は、従来の基本符号化規則(BER)等の処理系とは異なり、抽象構文定義に対応するプログラミング言語の型を持つ変数値を符号化結果格納領域に直接書き込むことで実現できた。

(2) 生成プログラムの符号化/復号処理時間

LWER処理系により生成された符号化/復号プログラムの処理時間は、自システムと同一の転送構文を使用する際には、固定長の要素のみの場合BERと比較して20倍以上高速となり、また、実際の応用層プロトコルの場合には約2~4倍程度高速となることがわかった。また、自システムと異なる転送構文の場合においても、一方の計算機は自システムの転送構文を使用することとなり、例えば、4.1.2節のm-Get操作要求では、通信時の符号化および復号時間の合

計が、LWERで約2.8秒(32Bit/LittleEndianの転送構文の場合の復号時間は1.89ミリ秒)、BERで7.7秒となり2.7倍程度の高速となった。このため、今回実装したLWER処理系が生成する符号化/復号プログラムが実用的な性能を達成していると考えられる。

(3)生成プログラムの規模

LWER処理系が生成するプログラムは、高速化のため転送構文毎に専用プログラムとしている点により、サポートする転送構文の数に比例してステップ数が増加する傾向にある。しかしながら、現在のワークステーション等では、ワード長が一種類(32ビット)でバイト順序が2種類(Big Endian / Little Endian)のものが一般的であるので、2種類の転送構文のサポートで十分であることを考慮すれば、4.2節で示したように、FTAM、CMIPに対する規模は、それぞれ21.9k、14.2kステップとなり、比較的コンパクトだと思われる。

5.2 LWERの適用領域と制約について

(1)適用領域

5.1節(2)の結果より、LWERは、伝送時間よりエンドシステムにおける処理時間が全体の通信時間に影響する高速なネットワークで有効である。しかし、LWERの符号化データ長がワード単位の符号化、Character Stringのワードによるアライメント、ポインタの使用等により長くなるため、低速なネットワークで転送時間が性能を支配する場合には、BERやPERの方が有利になると思われる。

(2) データ構造上の制約

ワード単位の符号化であるため、INTEGERの値、ポインタおよび長さの値に制約が加わる(例えばワードサイズが16ビットの時INTEGERの正の最大値32,767)が、実際に使用されるデータ値や長さを考えると実用上大きな制約とはならないと考えられる。

5.3 転送構文の選択と折衝について

(1) LWERにおける転送構文の選択

計算機のワード長やバイト順序も異なる場合には、処理能力の低い計算機の転送構文に合わせることが有効である。

(2) 複数符号化規則における選択と折衝

OSIでは、通信時に使用する符号化規則は、プレゼンテーション・コネクション確立時に折衝を行って、動的に決定される。PERの符号化/復号処理時間はBERより高速となる場合が多い[7]が、LWERで自システムと同一の転送構文を使用した場合より高速とはならない。さらに、BERは実装上必須であるとともに、幅広く普及している現状を考えると、高速ネットワークでのイニシエータ側での符号化規則の提案の順は、第1にLWERで自システムの転送構文,第2にLWERで自システムと異なる転送構文またはPER、第3にBERとして提案することが有効と考えられる。

6. おわりに

本稿では、ASN.1ライトウェイト符号化規則(LWER)を扱うOSI応用層プログラムを容易に開発可能とするため、ASN.1による抽象構文からLWER用符号化/復号プログラムを自動生成するLWER処理系のソフトウェア実装とその評価結果について報告した。

具体的には、ASN.1による抽象構文定義からプログラミング言語(C言語)用型定義ならびに符号化/復号関数を含むの符号化/復号プログラムを自動生成するための生成規則を明確にし、特に、符号化/復号関数では、従来の基本符号化規則(BER)等の場合とは異なり、抽象構文定義に対応するプログラミング言語の型を持つ変数値を符号化結果格納領域に直接書き込むことで実現できることを示した。また、実装したLWER処理系を、生成された符号化/復号プログラムの処理時間ならびにプログラム規模の観点より評価した。この結果、符号化/復号処理時間については、応用層でよく使用されるINTEGERを複数持つデータ要素の場合にはBERの約20倍以上高速となり、また、プログラム規模も比較的コンパクトになることを確認し、実装したLWER処理系が実用できることを示した。さらに、LWERは、符号化データ長がBERに比較して長くなり伝送時間が増加する傾向にあるため、特に、伝送時間よりエンドシステムにおける処理時間が全体の通信時間に影響する次世代LANや広帯域ISDN(ATM等)などの高速通信環境で有効となることを示した。

最後に日頃御指導頂くKDD研究所 小野所長、浦野次長に感謝します。

参考文献

[1]: ISO/IEC 8824 "ASN.1", 1989.
[2]: ISO/IEC 8825 "ASN.1 Baasic Encoding Rules", 1989.
[3]: ISO/IEC CD 8825-2 "Packed Encoding Rules", July 1991.
[4]: ISO/IEC CD 8825-3 "Distinguished Encoding Rules", July 1991.
[5]: ISO/IEC JTC1/SC21 N6131, "Working Draft for Light Weight Encoding Rules", July 1991.
[6]: Christian Huitema, Assem Doghri, "Defining Faster Transfer Syntaxes for the OSI Presentation Protocol", ACM SIGCOMM Computer Communication Review, vol19, No.5, Oct.1989.
[7]: 堀内,小花,鈴木, "OSI応用層プロトコル用ASN.1圧縮符号化規則の圧縮特性に関する一考察", 情処研資DPS-50-3, May. 1991.
[8]: 堀内,小花,鈴木, "ASN.1ライトウェイト符号化規則の評価", 情処第44回全大,4L-09,Mar. 1992.
[9]: 堀内, 小花, 鈴木、 "OSI応用層プロトコル実装におけるASN.1新符号化規則の適用性に関する検討", 1992年電子情報通信学会春季全国大会, B-602,Mar. 1992.
[10]: 長谷川, 野村, 堀内 "ASN.1支援ツールの開発-コンパイラおよびエディタ-", 情報処理学会マルチメディア通信と分散処理研究会,39-4, Sept.1988.
[11]; G.Neufield, Y. Yang, "The Design and Implementation of ASN.1 -C Compiler", IEEE Trans. Software Engineering Vol.16, No. 10, Oct. 1990.
[12]: 中川路, 勝山, 宮内, 水野, "OSI抽象構文記法支援ソフトウェア APRICOTの開発と評価", 電子情報通信学会論文誌,J73-D-I, No.2, 1990.