マルチメディア通信と分散処理　46－5
（1990．　7．　12）

SDLとCを組み合わせた通信プログラム仕様の記述法及びその処理系

長谷川亨　　野村真吾　　瀧塚孝志
国際電信電話株式会社 上福岡研究所

通信プログラムの実装コストを削減するためには、プロトコルの形式記述技法(FDT)で記述したプロトコル仕様を通信プログラムに変換するFDTトランスレータを用いることが有力な解決法の一つである。状態遷移に基づいて仕様を記述するSDLは、FDTトランスレータを実現するのに有望なFDTである。しかし、SDLだけで通信プログラム仕様の全てを記述することは困難であり、SDLで記述した仕様から完全な通信プログラムを生成することはできない。そこで、筆者らは、SDLで記述できない部分をプログラミング言語Cで記述する方法を考察し、この仕様をC言語のプログラムに変換して、実行する処理系を作成した。

Automatic Implementation System for Telecommunication Software Specifications based on SDL and C

Toru Hasegawa　Shingo Nomura　Takashi Takizuka
KDD Kamifukuoka R & D Labs.
2-1-15, Ohara, Kamifukuoka-shi, Saitama, 356 Japan

FDT (Formal Description Technique) compilers can reduce implementation costs of communication software drastically. We have adopted SDL standardized by CCITT as a target FDT, and have implemented a SDL compiler to C language. However, we cannot specify implementation dependent matters in SDL such as interfaces with computer systems. We, therefore, have proposed a protocol specification method to which C language and SDL are combined in order to specify the implementation dependent matters. This paper describes the proposed specification method and the details of the implemented compiler.

1. はじめに

通信プログラムの実装コストを削減するためには、形式記述技法(FDT)で記述したプロトコル仕様を通信プログラムに変換するFDTトランスレータを用いることが有力な解決法である。プロトコル仕様は状態遷移表により記述されることが多く、有限状態機械に基づいて仕様を記述するSDL[1]、Estelleは、状態遷移表との親和性も高く、FDTトランスレータを実現するのに有望なFDTである。SDL、Estelleで記述したプロトコル仕様を通信プログラムに変換するトランスレータ[2-5]を開発する試みも各所で行われている。

筆者らは、SDLで記述した仕様をAdaの通信プログラムに変換するトランスレータ[3]の開発を通じて、SDLによるプロトコル仕様からだけでは、完全な通信プログラムは生成できないことを明らかにしてきた。これは、SDLがプロトコル仕様を記述するための言語であり、プロトコル仕様では規定されない、実装に依存した処理等を含む通信プログラム仕様を完全に記述できないためである。そこで、SDLで記述できない部分をプログラミング言語で記述できるように、SDLを拡張することにより、通信プログラム仕様を記述できる方法を検討した。さらに、この拡張したSDLで記述した仕様をC言語のプログラムに変換して、実行する処理系を作成した。本稿では、拡張した仕様の記述法、処理系の実装について述べる。

2. 設計方針

FDTにより記述したプロトコル仕様を通信プログラムに変換して実行する処理系を実現するには、FDTおよびターゲットのプログラム言語の選択が重要である。筆者らは、以下に述べる理由から、それぞれSDLとC言語を採用した。有限状態機械に基づいてプロトコル仕様を記述するSDLは、状態遷移表との親和性が高いため、記述が容易で、プログラムに変換することも可能である[3,5]。CCITTで標準化されたSDLは、各種プロトコル仕様の記述に使用されており、C言語も通信ソフトウェアの開発に広く使用されているため、作成したトランスレータが広く使用されることも期待できる。

2.1 SDLの概要

SDL[1]は、有限状態機械に基づいて通信プロトコル仕様を記述するためのFDTである。SDLには、人間が読み易いように図形的な表記を用いるSDL/GRと、計算機で処理し易いようにプログラム的な表記を用いるSDL/PRの2種類の表現形式がある。

SDLにおいて、通信プロトコルは、1つ以上のブロックと、ブロック間又はブロックとシステムの環境を結ぶチャネルから構成される。ブロックの中には、並列に動作するプロセスが1つ以上存在し、固有のプロセス識別子が割り当てられる。プロセスの動作は有限状態機械に基づいて記述され、勧告に規定される状態遷移表の動作を記述することができる。プロセス間の通信はシグナルの授受によって行われる非同期通信である。シグナルは宛て先のプロセス識別子を指定して、チャネルを介して送信される。データ型は、値とそれを操作する方法を併せて記述する抽象データ型(ADT)を用いて定義する。

2.2 問題点

効率的な通信プログラムの開発を可能とするSDL処理系を実現するには、以下の点に注意する必要がある。

(1) SDLで記述できない仕様

SDLはプロトコル仕様の記述に適しているが、完全な通信プログラムに変換するのに必要な全ての情報を含む通信プログラム仕様を記述できない。例えば、変換したプログラムのシステムへの組み込み方など、実装に依存した処理を記述できない[2-5]。

(2) 並列動作の実現

標準的なC言語には、Ada等の並列プログラミング言語と異なり並列プログラミングの機能が無いため、SDLが提供する並列に動作する単位のプロセスを実現する方式が重要になる。

以上の問題を解決するために、2.3、2.4節の方針に基づいて、SDL処理系を構築することにした。

2.3 通信プログラム仕様の記述法

一般に、通信プログラム仕様は、

① 状態遷移により規定されるプロトコル動作、
② 抽象データ型を用いた、プロトコル要素のフォーマッティング等のデータ処理、
③ 個々のシステムに依存した動作、
④ 外部プログラムとのインタフェース

の4種類の仕様から構成される。

SDLは、①のプロトコル動作の記述に適しており、有限状態機械として表現されるプロセスを用いることにより、プロトコル動作を容易に記述することができる。また、そのプロセス仕様をC言語のプログラムに変換することも可能である。

しかし、②から④の処理に関しては、以下に述べる理由から、C言語により記述することにした[6]。

- ②の抽象データ型による仕様は、代数的意味を定義し、実現方式等を隠蔽する点に特徴がある。しかし、抽象データ型を用いて代数的な仕様を記述することは、プログラマにとって容易でない。

抽象データ型による仕様を効率的なプログラムに変換することは容易でなく、変換されたプログラムの高速な実行も期待できない。また、大規模な仕様を対象とする場合、Cのデータ型定義と抽象データ型の代数定義の等価性を検証することは困難なため、その実現を直接C言語により記述することにした。

- ③、④の実装に依存した仕様をSDLにより記述することは不可能である。HCP等の手続き的なプログラム仕様を記述するための仕様記述法をSDLに組み込むことも考えられるが、処理系の開発を簡単にするため、ターゲット・プログラミング言語によりコーディングする方法が一般的である[2-5]。

従って、プロセスや状態遷移の記述はSDL、それ以外の処理は全てC言語で記述する。ただし、SDLとC言語を組み合わせて仕様を記述することになるため、以下の点に留意して、2つを組み合わせる必要がある。

(1) SDLまたはC言語単独で記述できる処理に関しては問題無いが、変数の宣言や、プロセス間通信におけるデータの授受等の、C言語による記述とSDLによる記述が混在した記述が持つ意味を明確にする必要がある。

(2) トランスレータが、SDL記述だけでなく、C言語による記述の構文や意味の検査を可能とするような仕様の記述法を検討する必要もある。

2.4 並列プロセスの実現

SDLプロセスの粒度は、OS(オペレーティングシステム)が提供する並列動作の単位プロセスに比べると小さいため、OSプロセスの内部で並列性を実現することができる。また、SDLプロセス間で頻繁に通信を行うため、SDLプロセスをOSプロセスに対応させると、高速な実行を期待できない。そこで、SDLプロセスをライトウェイトプロセスに対応させることにした。

SDLプロセスをライトウェイトプロセスとして実現するには、① 既存のC言語のライトウェイトプロセス・パッケージを利用する方法と、② SDLプロセスをライトウェイトプロセスとして並列に実行するスケジューラを作成する方法が考えられるが、以下に述べる理由から、②の方式を採用した。

- プロセスの優先順位やプロセススイッチ等をユーザプログラムとして実現できるため、パッケージに任せることに比較して細かなプロセス制御が可能となる。
- 現在、標準的なライトウェイトプロセス・パッケージが存在せず、特定のパッケージを使用すると、他の計算機への移植が困難になる。
- 個々のOSに依存した処理をスケジューラで扱うことにより、トランスレータが生成するプログラムに影響を与えることなく、異なるOSに移植することが可能になる。

従って、SDL処理系(図1)は、トランスレータと並列プロセスのスケジューラから構成される。トランスレータは、複数のプロセスから構成される仕様をプロセス毎にCの関数に変換する。スケジューラはCの関数に変換されたSDLプロセス群を、OS上の1プロセス内でライトウェイトプロセスとして並列に実行する。

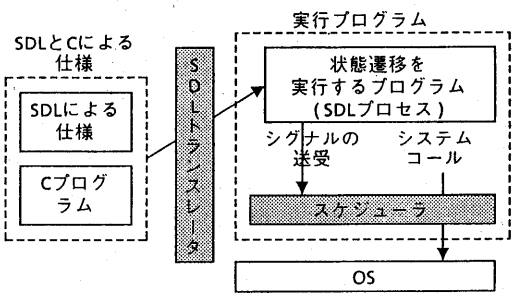

図1 SDL処理系の構成

3. トランスレータ

本章では、SDLの言語仕様の拡張法とCプログラムに変換するトランスレータについて述べる。

3.1 SDLの言語仕様の拡張

3.1.1 方針

2.3節で述べた条件を満足するように、以下の方針に従って、SDLの言語仕様を拡張した。

① 抽象データ型の代わりに、C言語のデータ型を採用する。

② 言語仕様の拡張はSDLのコメントまたは注釈部分に対して行うことにし、SDLの構文規則を変更しない。

③ 記述者が仕様を正しく記述できるように、C言語による記述とSDLによる記述が混在した記述が持つ意味を明確にする。具体的には、変数の宣言、変数に対する操作は、C言語の意味に従い、それ以外はSDLの意味に従う。例えば、プロセス間通信に伴うデータの授受等は、SDLの意味に従う。

3.1.2 拡張

OSIトランスポートプロトコル仕様の記述例(図2)を用いて、拡張部分について説明する。

(1) データ型定義

予約語*NEWTYPE*で指定されるSDLのデータ型定義における、値の性質を数学的に定義する公理の代わりに、直接Cでデータ型を定義する。図2の(c)に示されるように、SDLの注釈として記述する。

(2) 変数宣言

抽象データ型の代わりにCのデータ型を採用したため、*DCL*文で宣言される変数が実体かポインタかを指定する必要がある。そこで、宣言する変数の名前の表記はCに準じる(図2の(d))。

(3) 手続き的な処理

変数に対する代入等の手続き的な処理は、SDLのタスクで記述する。データ型定義と同様にタスクに対する注釈として記述する(図2の(e))。

(4) システム、ブロック、プロセス仕様

既に作成したCの関数や標準のCライブラリ関数を再利用することができるように、ブロック、プロセス仕様のコメント部に、Cコンパイラへの制御情報を記述できるようにした。図2の(a)では、include文、extern宣言により、Cのヘッダファイルのインクルード、関数の外部宣言が行われている。データ型として抽象データ型の代わりにCのデータ型を採用したため、Cの関数の再利用が可能になっている。

(5) 環境とのインタフェース

SDLでは、仕様の外部との相互動作は、環境と接続されたチャネルを介しての、シグナルの送受としてとらえられている。そこで、変換されたプログラムが外部のプログラムとインタフェースする処理を、環境と接続されたチャネルに対するコメントとして記述する。例えば図2の(b)では、スケジューラ関数receiveSignalを呼び出して、外部プログラムからシグナルを受け取り、シグナルのあて先のSDLプロセスに出力している。

```
SYSTEM T70
 COMMENT '
    #include <stdio.h>
    extern convert(), receiveSignal(), output(); ' ; ··· (a)
CHANNEL TSIN FROM ENV TO TPM WITH TCONreq
 COMMENT '
    SIGNAL *sig;
    sig = receiveSignal();
    if (sig->memid = CONreq) {
          output(sig); } ····· ' ; ···· (b)
ENDCHANNEL TSIN ;
NEWTYPE ADDRESS
    'typedef struct{ int size;
          char *str;} ADDRESS; ' ; ···(c) ENDNEWTYPE;
SIGNAL TCONreq(ADRESS); NCONreq(ADDRESS);
PROCESS TPM( 0, );
    DCL  *TSap, *NSap ADDRESS, ref int ; ···(d)
    START ; NEXTSTATE CLOSED;
    STATE CLOSED;
       INPUT TCONreq (TSap) ;
       TASK ' ref = 1; NSap = convert(TSap, ref); '; ··(e)
       OUTPUT NCONreq(NSap) TO Pid ;
       NEXTSTATE WFNC;
       ····
ENDPROCESS TPM ;
    ···
ENDSYSTEM;
```

図2 仕様の記述例

3.2 トランスレータの実装

トランスレータの実装方針を以下に示す。

(1) 言語仕様は、1988年版Basic SDLを対象とする。

(2) ISDNやOSI等の階層構造を持つ、大規模な(100K行以上の)通信プログラムを生成できるようにする。

(3) OS上の1プロセスとして実行されるプログラムを生成する。

(4) OSに依存しないCのコードを生成する。

(5) SDL言語仕様で規定される意味検査を全て行う。

(6) 変換したプログラムをCコンパイラ/リンカによりコンパイル/リンクするだけで実行できるようにする。すなわち、使用者が生成したプログラムを変更する必要がないようにする。

トランスレータはC言語により実装し、構文解析部、意味検査部、プログラム生成部から構成される。

3.2.1 構文解析部

UNIX上のツールyacc/lexを用いて実装した。入力仕様を、31種類のSDLのシステム、プロセス、遷移等の言語要素を表現するノードからなる構文木に変換する。また、構文解析部は、SDLで規定される構文規則に関する構文エラーや変数の2重定義等の検査を行う。エラーを検出すると、エラーの種別、行番号を出力して停止する。

SDLを拡張したC言語による記述は、SDLのコメントに対応するノードに文字列として保存する。トランスレータの開発効率を考慮して、構文解析部では、C言語の記述の構文解析を行わない。ただし、C言語により定義されたデータ型の型名および変数名に関しては、名前表に登録して、データ型および変数の未定義、2重定義を検査する。

3.2.2 意味検査部

構文解析部の出力である構文木を用いて、SDLの構文規則だけでは検査できない、チャネル定義の誤りや、シグナルルートとプロセスに対する入力シグナルの一貫性等の意味検査を行う。また、プロセスの状態遷移を実現するプログラムを生成し易いように、状態を表現するノードに、状態間の関係等の付加的な情報を加える。

3.2.3 プログラム生成部

プログラム生成部は、SDLのデータ型のスコープルールを実現するために、システム、ブロック、プロセス仕様毎に、ヘッダフィルを生成する。また、システム仕様とプロセス仕様からソースファイルを生成する(図3)。

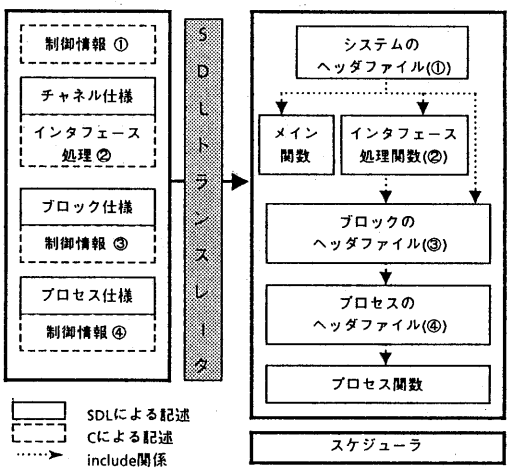

図3 トランスレータの機能

3.2.3.1 ヘッダファイル

ヘッダファイルには以下のコードを生成する。

- システム、ブロック、プロセスのコメント部に記述されたCコンパイラへの制御情報(3.1.2節(4)参照)。
- ブロック、プロセスのデータ型定義部に記述されたC言語によるデータ型定義(3.1.2節(1)参照)。
- SDLのシグナルに対して一意な識別子を割り当て、定数としてシステムのヘッダファイルに生成する。シグナルは、この識別子と実際のパラメータを持つフィールドからなるCの構造体で定義され、その型定義もシステムのヘッダファイルに生成する。

3.2.3.2 ソースファイル

(1) プロセス

プロセス仕様は、Cの関数(プロセス関数)に変換する。プロセスの動作は、シグナルの受信、受信シグナルに対する処理、シグナルの送信の繰り返しである。そこで、プロセス関数は、この一連の処理のループとして実現する。図4に、図2の仕様内のプロセス*TPM*を変換した例を示す。

プロセスの状態遷移の動作は、状態と入力シグナルをラベルとした、2重のswitch文により実現する。状態名、シグナル名を、Cのdefine文により定義することにより、元のSDL記述における名前を保存して、遷移の記述との対応付けを容易にしている。SDLにおいては、無効な受信シグナルは無視すれば良いが、変換されたCプログラムでは、無効シグナルのメモリ領域を解放する必要がある。そこで、受信したシグナルがswitch文のどのケースにも合致しない場合、無効シグナルとそのパラメータのメモリ領域を解放する。

プロセスの生成、シグナルの送受信等の、プロセスの機能は、スケジューラの提供する関数呼出しに変換する。図4の関数getSignal、outputSignalは、シグナルの送信、受信を行うスケジューラ関数である。

STOP文によりSDLプロセスが終了する時は、まずローカル変数のメモリ領域を解放した後、プロセスを消滅させるスケジューラ関数terminateProcessを呼び出して、終了する。関数terminateProcessでは、シグナルのキュー等の、プロセスの持つ資源を全て解放する。

```
void TPM()
{
 SIGNAL *insignal, *outsignal;
 ADDRESS *Tsap, *NSAP;
 int ref;
 int state;
 ProcessID Pid;
 初期化;
 for(;;)      {
   insignal = getSignal(); /*入力シグナルの受信*/
   switch(state) {
      case (CLOSED):
         switch((*insignal).memid) {
            case (TCONreq):
               ref = 1;
               NSap = convert(TSap, ref);
               出力シグナルoutsignalの作成;
               outputSignal(outsignal, Pid);
               state = WFNC;
               break; }
            .....
            defalut:
               入力シグナルinsignalのメモリ領域を
               解放; }
      ..... } }
}
```

図4 プロセスから生成されるプログラム

(2) システム

システム仕様からはメイン関数を生成する。この関数では、システム全体を管理するグローバル変数の初期化等を行なった後で、仕様の実行が開始された時点から存在するプロセスの生成を行って、プログラムの実行を開始する。3.1.2節(5)のインタフェース処理のC言語による記述は、外部プログラムとの間のインタフェース処理を行う関数として生成する。この関数は、SDLのプロセスと同様にプロセス(環境プロセス)として並列に実行され、環境との間のシグナルの送受は環境プロセスとの間で行われることになる。

4. スケジューラの実装

スケジューラ(図5)は、SDLのプロセスが提供する機能を実現する。個々の機能は、Cの関数として実現される。スケジューラの機能として、プロセスの生成/消滅の実行、プロセス間通信、タイマの管理を提供する。プロセスの消滅およびプロセス間通信の処理実行後にSDLプロセスのスケジューリング機能が呼び出されて、プロセス切り替えを実行する。

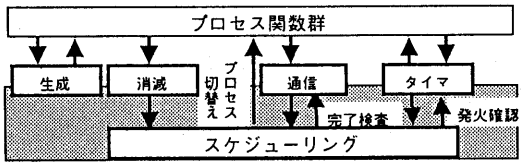

図5 スケジューラの機能

以下では、VAX/VMS上に実装した結果について述べる[7]。ただし、作成したスケジューラを汎用のライトウェイトプロセス・パッケージ上に移植できるように、種々のパッケージに標準的な機能を念頭において実装した。

また、以降、SDLで定義されるプロセスをSDLプロセスと呼び、スケジューラが実現するライトウェイトプロセスをプロセスと呼ぶことにする。

4.1 並列実行のメカニズム

スケジューラは、SDLプロセスから変換された個々のプロセス関数に対して固有のスタックを割当て、プログラムカウンタ(PC)/スタックポインタ(SP)等の動作情報を管理してコルーチンとすることによりプロセスの並列動作を実現している。スケジューラはCの関数群として実現され、プロセス関数からスケジューラ関数を呼び出すことにより実行される。図6にスケジューラの構成を示す。

一般に関数を呼び出す時には、実行中のプログラムカウンタ等のレジスタ情報をスタック上に格納する(コールフレーム)。関数から制御が戻る時には、このコールフレームのアドレスを指定して呼び出し時の状態を回復する。本スケジューラは、この機構を利用し、スケジューラの関数を呼び出した時のコールフレームのアドレスを管理してプロセス切り替えを実現する。プロセスの生成では、割当てられたスタック上にプロセス関数の開始アドレスと関数の引数を設定したコールフレームを作成し、プロセスが選択されるとこのコールフレームをもとにプロセスが起動される。VAX/VMSのデバッガがそのまま使用できるように、各プロセス関数がmain関数に呼び出されているとするフレームを余分に作成している。

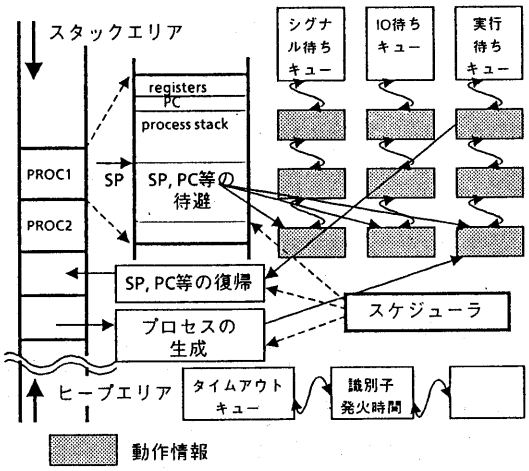

図6 スケジューラの構成

Ada, SIMULA等の並列言語がサポートできる機種であれば、プロシージャスタックの管理がOSとユーザで独立であるため、格納すべき情報の違いはあるがコールフレームを用いた同様なメカニズムの構築が可能である。

スケジューラの作成において、移植性を考慮してほとんどの機能をC言語により記述している。しかし実現上の問題から、次の2点についてはアセンブラで記述した。

① 現時点のスタックポインタの値を得る。
② コールフレームを用いて次のプロセスに制御を移す。

4.2 プロセス管理

スケジューラ上でプロセスは、実行中/実行待ち/シグナル待ち/IO待ち(外部プログラムとの間の入出力待ち)のいずれかの状態にある。スケジューラは個々のプロセスの動作情報を、各状態に対応するキューにつないで管理する。

SDLプロセスの動作は、シグナルを受信し、受信したシグナルに対応した処理を実行して、次のシグナルの受信を実行する一連の動作の集まりである。このためプロセスの切替えは、この一連の動作を単位として、シグナルの受信を実行する時に行えば十分である。実行待ちキューにつながれた実行可能なプロセスの先頭からプロセスを選択し、動作情報を復帰して、プロセスを実行する。

4.3 プロセス間通信

プロセスに受信シグナルのキューを持たせることにより、SDLが規定する非同期式の通信を実現した。シグナルはデータを運ぶためにパラメータを持つことができ、送受信プロセス間でのパラメータ値の授受はコピーにより行われる。従って、シグナ

ルのキューを効率的に実現しないとパラメータ値の無駄なコピーが多くなり、処理効率が悪くなる。そこで、パラメータ値へのポインタをキューを介して送受することにし、実際のパラメータ値のコピーは受信プロセスが行うようにした。これにより、シグナルのパラメータのコピーが1度だけになり、効率的なコピーが可能になる。

4.4 外部プログラムとの通信

外部のプログラムとの通信(IO:入出力)は、OSを介して実行されるため、システムコールを用いたメッセージの交換として実現される。このため、プロセスが外部プログラムからの受信待ちを行うと、他に実行可能なプロセスが存在しても、プログラム全体(OSプロセス)がウェイトしてしまう。

そこで、外部のプログラムとの通信によりプログラム全体が待ち状態になることのないように、スケジューラがまとめてこの通信を管理する。プロセスが通信の要求を行う時、スケジューラが呼び出され、このプロセスをIO待ちキューにつなぎ、VAX/VMSが提供する、待ち状態にならない入出力のシステムサービスを実行する。スケジューラがスケジューリングを実行する毎に、IO待ちキューをサーチして、システムサービスの完了を検査し、受信の成立したプロセスを実行待ちキューにつなぎ替える。これによりIO待ちにおいてもプログラム全体がウェイトしない構成となり、並列性を保証できる。

4.5 タイマ

スケジューラは、ユーザスタックを独自に管理しているため、OSから割り込みを受けると動作が保証されない。このためタイマの実現において、OSの提供する割り込み機能を使うことができない。そこで、タイマの時間情報をキューにつないで管理し、スケジューラが呼ばれる毎に絶対時間を受け取って、タイマの発火を確認する。具体的には、タイマ管理テーブルを用意して、タイマの生成、消滅に応じて、タイマ管理テーブルにタイマ識別子を登録、抹消する。タイマが設定されると、タイマ識別子と発火時間の情報をタイムアウトキューにつないで管理する。このキューではタイマの発火する順につなぐことにより、タイマ発火の確認処理を単純化している。

5. 結果と考察

実装したトランスレータ及びスケジューラを用いて実際に通信プログラムを開発して、それぞれを評価した。

5.1 結果

(1) トランスレータの評価実験

トランスレータを評価するために、OSIトランスポートプロトコル(TP)クラス0の仕様を、拡張したSDLで記述し、トランスレータによりCプログラムに変換した。その内訳を表1に示す。

SDLによる記述が約500行、C言語による記述が約700行の合計約1200行で、仕様を記述することができた。トランスレータは、この約1200行の仕様を、約2500行のCプログラムに変換している。

TPクラス0の記述	言語	記述量	生成プログラム
プロトコル動作	SDL	500行	1800行
手続き的な動作	C	300行	300行
システム依存の動作	C	50行	50行
インタフェース	C	350行	350行
計	C	1200行	2500行

表1 変換結果

(2) スケジューラの評価実験

スケジューラのプロセス切り替え時間を評価するために、VAX 8700(約6MIPS)において、4個のプロセスを実行させて、切り替え時間を測定した。プロセス切り替え時間は30μs程度であり、満足できる速度が達成できた。これはSDLのすべてのプロセスがアドレス空間を共有しており、プロセス切り替えにはレジスタ情報の格納と回復のみでよいためである。

(3) 通信実験

先に作成したセションプロトコル(SP)プログラムと生成したTPクラス0プログラムをVAX8700上で組み合せて動作させ、μVAX上のTP/SPプログラムの間で、X.25網上でファイル転送実験を行い、変換されたプログラムが正しく動作することを確認した。

また、変換したプログラムの処理速度を評価するため、TPクラス0プログラム上でデータを送信、受信するプログラムを動作させ、データ転送実験を行った。送信、受信プログラムとTPクラス0プログラムはそれぞれ、OS上のプロセスとして実行し、プログラム間の通信はVAX/VMSのメールボックスを用いた。TPプログラムでは、送信側と受信側のSDLプロセスを含めて、合計7個のSDLプロセスが実行されている。ネットワークレーヤは、TPクラス0プログラム内の内部折返しにより行っている。SPDUサイズ2048バイト、TPDUサイズ1024バイトで、1Mバイトのデータを転送し、約108.3Kバイト/秒のスループットを達成した。

5.2 考察

以上の実験結果より、以下のことが考察される。

(1) 変換したプログラムの処理効率

5.1節の実験結果より、TPクラス0の数倍程度までの規模のプロトコルを対象とした場合、イーサネット等のLAN上で動作する通信プログラムの開発に、本処理系は十分適用可能である。また、OSI FTAMプロトコルのようにOSIの全レーヤを実装するプロトコルにおいても、本処理系を用いて開発したプログラムは、10-20Kバイト/秒程度のスループットを達成できると考えられる。

(2) 開発効率。

人手によりプログラム言語Adaを用いて実装したTPクラス0プログラムは約3000行であるが[8]、本実験では約1200行のプログラム仕様を記述するだけで良かった。記述した言語が異なるので、直接比較することはできないが、全て人手によりコーディングするのに比較して、開発効率が大幅に向上している。また、変換したプログラム規模も2500行であり、人手によりコーディングしたプログラムに劣らないコンパクトなプログラムである。

また、TPクラス0の仕様は、ISOが作成した文献[9]のSDL記述に、プロトコル要素のフォーマッティングやシステムに組み込むためのプログラムである約700行を、新たに記述するだけで良かった。これは、構文規則を変更せずに、SDLの拡張を行ったためである。標準/勧告に付加されるSDLによるプロトコル仕様を用いる場合、大幅に開発効率を向上させることができる。

(3) デバッグ、修正のし易さ

デバッグは変換したCプログラムで行われることになるが、Cプログラム上で誤りを検出し易く、その誤りを容易にSDL仕様上の誤りに対応づけることができた。これは、トランスレータが、状態名、シグナル名等の名前や、条件分岐等の構造を、元のSDL記述のものを保存して、プログラムに変換するためである。

(4) C言語による記述の検査

C言語による記述における誤りをトランスレータが検査しないため、その誤りは変換したプログラムをCコンパイラによりコンパイルする時点で初めて検出される。トランスレータがC言語による記述も検査を行うことが望ましいが、TPの実装に使用した経験から、現実上は問題無かった。

トランスレータが検査しないのは、タスク及びチャネルに付加されたC記述だけであるので、誤りを特定する範囲が限定されるためである。さらに、SDLの仕様と生成したプログラムの対応が明確になるようプログラムを生成しているので、コンパイルエラーに対応する仕様における誤りを容易に特定できる。

(5) 移植性

トランスレータが生成したプログラムは、スケジューラを移植するだけで、他のOS上で実行できる。これは、生成されたプログラムが、スケジューラが提供する関数以外の関数を呼び出さないためである。さらに、スケジューラは、Cのソースコードで1.5k行程度であるが、この内VAX/VMSに依存した部分は200行以下(アセンブラは20ステップ)であり、移植性は高い。また、これら機種依存の処理を関数やマクロを用いてまとめて記述することにより、他の機種への移植性を高めている。

6. おわりに

本稿ではSDL処理系の設計、実装について報告した。現在、評価を進めるとともに、プロトコル要素がASN.1を用いて定義されるプロトコルも対象とできるように、SDLトランスレータとASN.1コンパイラの組合せを進める予定である。最後に日頃御指導頂くKDD上福岡研究所小野所長、安藤次長、小西通信ソフトウェア研究室長、鈴木コンピュータ通信研究室長に感謝する。

参考文献

[1]: CCITT Rec. Z.100, Oct. 1987.

[2]: 長谷川,堀内,加藤, "EstelleとASN.1に基づくプロトコル仕様からAdaへのトランスレータ",信学論B-II, pp 317-327, April 1990.

[3]: 野村,長谷川,堀内,"SDLからAdaへのトランスレータ",信学交換システム研究会, SSE88-115, Oct. 1988.

[4]: G. V. Bochmann, et al,"Semi-automatic implementation of communication protocols",IEEE Trans. Software Eng., SE-12, 8, pp 827-843, Aug. 1986.

[5]: 田中, 佐藤,宗森,水野,"SDLプログラマの提案", 昭和64年前期情報処理学会全国大会, 7M-2, Oct. 1989.

[6]: 長谷川,野村, "SDLとCを組み合せた通信プログラム仕様の記述法及びその処理系", 1989年信学全大, B-329, March 1989.

[7]: 野村,長谷川,瀧塚, "SDL処理系のための並列プロセス処理環境の実装", 1989年信学秋季全大, B-189, Sep. 1989.

[8]: 堀内,長谷川,加藤, "AdaによるOSIトランスポート/セションプロトコルの設計と実装", 情報処理学会マルチメディアと分散処理研究会, 34-3, July 1987.

[9]:ISO PDTR10167,"Guidlines for the Application of LOTOS, Estelle, and SDL", December 1989.