# 正規表現とプロセス代数に基づく通信プロトコルコンパイラ

A Compiler for Communication Protocols
Based on Regular Expressions and Process Algebra

服部 健太 †,††
Kenta HATTORI

数馬 洋一 †
Youichi KAZUMA

†(株) システム計画研究所
Research Institute of Systems Planning, Inc.
†† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
{hattori@isp.co.jp, kazuma@isp.co.jp}

近年，インターネットの普及と利用の高度化にともなって，通信プロトコルを実装する機会が多くなってきた．通信プロトコルは主として (1) メッセージの形式と (2) メッセージ送受信の手順から規定されるが，通信プロトコルを実装するプログラマは，これに応じて，受信メッセージの解析や送信メッセージの組み立て，タイムアウトや非同期入出力といった処理を書く必要がある．これらはある程度，定型的な処理ではあるが，比較的煩雑なコーディングを必要とするため，結果としてプログラムの保守性を低下させたり，バグの原因となりやすい．本稿では，通信プロトコルの記述から自動的にプロトコル処理コードを生成する Preccs コンパイラについて説明する．Preccs ではメッセージの形式を正規表現を独自に拡張した記法によって，また，メッセージの送受信手順はプロセス代数に基づいた記法によって記述を行う．Preccs コンパイラはこれらの記述からプロトコルの処理を行う C のコードを自動的に生成する．これによってプログラマは煩雑なコーディングから解放され，信頼性や保守性に優れた通信プログラムを短期間で開発することが可能となる．

## 1 はじめに

インターネットでは様々な種類のサービスが提供されており，それに応じて多様な通信プロトコルが用いられている．たとえば，VPN サービスを実現するための IKE [2] や IPsec[7]，ストリーミング配信の基盤となる RTP[17] など，実に多くの通信プロトコルが存在する．近年ではインターネットの利用の広がりとともに，このように多様な通信プロトコルを実装する機会が多くなっている．実際，インターネット接続機能を持つ情報家電向けに TCP/IP プロトコルを実装したり，モバイルネットワークやセキュリティなどの新しいサービスに対応したプロトコルを，既存のプログラムに追加したりするなどといった要請が存在する．

通信プロトコルを実装する場合，プログラマは RFC などのプロトコル仕様が記述されたドキュメントを参照しながら，C 言語を用いてコーディングを行なうといった手順が一般的である．この場合，以下のような問題点がある．

1. RFC などのプロトコル仕様は図表や擬似コードを用いながら，基本的には自然言語によって記述されることがほとんである．したがって，どうしても細かい部分での曖昧さが残る．このため，プログラマの解釈の違いによって実装間で差異が生じることになる．また，逆にそのような事態を避けるため，参照実装[1]を解析しながら仕様の細部を確認するといった本末転倒な場面も生じる．

2. 最近では IKE に代表されるような複雑なプロトコルが増えてきている．そのように複雑なプロトコルを C で実装するのは困難な作業である．また，出来上がったプログラムは読み難く，バグも埋め込み易い．

3. そもそも，決まりきったプロトコルを実装するのは，プログラマにとって退屈な作業である．

このような背景から，我々は通信プトロコルのための仕様記述言語 Preccs を提案している [3]．

Preccs は通信プロトコルの仕様を簡潔に記述できるように設計されている．通信プロトコルは主とし

[1] インターネットで用いられているプロトコルは，参照実装と呼ばれる実装系が存在することが多い．これらは，ほとんどが C のプログラムである．

て (1) メッセージ形式と (2) メッセージの送受信手順の 2 点によって規定される [6] が，Preccs ではメッセージ形式を独自に拡張した正規表現によって記述し，送受信手順はプロセス代数に基づいた記法によって記述する．これらの特徴により，複雑なメッセージ形式を持つ通信プロトコルでも，正規表現によって宣言的に記述することが可能となり，また，プロトコルの状態遷移や非同期入出力の処理もプロセス代数に基づいて簡潔に記述することができる．さらに，Preccs コンパイラは，これらの記述から通信プロトコルを処理するための C のコードを自動的に生成する．これにより，プログラマは煩雑なコーディングから解放され，結果として信頼性や保守性に優れた通信プログラムを短期間で開発することが可能となる．

本稿の構成は以下のとおりである．第 2 節では，Preccs による通信プロトコルの記述方法について具体的な例を用いながら説明する．第 3 節で Preccs コンパイラの概要について説明し，第 4 節では簡単なプログラムを対象として，Preccs コンパイラの生成したコードの処理性能の測定結果を示し，オーバヘッドについて考察する．第 5 節で関連研究について述べた後，最後に本稿をまとめる．

## 2 仕様記述言語 Preccs

本節では，Preccs を用いた通信プロトコルの仕様記述方式について，簡単な例を交えながら説明する．

### 2.1 メッセージ形式の記述

Preccs では，メッセージ形式は正規表現のパタンを記述することによって定義する．ただし，純粋な正規表現ではなく，通信プロトコルで用いられるメッセージ形式を定義しやすいようにいくつかの工夫がなされている．

#### 2.1.1 メッセージ形式の記述例

例として，通信プロトコルによく見られるメッセージ形式を図 1 に示す．図 1 に示したメッセージは，1 オクテットと 4 オクテットの二つの固定長フィールドの後に 0 個以上のオプションが続き，最後は 0xFF で終わるという形式である．また，オプションも構造を持ち，オプション種別を表すフィールド tag とオプションのデータ長を示すフィールド len，さらにオプションデータのフィールド data からなる．これ

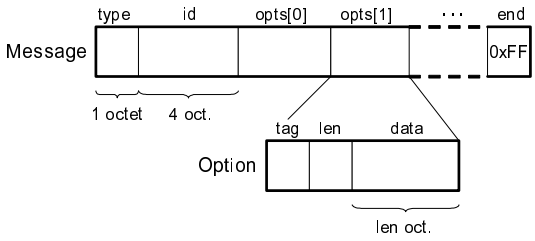


図 1: メッセージ形式の例

```
// メッセージ全体の構造定義
Message ::=
  type : octet,       // 1 オクテット長
  id   : octet[4],    // 4 オクテット長
  opts : Option*,     // オプションの並び
  end  : "FF"h;       // 0xFF で終端
// オプション構造の定義
Option ::=
  tag  : octet,
  len  : octet,
  data : octet[len]; // len オクテット長
```

図 2: Preccs によるメッセージの記述例

を Preccs で記述すると図 2 のようになる．図 2 において，`Message ::= …;` の部分でメッセージ形式 Message を定義している．Message の各フィールドにはラベル名を付与することができる．たとえば，`type`, `id` はラベル名であり，それぞれ `octet`, `octet[4]` というフィールドが対応している．フィールドの間を区切るコンマ (,) は，正規表現の連接に相当し，各フィールドが連続していることを意味している．`opts` ラベルで指定されるフィールド `Option*`はメッセージ形式 Option が 0 個以上連続することを意味しており，アスタリスク (*) は正規表現における閉包である．Preccs ではこの他にも選択 (|)，1 回以上の繰り返し (+)，0 回か 1 回のパタン (?) といった，正規表現の記法を使用することができる．

#### 2.1.2 ラベル参照による繰り返し

Preccs の正規表現の特徴的は，ラベルの示すフィールドの値を参照することによって可変長のデータが表現できることである．図 2 のメッセージ形式 Option の定義において，data フィールドの長さは直前の `len` ラベルで指定されたフィールドの値によって規定される．たとえば，len フィールドの値が 0x04 ならば，data フィールドの長さは 4 オクテットということに

```
// 要求メッセージの定義
RequestMsg ::= Message{type\"01"h};
// 応答メッセージの定義
ReplyMsg   ::= Message{type\"02"h};
```

図 3: 派生パタンを用いたメッセージの記述例

なる．このようなメッセージの構造は通信プロトコルでは一般的であり，Preccs ではそれを直接的に表現することが可能である．

#### 2.1.3 派生パタン

通信プロトコルで用いられるメッセージは，あるフィールドの値によってメッセージの種別が規定されることが多い．たとえば，図 1 の例で示した Message 中の type フィールドはメッセージの種別を表しており，このフィールドの値が 0x01 ならば要求メッセージ，0x02 の場合は応答メッセージを意味しているものとする．これを Preccs では図 3 のように記述することができる．

このように元々のメッセージ形式の一部のフィールドを置き換えることによって，新たなメッセージを定義することが可能である．これを派生パタンと呼ぶ．派生パタンによるフィールドの置き換えは，元々のメッセージ形式の範囲内でのみ許される．上記の例で言えば，type フィールドの元々のパタンは `octet` なので，置き換えができるのは `octet` に含まれるパタンのみである．たとえば，

```
Message{type\("01"h|"02"h)}
```

のような置き換えは可能だが，

```
Message{type\octet[4]}
Message{type\"0123"h}
```

のような置き換えは許されない．

派生パタンを用いることによって，通信プロトコルのメッセージが持つ自然な階層関係を簡潔に記述することができる．

### 2.2 送受信手順の定義

Preccs におけるメッセージ送受信手順の記述は R. Milner の CCS[14] や C.A.R.Hoare の CSP[5] に代表されるようなプロセス代数に基づいている．Preccs では，チャネルと呼ばれる仮想的な通信路に対して，同期的にデータの送受信を行う複数のプロセスを定義することによって，通信プロトコルの送受信手順を記述していく．

#### 2.2.1 送受信手順の記述例

ここでは，送受信手順の記述例として，標準入力から入力された文字列を echo サーバに送信し，受信した応答メッセージを標準出力に表示するという処理を，回数を数えながら繰り返すという，簡単な echo クライアントを取り上げる．図 4 は，echo クライアントの Preccs による記述例である．2 行目の `EchoClientProc(count:int) ::= …;` の部分が，クライアントの送受信手順の処理をプロセスとして定義しているところである．プロセスはパラメータを持つことができ，EchpClientProc プロセスは，現在の繰り返し回数を保持するための int 型のパラメータ `count` が与えられている．3 行目は標準入力から文字列を読み込み，その値を文字列型の変数 `key` に束縛するという動作を表している．この動作に続いて，4 行目で変数 `key` の内容をソケットに出力している．コンマ (,) 記号は逐次動作を意味しており，前の動作が完了した後に次の動作を実行する．

5 行目から 8 行目までは選択動作を表している．ソケットからメッセージを受信した場合には，標準出力に`"reply: "`という文字列とともに受信メッセージを出力する．また，5 秒間待ってもメッセージを受信できなかった場合にはタイムアウトし，標準出力に`"timeout\n"`を出力する．縦棒記号 (|) はこのような選択的な動作を表すために用いる．このように，あるメッセージの入力を待ちつつ，一定時間を過ぎた場合にはタイムアウトするという処理は，通信プロトコルを実装する際によく用いられる．このような処理を C で記述する場合には，一般に Unix の select システムコールや Win32 API の WaitForMultipleObjects などの比較的低レベルな API を用いた煩雑なコーディングが必要となる．Preccs では選択動作 (|) を用いることで，このような処理でも簡潔に記述することができる．

EchoClientProc プロセスは，これらの選択動作のいずれかを実行した後，9 行目で，`count` に 1 を足した値をパラメータとして新しいプロセスを再帰的に生成し，自身のプロセスは終了する．

```
1:  // echo クライアントの送受信手順の定義
2:  EchoClientProc(count:int) ::=
3:      stdin?key:string,
4:      sockout!key,
5:      (
6:          sockin?msg:string -> stdout!("reply: " + msg)
7:        | timer!5           -> stdout!("timeout\n")
8:      ),
9:      EchoClientProc(count+1);
```

図 4: echo クライアントの記述例

#### 2.2.2 チャネル入出力

プロセス同士は，チャネルを通じて同期的にデータを送受信することが可能である．先ほどの例に出てきた stdin や sockout は組み込みのチャネルであり，これらのチャネルは Preccs のプロセスが外部とのメッセージをやり取りするために用いられる．

プロセスの中で新しくチャネルを生成することもできる．たとえば，整数型のデータをやり取りするためのチャネル ich を生成するには，プロセス定義の中で new ich:<int>のように記述する．

すべてのチャネルは型を持っており，そのチャネルを通して送受信できるデータの型は制限される．標準入出力チャネルやソケット入出力チャネルは，文字列型のデータ[2]のみ送受信できる．

#### 2.2.3 タイムアウト処理

timer はタイムアウト処理を行うための特別な組み込みチャネルで，プロセスは timer チャネルに渡された秒数だけブロックされる．たとえば，timer!10 とすると，プロセスは 10 秒間ブロックされる．これによってタイムアウトやスリープといった処理を実現することができる．

#### 2.2.4 パタンマッチ

あるデータに対して，そのデータの持つパタン種類によって動作を振り分けたい場合にはパタンマッチ構文 (match〜with) を使用する．たとえば，先ほどの echo クライアントに機能を追加して，標準入力から"quit\n"と入力されたらプログラムを終了させることを考える．この場合，図 4 の 3, 4 行目を以下のように変更すればよい．

```
stdin?key:string,
( match key with
    "quit\n" -> stop
  | string   -> skip
),
sockout!key,
```

ここでは，変数 key が保持しているデータのパタンに応じて動作の振り分けを行っている．パタンマッチングは先頭から順に試みられ，最初にマッチしたパタンに対応する動作が実行される．したがって，上記の例では，変数 key は，まず"quit\n"とのマッチングが試みられる．このマッチングに失敗した場合には，続いて string パタンとマッチング[3]が行われる．ここで stop はプロセスの終了動作を，skip は後続の処理の実行を意味する予約語である．正規表現のパタンとして記述したメッセージ形式の定義をパタンマッチングの機構で用いることによって，受信メッセージの種類による動作の振り分けを簡単に記述することが可能となる．

#### 2.2.5 メッセージ生成

通信プロトコルでは，ネットワーク上に送信するメッセージを組み立てるという処理が必要である．Preccs のプロセス中で新しくメッセージを生成することが可能で，そのメッセージはデフォルトのデータで初期化される．たとえば，図 3 で定義した RequstMsg 形式のメッセージを生成したい場合には，プロセス中で new msg:RequestMsg と記述すればよい．新しく RequestMsg 形式のメッセージが生成され，msg 変数に束縛される．ここで生成されるメッセージはデフォルトのデータで初期化されている．octet のデフォルト値は 0x00 であり，閉包パタン (*) のデフォルト値は $\epsilon$ というように，各基本パタンに対応するデフォルト値が決められており，全体のデフォルトデータ

[2] Preccs における文字列型は octet*パタンと同義である．

[3] 変数 key は文字列型なので，このマッチングは必ず成功する．

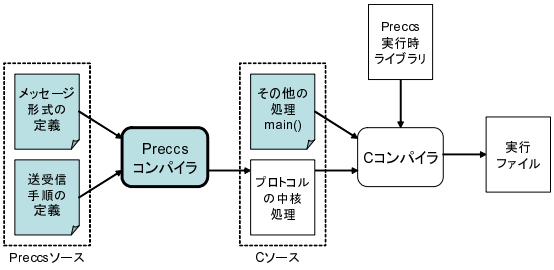


図 5: Preccs コンパイラを用いた開発の流れ

はそれらから導かれる．たとえば，RequestMsg のデフォルトデータは 0x0100000000FF となる．

## 3 Preccs 処理系

本節では，Preccs 処理系の概要について説明する．

### 3.1 全体の構成

図 5 に，Preccs コンパイラによるプログラム作成の流れを示す．プログラマは RFC などのプロトコル仕様に基づいて，メッセージ形式や送受信手順を Preccs のソースとして記述する．Preccs コンパイラは Preccs ソースから，パタンマッチに用いる状態遷移表やプロセス動作を実現するための処理を C のコードとして出力する．ここで出力されたコードは Preccs 実行時ライブラリと協調しながら，プロトコルの中核的な処理をインタプリティブに実行するものである．Preccs 実行時ライブラリは，パタンマッチやチャネル通信，プロセス管理などを実現するためのルーチンから構成される．

その他，Preccs で記述するのが適当でない雑多な処理，たとえば，コンフィグレーションの取得やログ出力，OS に関する情報の取得などはプログラマが直接 C でプログラムを記述する．最終的には，これらのコードと実行時ライブラリから，C コンパイラによって実行ファイルが生成される．C のコードを生成することによって可搬性が高くなり，通信アプリケーションのほかに，組み込み系の通信プログラム，さらには OS のカーネルレベルで実装されているプロトコルスタックなど，様々なプラットフォームに柔軟に対応することができる．

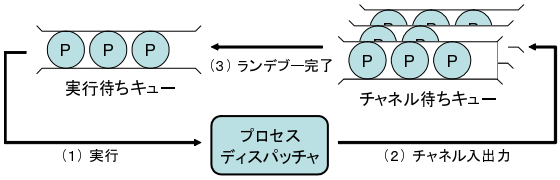


図 6: Preccs のプロセス実行モデル

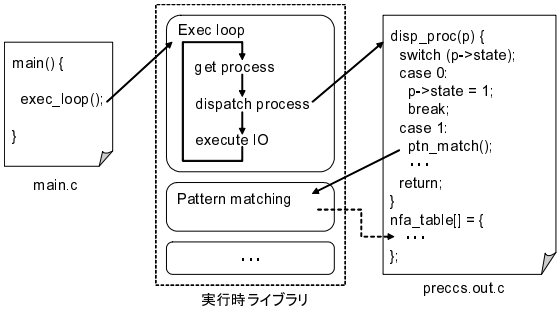


図 7: Preccs の実行時システムの構成

### 3.2 実行時システム

図 6 は，Preccs におけるプロセスの実行の様子を示したものである．実行待ちキューから取り出されたプロセスはディスパッチャによって実行される．実行中のプロセスがチャネル入出力処理を要求すると，プロセスは該当するチャネル待ちキューに置かれる．データ入出力が完了したプロセスは，再び実行可能状態とり，実行待ちキューに置かれる．

図 7 は Preccs 実行時における処理の流れを示したものである．Preccs の処理は，`exec_loop()` の呼び出しによって開始される．`exec_loop()` は実行時ライブラリ内の関数であり，プロセス処理のループを実現している．具体的には，実行待ちキューからプロセスを取得し，プロセスのディスパッチと標準入出力やソケットなどの IO 処理を繰り返す．preccs.out.c は，Preccs コンパイラによって生成されたコードであり，パタンマッチのための状態遷移表 (`nfa_table[]`) やプロセス実行のためのディスパッチャが含まれている．

以下では Preccs 実行時ライブラリが扱う主な処理について説明していく．

#### 3.2.1 プロセスの生成と削除

プロセスを作成 (`make_proc()`) すると，プロセスオブジェクトがヒープ領域に確保され，実行待ちキューに置かれる．プロセスは実行が終わると削除

(`free_proc()`) され，ヒープ領域に戻される．

#### 3.2.2 チャネル入出力

Preccs が扱うチャネルは，内部チャネル，IO チャネル，タイマーチャネルの 3 種類に分類される．内部チャネルは，Preccs 内部のプロセス同士でデータを送受信するためのものである．IO チャネルは組み込みの標準入出力チャネル (`stdin`, `stdout`)，ソケット入出力チャネル (`sockin`, `sockout`) である．タイマーチャネル (`timer`) はタイムアウトを処理するための特別なチャネルである．

プロセスがチャネルに対して入出力処理を要求 (`chan_in()`, `chan_out()`) すると，そのチャネルで待っているプロセスや（IO チャネルの場合には）IO 状態を調べ，可能ならばデータ送受信処理（ランデブー）を行い，これらのプロセスを実行待ちキューに置く．そうでなければ，入出力要求を行ったプロセスをチャネル待ちキューに置く．タイマーチャネルの場合には，チャネルに渡された秒数だけタイマーをセットし，プロセスをタイマーチャネルの待ちキューに置く．タイマーが切れたプロセスは取り出されて，再び実行待ちキューに置かれる．

#### 3.2.3 パタンマッチング

Preccs では正規表現によるメッセージ形式の定義に基づいて，メッセージの解析（パタンマッチング）を行う．Preccs では，非決定性有限オートマトン（NFA）をバックトラッキングによってシミュレートすることでパタンマッチングを行う．ただし，Preccs の正規表現は「ラベル参照による繰り返し」という独自の拡張を行っているので，これを実現するために，NFA はカウンタ用のスタックやラベル管理のためのテーブルを備えている．

### 3.3 コード生成

Preccs コンパイラは，メッセージ形式の記述部分から NFA が参照するための状態遷移表を生成する．また，送受信手順の記述部分からは，プロセスのディスパッチコードが生成される．ディスパッチコードは一つの巨大な switch〜case 文として実現され，各 case 文単位でプロセスの切り替えが行われる．

Preccs の各プロセスは，図 8 に示すように複数の case 文を含んだコードに翻訳される．このコードは図 4 の Preccs ソースから実際に Preccs コンパイラによって生成されたものを一部抜粋したものである．

## 4 実験

Preccs による処理のオーバヘッドを調べるために，いくつかの簡単なプログラムに対して，実行時間の測定を行った．本節ではその結果について報告する．

### 4.1 Fibonacci プロセス

最初の例として，Fibonacci 数の計算を Preccs のプロセスとして記述し，実行時間の測定を行った．Preccs によるプロセスの定義を図 9 に示す．FibProc プロセスは，n 番目の Fibonacci 数を計算する再帰的なプロセスであり，整数値 n と結果を返すためのチャネル out の 2 つの引数を取る．FibProc プロセスは，引数 n の値が 0 か 1 の場合には，チャネル out に対応する値を出力して終了する．それ以外の場合は，新しく 2 つのチャネルを生成し，n-1 と n-2 の値とともにそれぞれ FibProc プロセスを生成する．その後，生成したチャネルからの入力を待ち，その結果を足し合わせて out チャネルに出力する．

Preccs による処理コードと C による実行時間の比較結果を表 1 に示す．測定に用いた計算機は Intel Centrino 1.1GHz プロセッサ，760MB の主メモリを搭載した PC/AT 互換機であり，OS は Windows XP である．比較に用いた C のプログラムは Fibonacci 数の計算を再帰的な関数によって実現したものである．測定結果を見ると，プロセスの実行は C の関数呼び出しに比べて 70 倍〜120 倍程度のオーバヘッドが生じることが見てとれる．現在の実装では，プロセスやチャネルを生成するたびに malloc の呼び出しを行っているため，C の関数呼び出しと比較してその処理はかなり重いものとなっている．

### 4.2 SNTP プロトコル

次に実際の通信プロトコル処理にかかるオーバヘッドを調べるために，SNTP クライアントを実装し，実行時間の測定を行った．SNTP は計算機の時刻合わせのために用いるプロトコル [13] であり，NTP サーバにアクセスして，精度の高い時刻情報を得ることができる．

#### 4.2.1 SNTP クライアントの動作

図 10 に SNTP クライアントの動作を Preccs のプロセスとして定義したものを示す．メッセージ形式の

```
 1:  case 2: /* EchoClientProc */
 2:      chan_in(__chan_stdin, proc, 10, (void **)&proc->data[1]);
 3:      break;
 4:  case 10:
 5:      chan_out(__chan_sockout, proc, 9, (void *)proc->data[1]);
 6:      break;
 7:  case 9:
 8:      proc->state = 4; goto next;
 9:      break;
10:  case 5:
11:      chan_out(__chan_stdout, proc, 8,
                              (void *)value_append(&__val_3d8c78, proc->data[2]));
12:      break;
13:  case 8:
14:      proc->state = 3; goto next;
15:      break;
16:  case 6:
17:      chan_out(__chan_stdout, proc, 7, (void *)&__val_3d8e58);
18:      break;
19:  case 7:
20:      proc->state = 3; goto next;
21:      break;
22:  case 4:
23:      chan_in(__chan_sockin, proc, 5, (void **)&proc->data[2]);
24:      chan_out(__chan_timer, proc, 6, (void *)5);
25:      break;
26:  case 3:
27:  {
28:      proc_t *np = make_proc(2, 3);
29:      np->data[0] = (void *)((int)proc->data[0] + (int)1);
30:      proc_setrdy(np, 2);
31:  }
32:      free_proc(proc);
33:      break;
```

図 8: Preccs の生成コード

定義は省略するが，SNTP プロトコルで用いるメッセージは，いくつかの固定長のフィールドの並びの後に，最後にオプションで終わるという単純なものであり，Preccs によって簡潔に定義することが可能である．

図 10 の 1,2 行目は Main プロセスを定義している．Preccs の処理が開始されると Main プロセスが一つ生成され，そこから処理が続いていく．ここでは，Main プロセスの処理は単に SntpInitProc プロセスを生成するだけである．

4〜12 行目で定義されている SntpInitProc プロセスは標準入力からの入力を待ち，"quit"と入力されたら終了し，それ以外なら処理を続行する．10 行目では，SntpSndPkt として定義されている送信用メッセージを生成し，11 行目でそれをソケット出力チャネルへ出力している．これによって，メッセージがネットワーク上に送出される．最後に SntpWaitProc プロセスを生成して，返信待ち状態に移る[4]．

14〜25 行目が SntpWaitProc プロセスの定義である．ここでは，サーバからメッセージの返信を待っている（17 行目）が，タイムアウトしたり，標準入力があった場合にはメッセージの受信をあきらめて，初期状態（SntpProcInit）に戻る．メッセージを受信した場合には，パタンマッチを行い，SntpRcvPkt として定義されているメッセージ形式とマッチすれば正しいメッセージとみなし，時刻を表示する．パタンマッチに失敗したら（22 行目）エラーメッセージを出力を行う．その後，プロセスは SntpProcInit

[4]実際には SntpInitProc プロセスは終了し，削除されるが，プログラム全体としては返信待ち状態に移ったとみなせる．

```
1:  FibProc(n:int, out:<int>) ::=
2:       n = 0 -> out!0
3:     | n = 1 -> out!1
4:     | true  ->
5:           new ch1:<int>, new ch2:<int>,
6:           FibProc(n-1, ch1), FibProc(n-2, ch2),
7:           ch1?r1:int, ch2?r2:int,
8:           out!r1+r2;
```

図 9: Fibonacci プロセスの記述

表 1: Fibonacci の実行時間測定

| 問題サイズ (n) | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|
| (a) Preccs (msec) | 2387.4 | 3903.8 | 6401.4 | 10779.6 | 17963.4 |
| (b) C | 33.8 | 45.8 | 66.2 | 98.0 | 151.4 |
| 実行時間比 (a/b) | 70.6 | 85.2 | 96.69 | 110.0 | 118.6 |

プロセスを生成し，初期状態に戻る．19〜21 行目の C{ 〜 C}の部分は，C 言語のコードを埋め込んでおり，その中で時刻を表示する C の print_ntp_time() 関数[5]を呼び出している．このように Preccs ではインラインで C のコードを記述できるようになっている．$rpkt$は，インラインコード中からの rpkt 変数の参照を示している．

#### 4.2.2 実行時間の比較

Preccs による SNTP クライアントの記述と，それに相当する処理を C でスクラッチで記述したものとで実行時間の比較を行った．NTP サーバは ntp.nc.u-tokyo.ac.jp を利用し，このサーバに対してインターネット経由でアクセスし，時刻を取得するまでの実行時間を計測した．また，参考のために実行時間中に含まれるメッセージの往復遅延時間（RTT）も同時に計測した．それぞれ 10 回の平均値である．クライアントの実行に用いた計算機は Fibonacci プロセスの測定に用いたものと同一である．結果は表 2 のとおりである．これを見ると，Preccs と C での実行時間の差はごくわずかなものに収まっている．これは，クライアント側の実行時間のほとんどは，ネットワークによる通信遅延によるものだからである．さらに，SNTP クライアントでは Fibonacci プロセスのように頻繁にプロセスの生成，削除を行っているわけではないので，プロセス生成に関するオーバヘッドは無視できる．これらの事実から，実際的な通信プロトコルのクライアント実装においては，Preccs による処理のオーバヘッドは問題にならない程度であることが言える．

[5]プログラマが C の関数として定義済みとする．

## 5 関連研究

通信プロトコルのための仕様記述言語としては，一般に LOTOS や Estelle，SDL といった言語が知られており [15]，これらは主としてプロトコルの正しさを自動的に検証する目的で用いられてきた．また，これらの言語による仕様記述から実用的なコードを生成する処理系の開発もいくつか報告されている [12, 1, 11]．しかしながら，これらの言語は幅広い階層の通信プロトコルを対象として規定されたものであるため，その言語仕様は複雑である．さらにプロトコルの記述には形式的仕様記述に関する専門的な知識が必要となる．このため，一般的なプログラマがこれらの言語を用いて通信プロトコルを記述したり，また，その記述から通信プロトコルの仕様を理解するのは困難である．実際，これらの言語が広く通信プロトコルの実装に利用されているとは言い難い．

商用の通信プロトコル開発ツールとしては米国 Novilit 社が開発した AnyWare[4] がある．AnyWare では，プログラマは CMDL(Communication Machine Definition Language) と呼ばれる専用の言語で通信プロトコルの仕様を記述する．AnyWare は CMDL で書かれたプロトコルの記述から C/C++や Java，VHDL など，様々なコードを生成することが可能で

```
 1:  Main() ::=                // Main プロセス
 2:      SntpInitProc();
 3:
 4:  SntpInitProc() ::=        // 初期状態
 5:      stdin?cmnd:string,
 6:      ( match cmnd with
 7:          "quit\n" -> stop
 8:        | string   -> skip
 9:      ),
10:      new spkt:SntpSndPkt,
11:      sockout!spkt,
12:      SntpWaitProc();
13:
14:  SntpWaitProc() ::=       // 返信待ち状態
15:      (   timer!10          -> stdout!"Timeout.\n"
16:        | stdin?string      -> skip
17:        | sockin?msg:string ->
18:          ( match msg with
19:              rpkt:SntpRcvPacket -> C{
20:                  print_ntp_time($rpkt$);
21:              C}
22:            | string     -> stdout!"Illegal packet received.\n"
23:          )
24:      ),
25:      SntpInitProc();
```

図 10: SNTP 送受信手順の記述例

表 2: SNTP クライアントの実行時間測定

| | Preccs | C | RTT |
|---|---|---|---|
| 実行時間 (msec) | 2.50 | 2.46 | 2.34 |

ある．しかしながら，CMDL は通信プロトコルの振る舞いに対応したステートマシンを直接プログラマが記述する必要があるため，記述の容易性，可読性といった点からは Preccs の方が有利である．

Prolac[8] は MIT LCS の PDOS グループが開発したプロトコル記述言語で，言語仕様としてみると静的に型付けされたオブジェクト指向言語に分類される．Prolac コンパイラも Preccs と同様に C 言語のコードを生成する．また，生成したコードの性能も高く，文献 [9] によれば，Prolac による TCP 実装は Linux 2.0 の TCP 実装と同等の性能が得られたことを報告している．ただし，Prolac では Preccs のように正規表現に基づいたメッセージ形式を定義することはできないので，IKE のような複雑なメッセージ形式を持ったプロトコルを記述する場合には，Preccs の方がより適していると考えられる．

河野 [10] はクライアント・サーバ型のアプリケーション層プロトコルの実現を支援する April フレームワークを提案し，April によって実際に POP，HTTP，SMTP などのプロトコルを記述した結果を報告している．April では正規表現を用いてメッセージ形式の記述に行い，そこから受信メッセージを解析するコードを自動生成するなど，我々の提案する手法と共通する部分も見られる．ただし，April はアプリケーション層プロトコルを対象としており，送受信されるメッセージは文字列を前提としている．一方，Preccs ではデータリンク層からトランスポート層までのより低い階層のプロトコルを主たる対象としている．これらのプロトコルで扱うメッセージは一般にバイナリ形式であり，April の手法をそのまま適用することはできない．

## 6 まとめ

本稿では，Preccs による通信プロトコルの記述方法について述べ，これらの記述から C のコードを自動的に生成する Preccs コンパイラについて説明した．また，実際の通信プロトコルの例として，簡単な SNTP クライアントの動作を Preccs によって記述し，C 言語による実装との性能比較を行った．この結果，通信プロトコルの処理では，自動生成された処理コードによるオーバヘッドは問題にならない程度に小さいことが確認できた．これはメッセージの送受信による遅延時間がクライアントの実行時間の大半を占めるからである．ただし，Fibonacci プロセスの実験結果からわかるとおり，ネットワークによる通信遅延を含まない処理では，Preccs によるオーバヘッドはかなり大きい．このことは，サーバ側の処理を Preccs で実装する場合には，問題になってくると予想される．より広い範囲で Preccs を適用するためには，生成コードの最適化が必要である．文献 [16] では，プロセス切り替えやチャネル入出力をコンパイル時に静的に解析することによって，より効率の良いコード生成の手法が示されているが，これらの手法は Preccs にも適用可能であろう．

Preccs の有効性を示すには，今後，より複雑なプロトコルに対して Preccs を適用し，記述性や実行性能の評価を行っていくことが必要がある．また，現在の Preccs では，型システムに関する検討が不十分である．Preccs ではメッセージ形式を型とみなして，一応の型検査を実装しているが，整合性や一貫性など十分な検証はなされていない．これらを整理していく必要がある．

### 謝辞

Preccs の開発プロジェクトは，IPA（情報処理推進機構）の公募事業 2004 年度第 2 回未踏ソフトウェア創造事業（伊知地 PM）に採択され，支援を受けている．

### 参考文献

[1] Fischer, S. and Hofmann, B.: An Estelle Compiler for Multiprocessor Platforms, *FORTE '93: Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques, VI*, North-Holland, 1994, pp. 171–186.

[2] Harkins, D. and Carrel, D.: The Internet Key Exchange (IKE), 1998. (RFC2409).

[3] 服部健太, 数馬洋一: 正規表現とプロセス代数に基く通信プロトコルのための仕様記述言語の提案, 2005. 情報処理学会プログラミング研究会発表資料.

[4] 日下野友彦: プロトコル仕様記述言語 CMDL による通信プロトコル設計, *Interface, CQ* **出版社**, Vol. 30,No. 2(2004), pp. 155–163.

[5] Hoare, C.: *Communicating Sequential Process*, Prentice Hall, Reading, Massachusetts, 1985. (吉田信博訳: ホーア CSP モデルの理論, 丸善株式会社 (1992)).

[6] 笠野英松 (監修): **通信プロトコル事典**, アスキー, 1996.

[7] Kent, S. and Atkinson, R.: Security Architecture for the Internet Protocol, 1998. (RFC2401).

[8] Kohler, E.: Prolac: A Language Protocol Compilation, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Insutitute of Technology, 1997.

[9] Kohler, E., Kaashoek, M. F., and Montgomery, D. R.: A Readable TCP in the Prolac Protocol Language, *ACM SIGCOMM'99*, Vol. 29(1999), pp. 3–13.

[10] 河野健二: アプリケーション層プロトコルの実現を容易にするフレームワーク, **情報処理学会論文誌：プログラミング**, Vol. 44,No. SIG 2(PRO 16)(2003), pp. 25–35.

[11] Leue, S. and Oechslin, P. A.: On Parallelizing and Optimizing the Implimentation of Communication Protocols, *IEEE/ACM Trans. Networking*, Vol. 4,No. 1(1996), pp. 55–70.

[12] Mañas, J. A. and de Miguel, T.: From LOTOS to C, *Proceedings of the First International Conference on Formal Description Techniques*, North-Holland, 1989, pp. 79–84.

[13] Mills, D.: Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, 1996. (RFC2030).

[14] Milner, R.: *Communication and Concurrency*, Prentice Hall, 1989.

[15] 水野忠則 (監修): **プロトコル言語**, カットシステム, 1994.

[16] Oyama, Y., Taura, K., and Yonezawa, A.: An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus, *Euro-Par '97, Parallel Processing*, 1997, pp. 546–553.

[17] Schulzrinne, H., Casner, S., Frederick, R., et al.: RTP: A Transport Protocol for Real-Time Applications, 1996. (RFC1889).