正規表現とプロセス代数に基づく通信プロトコルのための仕様記述言語の提案

服部健太[†,††] 数馬洋一[†]

本稿では，通信プロトコルを簡単に記述するための仕様記述言語 Preccs を提案し，Preccs で書かれた仕様から C のプログラムを自動的に生成する方法について説明する．Preccs は正規表現とプロセス代数のアイディアに基づいており，通信プロトコルを分かりやすく記述することができるように工夫されている．通信プロトコルは主に (1) メッセージの形式と (2) メッセージ送受信の手順から構成されており，Preccs によるプロトコルの仕様記述はこれと素直に対応している．Preccs では，(1) は正規表現を独自に拡張した記法によって記述する．これによって複雑な構造を持ったメッセージでも簡単に記述することができる．また (2) は CCS などに代表されるプロセス代数にヒントを得てプロトコルの状態遷移や通信シーケンスを簡便に記述できるようにしたものである．Preccs を用いて通信プロトコルの仕様を厳密に記述し，そこからプログラムを自動生成することによって，信頼性の高い通信プログラムを短期間で開発することが可能となる．また，Preccs で書かれた仕様は読みやすく，プログラムの機能拡張や保守などが容易になることが期待できる．

A Formal Specification Language for Communication Protocols Based on Regular Expressions and Process Algebra

Kenta Hattori[†,††] and Youichi Kazuma[†]

We propose a simple formal specification language Preccs for communication protocols, and explain a method of translation from the Preccs source code to the C language. Preccs is based on concepts of regular expressions and process algebra. This language has been designed for a simple and intuitive description of communication protocols. A communication protocol generally comprises definitions of message formats and rules for message sequences. Thus, a protocol description via Preccs also comprises two parts: (1) definitions of message formats with extended regular expression that can simplify descriptions of complicated structured messages; (2) definitions of rules for message sequences with notation based on process algebra such as CCS, which is purposed to directly describe state transitions or protocol sequences. The generation of C programs with a Preccs compiler will enable the development of reliable communication programs in a short span of time. Moreover, because of the readability of this language, it is expected that a programmer will be able to easily add new functions and maintain programs.

1. はじめに

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，様々な場面で通信プロトコルを実装する機会が多くなってきた．たとえば，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機能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といったことが考えられる．

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 を用いてコーディングを行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どである．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

† 株式会社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, The University of Tokyo

うな事態を避けるため，参照実装☆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[4] 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，できあがったプログラムは読み難く，バグも埋め込みやすい．

(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．

そこで，本稿では通信プトロコルのための仕様記述言語 Preccs☆☆を提案し，Preccs によるプロトコル仕様記述から C のコードを自動生成する方式について説明する．

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[16]，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ために用いられてきた．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[3),12),13)]．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したり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いい難い．我々の提案する Preccs は，プログラマが通信プロトコルを簡単に記述することができ，さらに，記述したコードの可読性も高くなるように設計されている．また，Preccs による記述から C のコードを自動的に生成することも可能である．これによって，信頼性の高い通信プログラムを短期間で開発することができ，同時に保守性や拡張性も得られることが期待できる．

本稿の構成は以下のとおりである．2 章では，Preccs による通信プロトコルの記述方法について述べる．3 章では，Preccs 処理系の構成や実行モデル，実装方法を述べる．4 章では Preccs が適用対象とする通信プロトコルや関連研究について述べる．

☆ 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．

☆☆ Protocol Code Generator based on Regular Expression and CCS.

2. Preccs による仕様記述

本稿で提案する Preccs は通信プロトコルを素直に記述できるように工夫されている．通信プロトコルは主として (1) メッセージの形式と (2) メッセージの送受信手順の 2 点によって規定される[8)] が，Preccs によるプロトコルの仕様記述は，この事実と素直に対応した構成になっている．

本章では，Preccs によるメッセージ形式と送受信手順の記述方式について，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 によるメッセージ形式の記述は正規表現に基づいている．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 典型的なメッセージの例

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，2 オクテットと 4 オクテットの 2 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，種別を表すフィールド kind 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，`fld1`，`fld2` はラベル名であり，それぞれ `octet[2]`，`octet[4]` というフィールドが対応している．各フィールドを区切るコンマ（,）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 個以上連続するようなフィールドを意味しており，スター（*）は，正規表現における閉包演算子である．

Preccs で特徴的なことは，ラベルにマッチした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

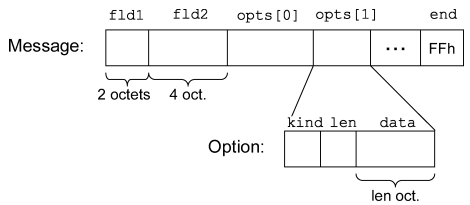

図 1 メッセージ形式の例
Fig. 1 An example of message format.

```
// メッセージ全体の構造定義
Message ::=
  fld1 : octet[2],  // fld1 は 2 オクテット長
  fld2 : octet[4],  // fld2 は 4 オクテット長
  opts : Option*,   // オプションが 0 個以上
  end  : "FF"h      // 0xFF で終端
;
// オプション構造の定義
Option ::=
  kind : octet,     // オプションの種別
  len  : octet,     // オプションのデータ長
  data : octet[len] // len オクテット分の長さ
;
```

図 2 Preccs によるメッセージの記述例
Fig. 2 An example of message description.

M ::= ID	定義済みパタン
| octet | *literal*	基本パタン
| M_1,M_2	連接
| M_1|M_2	選択
| M*	閉包
| l:M	ラベル定義
| M[n] | M[l]	繰り返し

図 3 メッセージ形式記述の構文
Fig. 3 Syntax of message description.

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とマッチした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．

2.1.2 メッセージ形式の記述方法

ここでは，Preccs におけるメッセージ形式の記述方法について詳しく説明していく．まず，メッセージ形式記述の構文の定義を**図 3** に示す．メッセージ形式は以下のように定義する．

ID ::= M;

ID は，メッセージ形式を定義する一意な名前である．ここで定義したメッセージ形式の識別子 ID は，後にメッセージの処理手順を記述する際に，型名として扱われる．各構文の意味は以下のとおりである．

定義済みパターン 定義したメッセージ形式を参照し，そのパターンとマッチする．

基本パターン octet は，1 オクテット長のデータとマッチする．*literal* は "*ascii-string*" か "*hex-string*"h の形式であり，それぞれ，指定された ASCII 文字列か 16 進文字列に対応するデータとマッチする．

連接 正規表現における連接と同様である．

選択 正規表現における選択と同様である．

閉包 正規表現における閉包と同様である．

ラベル定義 パターン M にマッチしたデータに対して，ラベル l を付与する．ラベル定義は，マッチングには直接的な影響を与えないが，後述する繰返し回数の指定の際に，定義済みのラベルを参照することができる．ラベルは後方参照のみ可能である．また，ラベルのスコープは閉包と選択の中で閉じており，その外側では参照することはできない．たとえば，$(l{:}M_1)*,M_2$ の場合，ラベル l を M_2 の中で参照することはできない．同様に $((l{:}M_1)|M_2),M_3$ の場合，ラベル l は M_2，M_3 の中で参照することはできない．$((l{:}M_1),M_2)*,M_3$ の場合，ラベル l は，M_2 の中で参照可能だが，M_3 では参照できない．このような制限を加えることによって，ラベルを参照した際に対応するデータがマッチ済みであることが保障される．

繰り返し パターン M を n 回，または l で参照される値分だけ繰り返したものとマッチする．$n \geq 0$ である．また，l で参照される値とは，定義済みのラベル l で指定されるフィールドとマッチしたデータを，ネットワークバイトオーダで表現された正整数と見なしたときの値である．

2.2 メッセージ処理手順の記述

Preccs におけるメッセージ処理手順の記述は Milner の CCS [14] や Hoare の CSP [6] に代表されるようなプロセス代数に基づいている．Preccs で定義される各プロセスはチャネルを通じて同期的にメッセージのやりとりを行う．

2.2.1 3 ウェイハンドシェークの記述例

はじめに，一般的な通信コネクション確立の手法である 3 ウェイハンドシェイクを例にして，メッセージ処理手順の記述方法を説明する．**図 4** は，3 ウェイハンドシェイクによるサーバ側におけるコネクション確立の状態遷移を示したものである．サーバ側の動作は次のようになる．初期状態 Init で Syn メッセージを受信すると，SynAck メッセージを送信した後に，Wait 状態に移行する．Wait 状態で Ack メッセージを受信するとセッション確立し，Established 状態に移行するが，Rst メッセージを受信したり，一定時間以内にメッセージを受信しないとタイムアウトし，初期状態 Init に戻る．Established 状態で Rst メッセージを受信するとセッションは終了し，再び初期状態 Init に戻る．

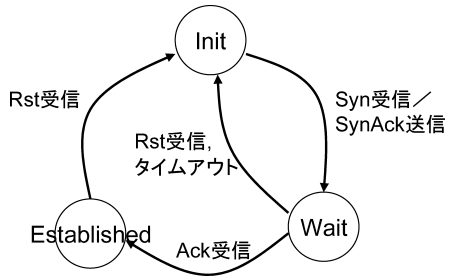

図 4 サーバ側の状態遷移の例
Fig. 4 An example of server side state transition.

```
// 初期状態
Init() ::=
    sock?rpkt:Syn,
    new spkt:SynAck, sock!spkt,
    Wait();
// 待ち状態
Wait() ::=
    sock?rpkt:Ack, Established()
  | sock?rpkt:Rst, Init()
  | timeout(10), Init();
// 確立状態
Established() ::=
    sock?rpkt:Rst, Init();
```

図 5 Preccs による送受信手順の記述例
Fig. 5 An example of process description.

図 4 に示したような 3 ウェイハンドシェイクのサーバ側の動作を Preccs で記述すると**図 5** のようになる．図 5 の `Init() ::= …;` では Init プロセスを定義している．Init プロセスは `sock` チャネルから `Syn` メッセージを受信し，`rpkt` 変数に代入する．次に，`SynAck` メッセージを新規に生成し，`spkt` 変数に代入した後，それを `sock` チャネルから送信し，最後に Wait プロセスを呼び出し終了する．

Wait プロセスは，`sock` チャネルからの受信メッセージの種類やタイムアウトによる非決定的選択動作を行う．これは縦棒（|）によって表現する☆．`sock` チャネルから `Ack` メッセージを受信した場合は，Established プロセスを呼び出す．また，`Rst` メッセージを受信したり，`timeout(10)` とあるように，10 秒間待ってもメッセージが到着しなかった場合には，Init プロセスを呼び出し，初期状態に移行する．このようにプロセス代数における選択動作を扱えるようにすることで，送受信の動作を自然に表現することが可能となる．

Established プロセスは，`sock` チャネルから `Rst` メッセージを受信すると，Init プロセスを呼び出し，終了する．

ここで定義した各プロセスはサーバの各状態に相当する．いい換えると状態遷移の各状態は Preccs のプロセスとして表現することができる．

☆ CCS における選択動作は “+” によって表現されるが，算術演算子と紛らわしいので，“|” を採用した．

$P ::= G$	ガードプロセス
$\mid$ `new` var : $type$	変数生成
$\mid P_1, P_2$	逐次動作
$\mid$ [c_1] P_1 [c_2] P_2 …[c_n] P_n	条件分岐
$\mid proc(v_1, v_2, \cdots, v_n)$	プロセス呼び出し
$\mid$ {...}	C インライン
$G ::= G, P$	逐次実行
$\mid G_1 \mid G_2$	選択実行
$\mid ch$? var : $type$	入力動作
$\mid ch$! val	出力動作
$\mid$ `timeout`(n)	タイムアウト

図 6 プロセス動作式の構文
Fig. 6 Syntax of process expression.

2.2.2 プロセス定義の方法

Preccs では，プロセスの動作を定義することによって，メッセージ処理手順を記述していく．ここでは，プロセス定義の方法について説明する．Preccs におけるプロセス定義の形式は次のとおりである．

$$p\text{-}name(v_1 : t_1, \ldots, v_n : t_n) \texttt{ ::= } P;$$

ここで，$p\text{-}name$ は定義するプロセスの名前であり，$v_1, \ldots, v_n$ はプロセスのパラメータである．パラメータにはそれぞれ型 $t_1, \ldots, t_n$ を指定する必要がある．P は，**図 6** に示す構文で定義されるプロセス動作式である．型の種類は，組み込みの型である整数型（`integer`）や文字列型（`string`），真偽型（`boolean`），チャネル型（`channel`）などに加えて，メッセージ形式の定義部で定義したメッセージ形式の識別子が含まれる．

各構文に対応するプロセスの動作を以下に記述する．

変数生成 型 $type$ で指定される変数 var を新しく生成する．変数は，後続の逐次動作の中で参照することができる．

逐次実行 P_1 の動作が完了した後，P_2 の動作を開始する．

選択実行 後述するチャネル入出力動作やタイムアウトで始まるプロセスをガードプロセスと呼ぶ．すなわち，ガードプロセスとは実行を開始すると最初に待ち状態になるプロセスのことである．選択実行で指定できるプロセスはこのガードプロセスのみである．G_1 と G_2 のどちらかのガードプロセスが実行可能となったとき，そのプロセスを実行し，一方のプロセスを破棄する．

条件分岐 条件式 $c_1, \ldots, c_n$ を順に評価していき，最初に真となった条件式 c_i に対応するプロセス P_i

を実行する．

プロセス呼び出し 引数 $v_1, \ldots, v_n$ を評価し，それらをプロセスの初期パラメータとして新しくプロセスを生成する．生成されたプロセスは，呼び出し元のプロセスと並行に動作を開始する．

入力動作 チャネル ch から型 $type$ であるような値の入力を待つ．別のプロセスが ch に対して値を出力すると，新しく変数 var を生成し，その値を代入する．

出力動作 値 val を評価し，チャネル ch へその結果を出力する．Preccs では，チャネル上の送受信は同期的に実行される．したがって，ch で入力を待つプロセスが存在しない場合は，この出力動作はブロックされる．

タイムアウト n 秒間待った後，動作を再開する．

C インライン {} 内に記述した C のコードを実行する．C のコード中から，プロセス内で定義された変数を参照することも可能である．

Preccs では，並行動作を直接的に表現する構文は用意していないが，プロセス呼び出しによって複数のプロセスを並行して動作させることが可能である．

2.2.3 予約済みチャネル

`sock`，`stdin`，`stdout` は予約済みのチャネルで，Preccs システムの外の世界とメッセージをやりとりするために使用する．`sock` はソケットチャネルである．ソケットチャネルを用いて，ネットワーク上のリモートホストとメッセージを送受信することが可能である．`stdin` は標準入力チャネルで，コンソールからの入力文字列を読み込む．`stdout` は標準出力チャネルで，コンソールに文字列を出力する．

3. Preccs 処理系

本章では，Preccs 処理系の概要を説明する．

3.1 全体構成

図 7 に，Preccs 処理系の全体構成を示す．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 のソースとして形式的に記述する．Preccs コンパイラは Preccs ソースから，メッセージ解析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 C のコードは 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，メッセージ解析やチャネル通信，プロセスのスケジューリングなどを実現するためのルーチンから構成される．それ以外の 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の出力，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．これらの C のコードから C コンパイラによって最終的に実行ファイルを生成する．C のコードを生成することによって，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．

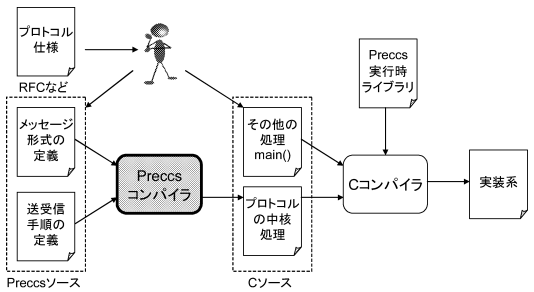

図 7 Preccs 処理系の全体構成
Fig. 7 Architecture of Preccs system.

3.2 実行モデル

図 8 に，Preccs の実行モデルを示す．ネットワークからの入力メッセージは単なるバイト列であるが，メッセージエンコーダがこれを解析（パターンマッチング）し，**図 11**（後出）に示すような構造を持ったメッセージの形式（木構造メッセージ）に変換する．木構造メッセージについては，3.3 節で説明する．プロトコル処理エンジンでは，メッセージの送受信手順に関する処理や，メッセージの生成や書き換えなどを行う．また，必要に応じて，C で記述された処理ルーチンを実行する．メッセージエンコーダは木構造メッセージを再びネットワークへ送信するためのバイト列に変換する．

Preccs ソースのメッセージ形式定義部分から，メッセージデコーダとエンコーダの処理コードが生成される．送受信手順の定義部分からは，プロトコル処理エンジンのコードが生成される．以下では，メッセージ

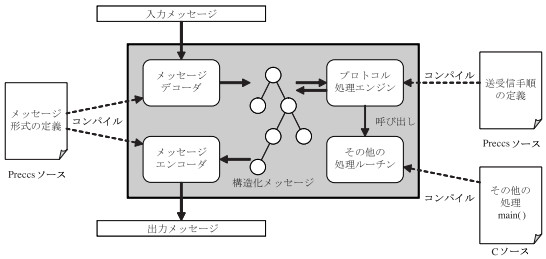

図 8 Preccs 実行モデル
Fig. 8 Execution model of Preccs.

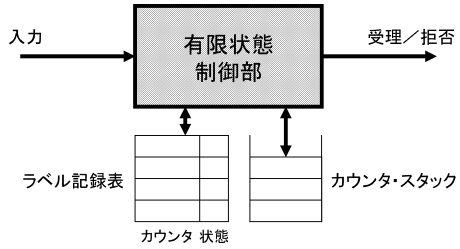

図 9 拡張 NFA
Fig. 9 An extended NFA.

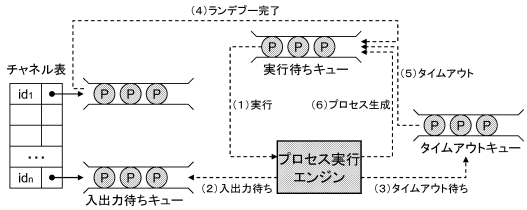

図 10 プロトコル処理エンジン
Fig. 10 Protocol Engine.

デコーダ，メッセージエンコーダ，プロトコル処理エンジンの実装方式について説明する．

3.2.1 メッセージデコーダ

メッセージデコーダは基本的に，非決定性有限オートマトン（NFA）に基づいて，入力メッセージのパターンマッチングを行い，メッセージ形式に対応する木構造メッセージを作成する．ただし，Preccs では，パターンのラベル付けやラベル参照による繰返しを許すように正規表現の拡張を行ったので，一般的な NFA では解析することはできない．そこで，NFA を拡張し，**図 9** に示すようにラベル記号表とカウンタ・スタックを備えた有限状態機械☆を考える．ラベル記録表はラベル識別子をキーとし，カウンタ欄と状態欄をエントリとした表である．ラベル記録表は，ラベルで指定されたパターンとマッチしたデータを，整数値としてカウンタ欄に記録する．状態欄には記録中を示すフラグを立てることができる．カウンタ・スタックは可変長データのマッチング（ラベル参照による繰返し）を実現するためのもので，スタックの要素は整数カウンタである．

拡張 NFA は一般的な NFA に加えて，次のような動作を行う．

記録開始 ラベル l_i の状態欄に記録中を示すフラグを立てる．

記録停止 ラベル l_i の状態欄のフラグを消去する．

入力 入力からデータを 1 オクテット分消費して，次の状態に移る．このとき，記録中状態にあるラベル l_i に対応するカウンタ c_i を以下のように更新する．

$$c_i \leftarrow c_i \times 256 + val(data)$$

ここで，$val(data)$ は 1 オクテット分の入力データを 0〜255 までの値として解釈した値である．

カウンタプッシュ カウンタを 0 で初期化し，カウンタ・スタックにプッシュする．

カウンタポップ カウンタ・スタックをポップする．

インクリメント カウンタ・スタックのトップにあるカウンタをインクリメントする．

カウンタ比較 カウンタ・スタックのトップとラベル l_i の値を比較し，その結果に応じて状態遷移する．

記録開始と停止の間にフラグを立てることによって，ラベルで指定されたパターンを拡張 NFA が走査しているときに，データ列をカウントすることが可能となる．ここでカウントした値は，カウンタ比較時に参照される．これによって，可変長データのマッチングを実現することができる．

3.2.2 メッセージエンコーダ

メッセージエンコーダはメッセージ形式定義の情報をもとに木構造メッセージをたどり，バイト列を生成するだけである．

3.2.3 プロトコル処理エンジン

プロトコル処理エンジンは，メッセージ送受信の手順として定義されたプロセスを実行し，木構造メッセージに対する処理やデータの入出力を行う．**図 10** にプロトコル処理エンジンの概要を示す．プロトコル処理エンジンでは，以下に示すような手順によって，プロセスを実行する．

(1) 実行：実行待ちキューから実行可能状態にあるプロセスを取り出し，プロセスの処理を行う．

(2) 入出力待ち：実行中のプロセスがチャネルに対する入出力を要求すると，対応するチャネルの入出力待ちキューにプロセスを置く．

(3) タイムアウト待ち：実行中のプロセスがタイムアウト待ちになると，そのプロセスをタイムアウトキューに置く．

(4) ランデブー完了：入出力待ちキュー上で，ランデブー可能なプロセスが揃うと，プロセス間でデータ送受信（ランデブー）処理を行い，それらのプロセスを実行待ちキューに置く．

(5) タイムアウト：タイムアウトキューはタイムアウトする時間順にソートされており，タイムア

☆ プッシュダウン・オートマトンに表を追加したものと考えてもよい．

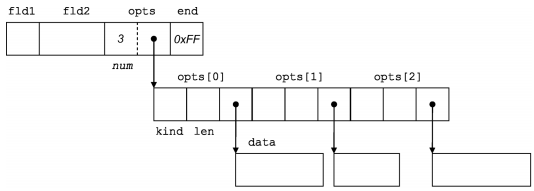

図 11 木構造メッセージの例
Fig. 11 An example of message tree.

ウト時間に達したプロセスは取り出され，実行待ちキューに置かれる．

(6) プロセス生成：プロセスを生成し，初期パラメータをセットした後，実行待ちキューに置く．

3.3 木構造メッセージ

ネットワークから入力されたデータ列は，メッセージデコーダによって解析され，木構造メッセージと呼ぶ，構造を持ったメッセージへと変換される．図 11 は，図 1 で示したメッセージがネットワークから入力された場合に生成される木構造メッセージの例である．fld1，fld2 は，それぞれ 2 オクテットと 4 オクテットの長さを持つフィールドである．これらは固定長のフィールドなので，そのまま木構造メッセージのフィールドへとコピーされる．一方，その後に続く Option 形式を持つフィールドは閉包が指定されているため，0 個以上のデータが連続する可能性を持っている．そこで，木構造メッセージの中では，閉包に対応するフィールドを，実際にマッチした数とデータ領域へのポインタによって構成する．図 11 では，3 個の Option 形式とマッチしたので，*num* で示した領域に 3 を記録し，その直後に opts フィールドのデータ領域へのポインタを保持している．opts フィールドは 3 個の Option 形式を保持するだけの領域を持っており，Option 形式は，それぞれ kind，len，data フィールドから構成されている．data フィールドは len フィールドの値によってその長さが規定される可変長データであり，実際のデータは閉包と同様にポインタの指し示す先に確保される．閉包と異なり，要素数はラベルによって陽に指定されているので，これを記録しておく領域を新たに設ける必要はない．また，ここで示した例には現れないが，選択の場合も閉包と同様の構造となる．ただし，実際にマッチした要素数ではなく，どのパターンとマッチしたかを示すタグを記録しておく．

3.4 コード生成

ここでは，Preccs の記述から C のコードを生成する方法について説明する．メッセージ形式記述部からは拡張 NFA の状態遷移表が生成され，メッセージ処理手順定義部からは，プロセスの各状態に対応するコードが生成される．

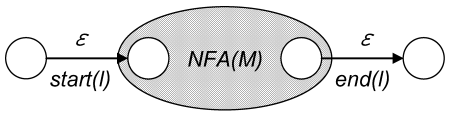

図 12 $l : M$ に対する拡張 NFA
Fig. 12 An extended NFA for $l : M$.

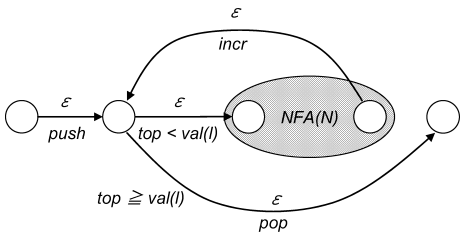

図 13 $N[l]$ に対する拡張 NFA
Fig. 13 An extended NFA for $N[l]$.

3.4.1 拡張 NFA の構成

Preccs のメッセージ形式記述から，3.2.1 項で説明した拡張 NFA を構成する方法について説明する．基本的には Thompson の構成法[1] を用いる．ラベル定義，ラベル参照に対しては，次に示す方法によって拡張 NFA を構成する．

(1) ラベル定義 $l{:}M$ に対しては，**図 12** に示すような拡張 NFA を作成する．ここで，$NFA(M)$ は，メッセージ形式 M に対応する拡張 NFA である．また，$start(l)$，$end(l)$ は，それぞれラベル l に対する記録開始と記録停止の動作を表す．すなわち，$NFA(M)$ の直前で記録開始を行い，直後で記録停止することで，$NFA(M)$ でマッチしたデータが，ラベル l に対応する値としてラベル記録表に記録される．

(2) ラベル参照 $N[l]$ に対しては，**図 13** のような拡張 NFA を作成する．$push$，pop，$incr$ は，カウンタ・スタックに対する操作を表す．また，$top < val(l)$, $top \geq val(l)$ はカウンタ・スタックのトップとラベル l の値を比較した結果の動作を意味する．$NFA(N)$ の前で，カウンタ・スタックに対するプッシュ操作を行い，0 で初期化したカウンタをスタックに積む．次に，ラベル記録表からラベル l に対応する値 $val(l)$ を参照し，カウンタ・スタックのトップの値（この場合は直前に積まれた値 0 のカウンタ）と比較する．比較した結果，カウンタ値が $val(l)$ 以下ならば，$NFA(N)$ とのマッチングを行い，そうでなければカウンタ・スタックをポップし，スタックを元の状態に戻す．$NFA(N)$ とのマッチング

```
while (1) {
  p = get_proc();  // 実行可能プロセスの取得
  switch (p->state) {
  case 0:
    ...  // プロセスの実行コード
    break;
  case 1:
    ...
    break;
  ...
  }
}
```

図 14 プロトコル処理エンジンの C コード
Fig. 14 C code for protocol engine.

が成功すると，カウンタ・スタックのトップの値をインクリメントし，再び，$val(l)$ との比較を行う．結果として，$val(l)$ 回分だけ $NFA(N)$ とのマッチングを試みることになる．

3.4.2 プロトコル処理エンジンのコード生成

プロトコル処理エンジンは，実際には**図 14** に示すような C の while ループと switch 文を用いて実装される．switch 文の各 case 節はプロセス動作式から翻訳されたプロセスの実行コードである．while 文の先頭で，実行待ちキューから実行可能プロセスを取得し（`get_proc()`），プロセス構造体の変数 p へ代入する．そして，p の状態番号によって，各 case 節へ実行を移す．プロセス構造体は，プロセス識別子，状態番号，データ領域などを持ったメモリオブジェクトである．switch 文中の各 case 節をコードブロックと呼ぶ．コードブロックの実行が終了すると，次のプロセスが実行待ちキューから取り出され，それに応じたコードブロックが実行される．このように，プロセス切替えはコードブロック単位で発生する．

各プロセス動作式に対して生成される C のコードを以下に示す．

プロセス定義

p_sig `::=` P`;`

```
case c_i:
  code(P)
  del_proc(p);
  break;
```

プロセス本体のコードを出力した後，自分自身のプロセスを削除するコードを追加する．ここで，p_sig は，プロセスのシグネチャである．$code(P)$ はプロセス定義の本体 P を翻訳した結果のコードを意味する．`del_proc(p)` は，引数で指定したプロセスを削除する C の関数である．c_i は，ここで定義したプロセスの開始状態番号となる．

プロセス呼び出し

$p(v_1, ..., v_n)$

```
...
np = create_proc(p.st, p.sz);
np->data[offset(p.arg_1)] = code(v_1)
  ...
np->data[offset(p.arg_n)] = code(v_n)
...
```

新規にプロセスを生成し，各パラメータをプロセスのデータ領域にコピーする．プロセス p の i 番目の引数 $p.arg_i$ に対応するプロセスのデータ領域の位置は $offset(p.arg_i)$ によって，コンパイル時に計算される．$code(v_i)$ はパラメータ v_i の評価に対応する翻訳コードを意味する．`create_proc`(st, sz) は，開始状態番号 st，データ領域サイズ sz を引数として，プロセスを新規に生成し，そのプロセスを実行待ちキューに置く．$p.st$，$p.sz$ はプロセス p に対応する開始状態番号，データ領域サイズを示す．これらの値はコンパイル時に計算される．

変数生成

`new` var `:` $type$

```
...
p->data[offset(p.var)] =
                create_val(type);
...
```

新規に値を生成し，現在のプロセスの対応するデータ領域にコピーする．var に対応するプロセスのデータ領域の位置は $offset(var)$ によって，コンパイル時に計算される．`create_val`$(type)$ は，型 $type$ で指定した値を生成し，その結果を返す．

条件分岐

$[c_1]\ P_1\ [c_2]\ P_2 \cdots [c_n]\ P_n$

```
...
if (code(c_1)) { code(P_1) }
else if (code(c_2)) { code(P_2) }
...
else if (code(c_n)) { code(P_n) }
...
```

条件分岐は，C の if〜else 文を用いて直接的に翻訳される．

タイムアウト

```
timeout(n)
```

```
case c_i:
  ...
  set_timeout(p,n);
  p->state = c_{i+1};
  break;
case c_{i+1}:
  ...
```

現在実行中のプロセスがタイムアウト処理を行うと，別のプロセスに実行が切り替わる．これを実現するために，コードブロックを 2 つに分割する．最初のコードブロック c_i の中では，後のコードブロック c_{i+1} を指すようにプロセスの状態を設定し，タイムアウトキューに置く．これによって，タイムアウト後に実行可能状態になったプロセスは，コードブロック c_{i+1} から実行が再開されることになる．`set_timeout`(p, n) は，プロセス p が n 秒後にタイムアウトするように，タイムアウトキューに置く．

入力動作

```
ch ? var : type
```

```
case c_i:
  ...
  put_chanin(ch, type, offset(p.var), p);
  p->state = c_{i+1};
  break;
case c_{i+1}:
  ...
```

入力動作もプロセス切替えをともなうので，コードブロックを 2 つに分割する．`put_chanin`$(ch,$ $type,$ $off,$ `p`$)$ は，型 $type$ のデータ入力を待つようにチャネル ch の入力待ちキューにプロセス `p` を置く．このとき，出力待ちプロセスがあればランデブーが成立し，受け取った値をプロセスの対応するデータ領域 `p->data[`off`]` にコピーする．

出力動作

```
ch ! val
```

```
case c_i:
  ...
  put_chanout(ch, code(val), p);
  p->state = c_{i+1};
  break;
case c_{i+1}:
  ...
```

出力動作も入力プロセスと同様にコードブロックを 2 つに分割する．`put_chanout`$(ch, v,$ `p`$)$ は，チャネル ch の出力待ちキューにプロセス `p` を置く．ランデブーが成立すると，値 v が入力プロセスにコピーされる．

逐次実行

```
G_1 , G_2
```

```
  ...
  code(G_1)
  code(G_2)
  ...
```

逐次実行はプロセス切替えをともなわないので，コードブロックの中にプロセス G_1，G_2 の翻訳結果を並べるだけでよい．

選択実行

```
G_1 | G_2
```

```
case c_i:
  ...
  create_dummy_proc(p, c_j);
  create_dummy_proc(p, c_k);
  break;
case c_{i+1}:
  ...
  break;
case c_j:
  code(G_1)
  p->state = c_{i+1};
  put_readyq(p);
  break;
case c_k:
  code(G_2)
  p->state = c_{i+1};
  put_readyq(p);
  break;
```

選択実行は G_1，G_2 に対応するダミーのプロセスを生成し，それらをプロセス実行待ちキューに置く．`create_dummy_proc(p, st)` は，開始状態番号を st とし，プロセス `p` に対応するダミープロセスを生成し，実行待ちキューに置く．ダミープロセスは，G_1，G_2 に対応する各コードブロックから実行を開始するように設定される．したがって，ここで生成した各ダミープロセスは，それぞれのガード式に対応して，入出力待ちやタイムアウト待ちの状態となる．G_1，G_2 に対応する各コードブロックの末尾では，プロセスの次状態を選択実行を抜けた直後のコードブロックに相当する c_{i+1} に設定するコードが置かれる．`put_readyq(p)` は，プロセス `p` を実行待ちキューに置く．

あるダミープロセスにおいて，ランデブーが成立したりタイムアウトしたりすると，もう一方のダミープロセスはキャンセル（削除）され，元々のプロセス `p` の実行が続けられる．

C インライン プロセス動作式中のインラインの C コードは，そのまま該当するコードブロックに埋め込まれる．このとき，プロセス変数へのアクセスは，プロセスのデータ領域を指すポインタへ適宜変換される．

4. 議　　論

本章では，Preccs が対象とする通信プロトコルと関連研究について述べる．

4.1 適 用 範 囲

Preccs の対象として想定している通信プロトコルは，主として OSI7 階層モデルにおける第 2 層（データリンク層）から第 4 層（トランスポート層）までである．Preccs で扱うメッセージはバイナリ形式のパケットを想定しているため，FTP や HTTP などのテキスト形式のメッセージを用いた通信プロトコルは，Preccs の対象外である．

特に Preccs による効果が得られる通信プロトコルは，メッセージの形式と手順が比較的複雑なものである．そのようなプロトコルの例としては，OSPF [17] や IKE などがあげられる．たとえば，IKE ではメッセージの種類に応じて，ヘッダの後に連なるペイロードの構造や要素の数が異なってくる．そのようなメッセージでも 2.1 節で示したような方法を用いて直接的に定義することが可能である．さらにメッセージ形式の定義からメッセージを解析するコードは自動的に生成される．したがって，プログラマがこれらの処理を行うためのプログラムを明示的に書く必要はなくなる．これらのプロトコルは頻繁にパケットをやりとりする性質のものではないので，処理速度を必要とせず，Preccs による実装でも十分に実用的な性能が得られると期待できる．逆に，インターネットの代表的な通信プロトコルである TCP/IP などは，高いスループットや応答性能を求められるため，Preccs で記述することによる恩恵は少ないと考えられる．

また，通信プロトコルの試験を行うためのテストプログラムを Preccs によって記述することも可能である．通信プロトコルの試験を行うには，メッセージを様々なパターンで送受信するようなプログラムを書く必要がある．Preccs ではメッセージ形式の定義から実際に送信するメッセージを作成するコードを自動的に生成でき，さらにプロセスの選択実行により，受信メッセージの種類によって動作を振り分けることが可能であるため，このようなテストプログラムの記述にも適しているといえる．

4.2 関 連 研 究

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 [5] 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ある．しかしながら，CMDL は通信プロトコルの振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 [9] 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 のソースコードを生成する．また，生成したコードの性能も高く，文献 10) 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[11] 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を

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 が対象としているような，より低い階層のプロトコルでは，一般にメッセージは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．

プロセス代数では，現在，π 計算[15] に関する研究がさかんに行われており，これに基づいた Pict[19] などのプログラミング言語が知られている．π 計算では，プロセス間でチャネル自身を送受信できるという特徴を持っている．Preccs ではプロセス間でチャネルを送受信することはできないが，通信プロトコルを記述するという観点からは，このような機能は特に必要ないと考えている．

5. まとめ

本稿では，仕様記述言語 Preccs を提案し，Preccs の実行モデルを示したうえで Preccs のソースから C のコードを生成する方法について説明した．Preccs は通信プロトコルで用いられるメッセージ形式や通信手順といった仕様を自然に，かつ直感的に記述することができる．これによって，通信プロトコルの実装にかかる工数の削減や，信頼性の高い通信プログラムの実現，さらに保守性，可読性の高いコードを得ることができる．

Preccs におけるプロセスの実体はヒープ上に確保されたメモリオブジェクトである．プロセス切替えにともなう処理は複数キューの間でのプロセスの出し入れにすぎず，OS のコンテクストスイッチは絡まない．したがって，Preccs によるオーバヘッドは問題にならない程度であり，十分実用的なコードが生成可能であると考えられる．さらに，文献 18) で示されているように，プロセス切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を生成することも可能と考える．

本手法の有効性を示すには，Preccs と C で通信プロトコルを記述した場合の記述量の比較や，実行時性能の計測などを行う必要がある．そのために，現在，ホスト設定のためのプロトコルである DHCP[2] を対象に動作検証や評価実験の準備を進めている．DHCP は広く用いられている実際的なプロトコルであるが，メッセージ形式やプロトコルの状態遷移は比較的複雑であり，Preccs の有効性を示すには適切な規模の大きさと複雑さを備えている．さらに，IKE や OSPF など，より複雑なプロトコルを対象とした評価も順次行っていく予定である．また，モデル検証の手法[7] などを取り入れることで，Preccs のソースを直接検査するようなツールについても今後，検討していきたい．

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) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers Principles, Techniques and Tools*, Addison-Wesley (1986). 原田賢一（訳）：コンパイラ I 原理・技法・ツール，サイエンス社 (1990).
2) Droms, R.: Dynamic Host Configuration Protocol, RFC2131 (1997).
3) Fischer, S. and Hofmann, B.: An Estelle Compiler for Multiprocessor Platforms, *FORTE '93: Proc. IFIP TC6/WG6.1 6th International Conference on Formal Description Techniques, VI*, pp.171–186, North-Holland (1994).
4) Harkins, D. and Carrel, D.: The Internet Key Exchange (IKE), RFC2409 (1998).
5) 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*, Vol.30, No.2, pp.155–163, CQ 出版社 (2004).
6) 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).
7) Holzmann, G.J.: *The Spin Model Checker — Primer and Reference Manual*, Addison-Wesley (2004).
8) 笠野英松（監修）：通信プロトコル事典，アスキー (1996).
9) Kohler, E.: Prolac: A Language Protocol Compilation, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Insutitute of Technology (1997).
10) Kohler, E., Kaashoek, M.F. and Montgomery, D.R.: A Readable TCP in the Prolac Protocol Language, *ACM SIGCOMM'99*, Vol.29, pp.3–13 (1999).
11) 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), pp.25–35 (2003).
12) Leue, S. and Oechslin, P.A.: On Parallelizing and Optimizing the Implimentation of Communication Protocols, *IEEE/ACM Trans. Networking*, Vol.4, No.1, pp.55–70 (1996).
13) Mañas, J.A. and de Miguel, T.: From LOTOS

to C, *Proc. 1st International Conference on Formal Description Techniques*, pp.79–84, North-Holland (1989).
14) Milner, R.: *Communication and Concurrency*, Prentice Hall (1989).
15) Milner, R.: *Communication and Mobile Systems: the π-Calculus*, Cambridge University Press (1999).
16) 水野忠則（監修）：プロトコル言語，カットシステム (1994).
17) Moy, J.: OSPF Version 2, RFC2328 (1998).
18) Oyama, Y., Taura, K. and Yonezawa, A.: An Efficient Compilation Framework for Languages Based on a Concurrent Process Calculus, *Euro-Par '97, Parallel Processing*, pp.546–553 (1997).
19) Pierce, B.C. and Turner, D.N.: Pict: A Programming Language Based on the Pi-Calculus, CSCI Technical Report 476, Computer Science Department, Indiana University, Indiana (1997).

(平成 17 年 5 月 2 日受付)
(平成 17 年 8 月 8 日採録)

服部 健太（正会員）

1972 年生．1995 年東京大学理学部情報科学科卒業．1997 年東京大学大学院理学系研究科情報科学専攻修士課程修了．同年（株）システム計画研究所入社．主に通信ネットワークに関するソフトウェアの研究開発に従事．2005 年より東京大学大学院情報理工学系研究科創造情報学専攻博士課程に在籍中．

数馬 洋一

1978 年生．2002 年東京都立大学理学部物理学科卒業．2004 年東京都立大学大学院理学研究科物理学専攻修士課程修了．同年（株）システム計画研究所入社．