アプリケーション層プロトコルの記述に基づく拡張性に優れたプロトコル処理コード生成系

An Extensible Protocol-processing Code Generater from Definitions of Application-level Protocols

阿部 勝幸 †, 岩崎 英哉 ††, 河野 健二 †††
Katsuyuki Abe, Hideya Iwasaki, Kenji Kono

† 電気通信大学大学院電気通信学研究科
Graduate School of Electro-Communications, The University of Electro-Communications
abe@ipl.cs.uec.ac.jp
†† 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
iwasaki@cs.uec.ac.jp
††† 慶應義塾大学理工学部情報工学科
Department of Information and Computer Science, Keio University
kono@ics.keio.ac.jp

アプリケーション層プロトコルは HTTP/1.0 から HTTP/1.1 のような拡張がしばしば行われるため，アプリケーション層プロトコルを用いて通信を行うサーバ/クライアントは，プロトコルの拡張が行われるたびに拡張前との差分に対応する必要がある．そのためにはサーバ/クライアントのプログラムコードも随時更新していく必要があり，プログラムの保守性を維持していくことが難しい．この問題を解決するため本論文では，インターネットサーバ/クライアントにおけるアプリケーション層プロトコルに従った状態遷移を記述することにより，サーバ/クライアントのプロトコル処理部のコードを自動生成するシステムを提案する．本システムによる状態遷移記述は，状態の親子関係と，状態遷移のオーバーライドという 2 つの機構を用いることによって拡張性を実現しており，プロトコル拡張の際も，差分のみを記述することでプロトコル処理コードを容易に拡張できる．本システムにより，サーバ/クライアントプログラムの拡張性・保守性の向上が期待できる．

1 はじめに

近年，インターネットの普及と共に，Web やメールのような，インターネット上でサービスを提供する様々なアプリケーションが広く利用されている．インターネット上で動作するこれらのアプリケーションは，アプリケーション層プロトコルを用いて通信を行う．

アプリケーション層プロトコルは，HTTP/1.0[1] から HTTP/1.1[2]，SMTP[3] から ESMTP[4]，POP3[5] から APOP 認証つきの POP3 のような，プロトコルの拡張が行われることがある．このため，アプリケーション層プロトコルを用いて通信を行うアプリケーションのサーバ/クライアントのプログラムでは，プロトコルの拡張が行われるたびに拡張前との差分に対応する必要がある．そのためにはサーバとクライアントのプログラムコードも随時更新していく必要があり，プロトコル処理部の保守性を維持していくことが難しい．

また，アプリケーション層プロトコルは時系列に沿ったメッセージのやり取りとして定義されており，それぞれのメッセージは文字列を主体として構成されている．よって，アプリケーション層プロトコルを用いて通信を行うアプリケーションのサーバ/クライアントのプログラムを記述する際には，多くの文字列処理を必要とする．こうした文字列処理は特別難しいわけではないが，面倒な記述になりがちで，些細なプログラミング上の誤りを犯しやすい．また，時系列に沿ったメッセージのやり取りや状態遷移がプログラムコード中に埋没し，プロトコルの挙動を把握することが難しい．こうした問題は，プロトコル処理部の安全性，プログラムコードの可読性の低下

に繋がる．

これらの問題は，以下のような特徴を持つシステムを開発すれば解決することができる．

1. 頻繁に行われるプロトコルの拡張にも容易に対応できる．
2. サーバ/クライアント間で通信されるメッセージを定義することで，文字列処理を含むプロトコル処理コードを自動生成することができる．
3. サーバ/クライアントの挙動をプロトコルに従った通信メッセージに従う状態遷移として定義することができる．

河野 [6] は，上の 2，3 の特徴を持つシステム April を開発した．April はメッセージ・フォーマットの定義，時系列に沿ったメッセージのやり取り，システムの状態遷移の 3 つの記述から，C 言語で実装されたプロトコル処理部のプログラムコードを自動的に生成するコード生成器である．しかし April はモジュール化の機能を持たないため，プロトコルの拡張には対応していない．このため，プロトコルの拡張に対応するための追加部分を記述すると，この記述が拡張前の記述の中に混在してしまい，プロトコル定義の可読性，保守性が低下する．アプリケーション層プロトコルの拡張は頻繁に行われるため，上の特徴 1 はきわめて重要であり，この点において April は不十分である．

本論文では April のアイデアをベースとし，アプリケーション層プロトコルを対象に，上の 3 つの特徴を持つ拡張性に優れたプロトコル処理コード生成系である April++を提案する．April++で重要なのは，状態の親子関係と状態遷移のオーバーライドという 2 つの機構を用いることで，プロトコルの拡張にも対応しているという点である．これによって，プロトコルの拡張が行われた際にも，拡張前との差分のプロトコル定義を別のファイルに記述することができ，プロトコル定義の保守性を保つことができる．また，どの記述がプロトコルの拡張に対応するための差分の定義かということが明確になり，可読性も向上する．例えば，HTTP/1.0 から HTTP/1.1 のようなプロトコルの拡張に対しても，差分のみを記述することで容易に対応できる．

本研究で作成した April++のコード生成系は，プロトコル定義を読み込み，プロトコル処理コードを自動生成する．具体的には，メッセージの送受信に伴う文字列処理や，プロトコルの状態管理を行う Java のクラス定義を生成する．April++で生成されたプロトコル処理コードはデザインパターンの State パターンを用いて実装されており，各状態ごとにその状態を定義するクラスが存在するため，プロトコルの挙動を状態遷移として捉えやすい．生成されたコードは，拡張前のコードと共に拡張されたプロトコルに対するプロトコル処理コードを形成するので，拡張前のコードが変更されることは一切ない．

April++により自動生成されたコードを用いて POP サーバを実装し，既存の Java で実装された POP サーバと性能比較を行ったところ，オーバーヘッドは 4%程度と，十分に小さいものであった．

以下，2 章では既存の研究の問題点を例を挙げて具体的に述べ，3 章で本システムの基本設計について述べる．4 章では April++の記述法等について詳しく述べ，5 章では April++の記述からサーバ/クライアントのクラス定義を生成するコード生成系について述べる．6 章では April++によるプロトコル定義の記述性と，コード生成系によって生成されたプロトコル処理コードの性能について議論する．7 章で関連研究について述べ，8 章で本論文をまとめる．

2 既存のシステムの問題点

前節でも述べた通り，インターネットサーバ/クライアントプログラムの保守性を維持するためには，プロトコルの拡張に容易に対応でき，サーバ/クライアントのメッセージのやり取りに従う状態遷移の定義からプロトコル処理部のコードを自動生成するようなシステムが望まれる．本節では，後者の特徴を持つ既存のシステムである April[6] について，その具体的な記述法を説明した後，プロトコル拡張に関する問題点について議論する．

2.1 April の記述例

アプリケーション層プロトコルを用いて通信を行うサーバ/クライアントのプログラムでは，文字列で構成されたメッセージが送受信され，そのメッセージの内容によって何らかの処理を行い，次のステップへと進む．April ではこの状態遷移をオートマトンとして表現し，各状態で送受信されるメッセージと，そのメッセージの送受信に伴う状態遷移をテキストで記述する．例として，HTTP/1.0 の状態遷移をオートマトンで表したものを図 1 に，April でこの状態

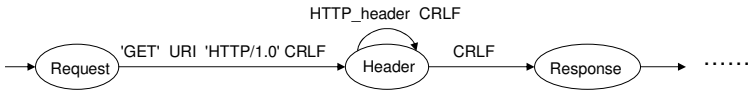


図 1: HTTP/1.0 の状態遷移の一部

```
 1: URI = ('/'['a'-'z', 'A'-'Z', '0'-'9']+)+ ;
 2: HTTP_header = "\w+" + ': ' + "\w+";
 3: CRLF = '\r\n';
 4: state Request = {
 5:     request: <= 'GET' ++ URI ++ 'HTTP/1.0' + CRLF -> Header;
 6: }
 7: state Header = {
 8:     header: <= HTTP_header + CRLF -> Header;
 9:   | crlf: <= CRLF -> Response;
10: }
```

図 2: HTTP/1.0 の April による記述の一部

遷移を記述したものを，図 2 に示す.

図 2 は，クライアントがサーバに対して要求メッセージ (GET メッセージ) を送信する状態を定義している．要求メッセージは，リクエストラインと複数行に渡るヘッダで構成されている．図 2 の例では，4〜6 行目がクライアントからサーバへリクエストラインを送信する状態，7〜10 行目が，クライアントからサーバへヘッダを送信する状態の定義である．各状態の状態遷移は，サーバ/クライアント間で送受信されるメッセージ，遷移先の状態という順に定義されている．

5 行目は，リクエストラインの送受信による状態遷移を表している．`request` は，このメッセージを送受信するために用いるメッセージハンドラの名前である．自動生成されたコードの中でこのメッセージハンドラを呼び出すことによって，メッセージの送受信が行われる．`<=` は，このメッセージがクライアントからサーバへの送信であることを表している．`'GET' ++ URI ++ 'HTTP/1.0' + CRLF` は，送受信されるメッセージを表している．メッセージの中で使われる文字列は，1 行目の `URI` のように，正規表現であらかじめ定義しておくことができる．最後の `-> Header` は，メッセージが送受信されることによって，次にどの状態に遷移するかを表している．

2.2 プロトコル拡張の記述例と問題点

April でプロトコルの拡張を定義する際には，拡張前のプロトコル定義の中に拡張部分を埋め込む形で記述する．このため，プロトコルの拡張が行われた際には追加部分の記述が随所に散らばり，プロトコル定義の保守性，可読性が低下する．例として，HTTP/1.0 から HTTP/1.1 へ拡張したときの状態遷移を図 3 に，この状態遷移を April で記述したものを図 4 に示す.

図 4 の行番号の左に*がついているのが追加した記述である．この例では，6 行目に HTTP/1.1 のリクエストラインの送受信による状態遷移の定義を追加し，12〜15 行目に HTTP/1.1 のヘッダを送受信する状態の定義を追加している．HTTP/1.1 で扱うメッセージは HTTP/1.0 のメッセージとフォーマットが同じであるため，April の定義でも同じような内容の記述が混在し，一見してどこが追加部分であるかがわかりにくい．この例では追加部分はごく一部であるが，大幅な拡張が行われた際には，どこにどのような処理を追加するか，またはどのように変更しなければならないのかを検討した上で，既存の状態定義の中に拡張部分に関する記述を追加する必要がある．そのため，拡張前の定義と拡張部の定義がひとつの状態の中に混在してしまい，可読性，保守性が著しく低下する．また，記述の手間に関しても，一からプロトコルの定義を書き直すのとほぼ同等の手間になってしまうこともある．アプリケーション層プロトコルの更新は頻繁に行われるため，プロトコ

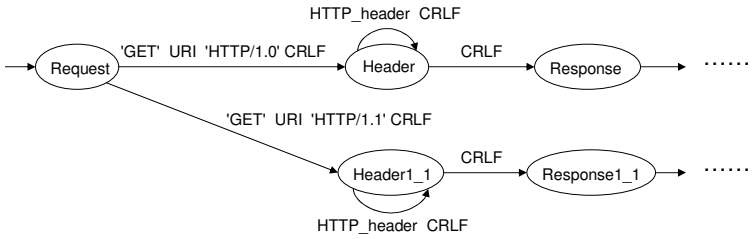


図 3: HTTP/1.1 の状態遷移

```
   1: URI = ('/'['a'-'z', 'A'-'Z', '0'-'9']+)+ ;
   2: HTTP_header = "\w+" + ': ' + "\w+";
   3: CRLF = '\r\n';
   4: state Request = {
   5:     request: 'GET' ++ URI ++ 'HTTP/1.0' + CRLF -> Header;
*  6:   | request1_1: 'GET' ++ URI ++ 'HTTP/1.1' + CRLF -> Header1_1;
   7: }
   8:  state Header = {
   9:      header: HTTP_header + CRLF -> Header;
  10:    | crlf: CRLF -> Response;
  11:  }
* 12:  state Header1_1 = {
* 13:     header1_1: HTTP_header + CRLF -> Header1_1;
* 14:   | crlf: CRLF -> Response1_1;
* 15: }
```

図 4: HTTP/1.1 に拡張したときの April による記述の一部 (∗ のついた行が追加部分)

ルの拡張を行う際には前のバージョンとの差分のみを独立に記述するだけで対応できることが望ましい．

3 April++の基本設計

本節では，April++で HTTP/1.0 のサーバ/クライアントの開発をする場合と，HTTP/1.1 への拡張を行う場合を例として，April++の基本設計について説明する (図 5 参照)．

まず，HTTP/1.0 のサーバ/クライアントを開発する際には，HTTP/1.0 で送受信されるメッセージと，それに伴う状態遷移を記述する．これをプロトコル定義と呼ぶ．HTTP/1.0 のプロトコル定義を April++のコード生成系に入力として与えると，HTTP/1.0 で行われるメッセージの送受信に伴う文字列処理と，サーバ/クライアントの状態管理を行う Java のクラス定義が自動生成される．プログラマは状態遷移時のアクションのみを記述し，April++が自動生成したクラス定義と組み合わせることで，HTTP/1.0 のサーバ/クライアントのプログラムコードを得ることができる．

HTTP/1.1 への拡張を行う場合，プログラマは HTTP/1.0 と HTTP/1.1 の差分のみのプロトコル定義を記述する．これを April++のコード生成系に入力として与えると，その差分に対応したクラス定義が自動生成される．これを HTTP/1.0 に対応したクラス定義と組み合わせることによって，HTTP/1.1 に対応したクラス定義が得られる．これをさらにプログラマが記述した HTTP/1.0 と HTTP/1.1 の状態遷移時のアクションと組み合わせることで，HTTP/1.1 のサーバ/クライアントのプログラムが得られる．

4 April++言語によるプロトコル記述

April++を用いてプロトコル処理部を開発する場合，プロトコルを定義する言語である April++言語を用いてプロトコル定義を記述する．4.1 節では HTTP/1.0 を例に April++言語による具体的な記述法を示す．また，4.2 節ではプロトコルの拡張を記述する際に，差分の定義を拡張前の定義と独立に記述するための基本機構について説明し，4.3 節では HTTP/1.0 から HTTP/1.1 への拡張を例に，本論文

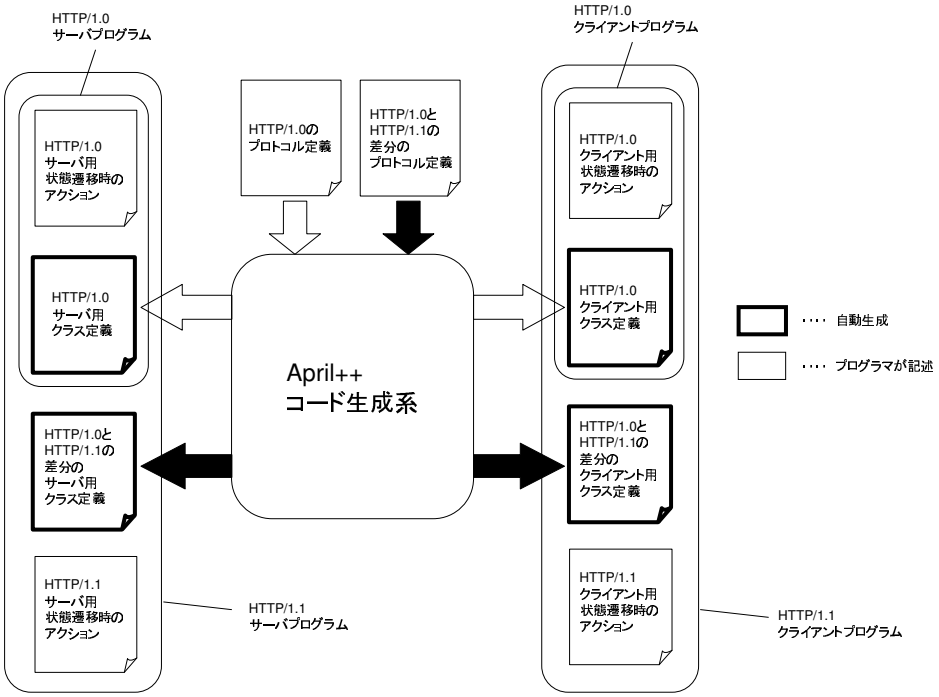


図 5: April++の基本設計

で提案する機構を用いてプロトコルの拡張を記述したときの例を示す.

4.1 April++の記述例

April++でプロトコル定義を記述する場合，Aprilと同様にプロトコルの状態遷移をオートマトンで表現し，各状態ごとに送受信するメッセージと遷移先の状態を定義する．例えばHTTP/1.0では，コネクションの確立後，まずサーバはクライアントからリクエストラインである `GET /index.html HTTP/1.0\r\n` というようなメッセージを受信し，続いてヘッダとして `Content-Type: text/html\r\n` というようなメッセージを, あれば複数個受信する．最後に空行 `\r\n` により，次の状態へと進む．これを次のような状態遷移と考える (図 1，2 参照).

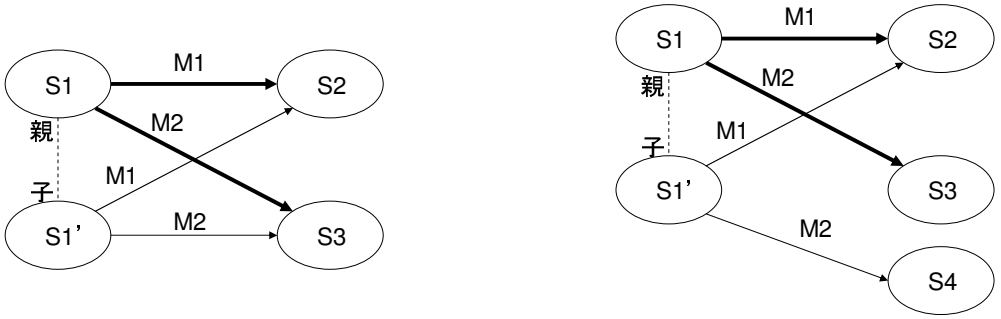
1. コネクション確立後の状態 (`Request` とする) から，リクエストラインの送受信により，次にヘッダを受信する状態 (`Header` とする) に遷移する.
2. 状態 `Header` でヘッダの受信をしても，状態は `Header` のままである.
3. 状態 `Header` で `\r\n` を受信すると，状態は次の状態 (`Response` とする) に遷移する.

上で説明した状態遷移を実際に April++で記述すると，図 6 のようになる．まず 1 行目の`%start` と 2 行目の`%terminal` で，初期状態と終了状態を定義している．次に，HTTP ヘッダや URI，CRLF 等の，送受信されるメッセージの中で使われる文字列の定義を必要に応じて行う．例えば`\r\n` という文字列は全てのメッセージの中で使われるため，5 行目で `CRLF` という名前の文字列であると定義している．文字列は正規表現で定義し，ここで定義した文字列はサーバとクライアントの間で送受信されるメッセージを定義するときに使うことができる．この正規表現の文法は，Java.util.regex で定義されている正規表現構文に準じており，アプリケーション層プロトコルのメッセージを記述できるだけの記述力を持っている.

次に，各状態を定義する．6〜9 行目は，`Request` 状態の定義である．8 行目で，サーバとクライアントの間で送受信されるメッセージと，そのメッセージを送受信したときの遷移先の状態を定義している．ここは，`'GET' ++ uri ++ 'HTTP/1.0' + CRLF` というメッセージを送受信し，`Header` という状態に遷移することを表している．ダブルクオートで囲まれた文字列は大文字と小文字を区別しないことを表しており，シングルクオートで囲まれた文字列は，大文字と小文字を区別することを表す．また，`+` は単純

```
 1: %start Request;
 2: %terminal Close;
 3: URI = ('/'['a'-'z', 'A'-'Z', '0'-'9']+)+ ;
 4: HTTP_header = "\w+" + ': ' + "\w+";
 5: CRLF = '\r\n';
 6: state Request {
 7:     uri = URI;
 8:     'GET' ++ uri ++ 'HTTP/1.0' + CRLF :< Header;
 9: };
10: state Header {
11:     HTTP_header + CRLF :< Header;
12:     CRLF :< Response;
13: };
```

図 6: HTTP/1.0 の April++による記述の一部



(a) S1' を S1 の子状態としたときの状態遷移

(b) (a) の子状態 S1' で，遷移先の状態を S3 から S4 へオーバーライドした時の状態遷移．

図 7: プロトコル拡張を記述するための基本機構．点線は状態の親子関係を表す．太線は各状態で定義された状態遷移，細線は親状態から継承した状態遷移を表す．

な文字列の連結を，++ は 1 個以上の空白を挟んだ文字列の連結を表している．メッセージの後の :< は，クライアントからサーバへのメッセージの送信であることを表している．また，7 行目の定義は URI にあたる文字列を切り出し，uri という名前の変数に保持することを定義している．ここで保持した文字列は，状態遷移時のアクションを記述する際に使用することができる．また，1 つの状態の中で状態遷移が複数通りある場合は，10〜13 行目の Header 状態の定義のように，1 つの状態の中に複数の状態遷移を記述する．

4.2 プロトコル拡張を記述するための基本機構

ここでは，モジュール化を実現するための基本機構である状態の親子関係と状態遷移のオーバーライドについて説明する．この 2 つは，オブジェクト指向言語におけるクラスの継承と，メソッドのオーバーライドに類似している．

プロトコルの拡張を記述する際に問題となるのは，ある状態に新しい状態遷移を追加する部分である．April (図 3) では，図 1 で定義されていた Request 状態に HTTP/1.1 の要求メッセージによる状態遷移を追加しているため，April の記述としても，Request 状態の定義中に新しい状態遷移の記述を追加しなければならず，これが拡張前の記述と差分の記述が混在する原因となっていた．これを解消するためには，拡張前の状態に新しい状態遷移を追加するのではなく，拡張前と拡張部の両方の状態遷移を持つ新しい状態を定義する必要がある．

このような状態を定義するため，拡張前のある状態の状態遷移を受け継いだ子状態というものを考える．状態の親子関係を適用したオートマトンを，図 7 (a) に示す．子状態として宣言された状態は，親状態が持つ状態遷移を引き継ぐ．図 7 (a) の例では，状

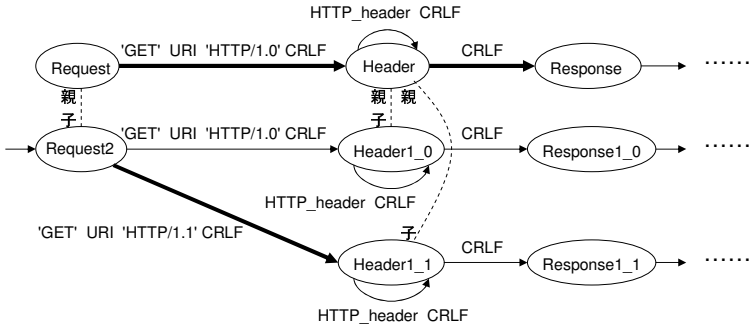


図 8: 状態の親子関係を用いた HTTP/1.1 の状態遷移

```
 1: import HTTP1_0.april;
 2: %start Request2;
 3: %terminal Close;
 4: URI = ('/'['a'-'z', 'A'-'Z', '0'-'9']+)+ ;
 5: HTTP_header = "\w+" + ': ' + "\w+";
 6: CRLF = '\r\n';
 7: state Request2 :parent Request :override Header -> Header1_0 {
 8:     uri = URI;
 9:     'GET' ++ uri ++ 'HTTP/1.1' + CRLF :< Header1_1;
10: };
11: state Header1_0 :parent Header :override Header -> Header1_0,
                                           Response -> Response1_0;
12: state Header1_1 :parent Header :override Header -> Header1_1,
                                           Response -> Response1_1;
```

図 9: HTTP/1.1 へ拡張したときのの April++による記述の一部

態 S1 の子状態 S1' を定義すると，状態 S1' は親状態 S1 と同じ状態遷移を引き継ぎ，図のような状態遷移を持つことになる．

また，子状態が親状態と同じメッセージによって状態遷移をするが，親状態における遷移先とは別の状態に遷移させたいとき，状態の遷移先だけを変えるという機構を用意する．これを，状態遷移のオーバーライドと呼ぶ．図 7 (a) のオートマトンに状態遷移のオーバーライドを適用したときの状態遷移を，図 7 (b) に示す．図 7 (a) の状態 S1' が単純に親状態と同じ状態遷移をするのではなく，メッセージ M2 による状態遷移先を状態 S4 に変更したい場合，遷移先を S3 から S4 へオーバーライドすると宣言すれば，図 7 (b) のような状態遷移を定義することができる．状態の親子関係と状態遷移のオーバーライドを用いることにより，拡張前と拡張部の両方の状態遷移を持つ新しい状態の定義を，一から書き直すよりも簡単に記述することができる．

状態の親子関係と，状態遷移のオーバーライドを適用した HTTP/1.1 の状態遷移を表すオートマトンを図 8 に示す．Request 状態の子状態 Request2 を定義すると，Request2 状態では HTTP/1.0 の要求メッセージによる状態遷移が引き継がれる．ここに差分である HTTP/1.1 の要求メッセージによる状態遷移を記述することで，HTTP/1.0 と HTTP/1.1 の両方の要求メッセージによる状態遷移をもつ状態を定義することができる．

また，図 8 のサーバがヘッダを受信する Header1_1 状態は，メッセージに関しては HTTP/1.0 でヘッダを受信する Header 状態と同じで，遷移先だけが異なる．このような場合，Header1_1 状態は Header 状態の子状態であり，さらに遷移先を Response 状態から Response1_1 状態にオーバーライドすると定義すれば，図 8 のようなオートマトンで表される状態遷移を定義することができる．

このようにして，拡張前と拡張部の両方の状態遷移を持つ状態を一から定義するのではなく，拡張前の定義を利用して差分のみを記述することで，プロ

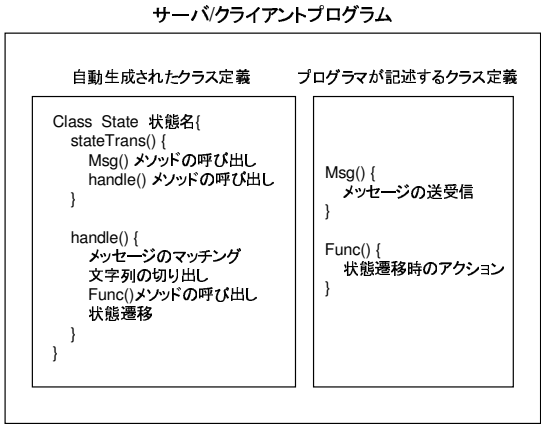


図 10: April++によって得られるサーバ/クライアントプログラムの構成

トコルの拡張に対応した状態を新しく定義することができる．

4.3 プロトコルの拡張の記述例

April++では，4.2 節で説明した状態の親子関係と状態遷移のオーバーライドという 2 つの機構を用いることで，プロトコルの拡張を定義する際にも差分を記述するだけでよい．この 2 つの機構の記述例として，図 6 の HTTP/1.0 の定義を HTTP/1.1 へ拡張したときの記述例を図 9 に示す．これは，図 8 のオートマトンに対応している．

まず 1 行目で，この記述はどの記述に対する拡張であるかをファイル名を指定することで定義している．7 行目の`:parent Request` という記述は，この状態の親状態が図 6 の `Request` 状態であるということを定義している．この定義によって `Request2` 状態では，図 6 の `Request` 状態の状態遷移を引き継ぐ．また，`:override Header -> Header1_0` という記述は，親状態である `Request` 状態から受け継いだ `Header` 状態への遷移を，`Header1_0` という状態への遷移に変更するものである．さらに 9 行目の HTTP/1.1 のリクエストラインによる状態遷移を記述することで，HTTP/1.0 と HTTP/1.1 の両方のメッセージに対応した状態を定義している．April++では，このようにして拡張前の記述を利用し，拡張前のプロトコル定義を変更することなく差分のみを記述することでプロトコルの拡張を定義することができる．

5 コード生成系

April++のコード生成系は，April++のプロトコルの定義を入力として与えられると，メッセージの送受信に伴う文字列処理，サーバ/クライアントの状態管理を行う Java のクラス定義を自動生成する (図 5 参照)．自動生成されたクラス定義は，実際に送受信されたメッセージと April++の記述中の正規表現とのマッチングに成功すれば，プログラマが記述したアクションを呼び出す．また，このクラス定義はデザインパターンの State パターンを用いて実装されており，各状態ごとにクラスが定義されている．4.2 節で説明した状態の親子関係と状態遷移のオーバーライドは，クラスの継承とメソッドのオーバーライドによって実現されているため，April++の定義だけでなく，自動生成されるコードについても，プロトコルの拡張により拡張前のクラス定義が変更されることはない．

April++によって得られるサーバ/クライアントプログラムの構成を図 10 に示す．April++のコード生成系が自動生成したサーバ/クライアントのクラス定義は各状態ごとに `stateTrans()` というメソッドを定義しており，このメソッドを呼び出すと，メッセージの送受信，メッセージの解析，文字列の切り出し，アクション，状態遷移の順で処理を行う．サーバ/クライアントの主ループは，現在の状態から `stateTrans()` メソッドによって状態を更新することを繰り返す．`stateTrans()` メソッドの中では，メッセージの送受信を行う `Msg()` メソッドと，そのメッセージを解析し，結果に応じてアクションを呼び出し，状態遷移を行う `handle()` メソッドを呼び出している．また，`handle()` メソッドの中では，プログラマが記述したアクションを実行する `Func()` メソッドを呼び出している．プログラマは，`Msg()` メソッドと `Func()` メソッド (実際のメソッド名は状態名に `Msg` あるいは `Func` を続けたものになっている) を然るべく定義するだけでよく，それ以外の処理に関するコードは April++のコード生成系が自動生成する．

プログラマが定義するメソッドは一つのクラスの中でまとめて定義されており，プロトコルの拡張が行われた際にプログラマが定義するメソッドは，拡張前の `Msg()` メソッドと `Func()` メソッドが定義されているクラスを継承したクラスの中で定義する．

例として，図 6 の `Request` 状態に関して自動生成されたサーバ側のクラス定義を図 11 に示す．また，プロトコルの拡張を記述した定義から自動生成され

```
 1: public class Server {
 2:     public static void main(String[] args) {
 3:         ServerFunc serverFunc = new ServerFunc();
 4:         State state = new Request();
 5:         while ( true ) {
 6:             state = state.stateTrans(serverFunc);
 7:             if (state instanceof Close) {
 8:                 break;
 9:             }
10:         }
11:     }
12: }
13: class Request {
14:     public State nextHeader() {
15:         return new Header();
16:     }
17:     public State stateTrans(ServerFunc serverFunc) {
18:         String message = serverFunc.RequestMsg();
19:         return handle(serverFunc, message);
20:     }
21:     public State handle(ServerFunc serverFunc, String message) {
22:         Pattern p1 = Pattern.compile("GET (/\\w+)+ HTTP/1.0\r\n");
23:         Matcher m1 = p1.matcher(message);
24:         if (m1.matches()) {
25:             String uri = m1.group(1);
26:             serverFunc.RequestFunc(uri);
27:             return nextHeader();
28:         }
29:     }
30: }
```

図 11: April++によって自動生成された HTTP/1.0 のサーバ用クラス定義の一部

るクラス定義の例として，Request 状態の子状態である図 9 の Request2 状態に関して自動生成されたサーバ側のクラス定義を図 12 に示す.

5.1 Msg() メソッド

Msg() メソッドは，サーバ/クライアント間でのメッセージの送受信を行うメソッドである．例えば図 11 の 18 行目で呼び出されている RequestMsg() メソッドは，図 6 の 8 行目で定義されているように，'GET' ++ uri ++ 'HTTP/1.0' + CRLF というようなメッセージを，クライアントならサーバに送信するように，サーバならばクライアントから受信するようにプログラマが定義する (図 5 参照)．このメソッドによって送受信されたメッセージは handle() メソッドに引数として渡される．

5.2 handle() メソッド

handle() メソッドは Msg() メソッドによって送受信されたメッセージを引数に受け取り，その文字列と，April++言語で定義されたメッセージの正規表現とのマッチングを行い，マッチングの結果によってアクションを呼び出し，状態遷移を行う．例えば図 6 の Request 状態では，Msg() メソッドで送受信したメッセージが GET URI HTTP/1.0 CRLF という正規表現にマッチするかどうかを判定し，マッチすればこのメッセージに対応した Func() メソッドを呼び出し，Header 状態への状態遷移を行う．また，メッセージのマッチングを行う際，必要な文字列の切り出しを行うことができる．図 11 の例では，22，23 行目でメッセージのマッチングを行い，24〜28 行目でマッチングに成功したときの処理を行っている．25 行目では URI の切り出し，26 行目で Func() メソッドの呼び出し，27 行目で状態遷移を行っている．

```
 1: public class Server1_1 {
 2:     public static void main(String[] args) {
 3:         ServerFunc2 serverFunc2 = new ServerFunc2();
 4:         State state = new Request2();
 5:         while ( true ) {
 6:             state = state.stateTrans(serverFunc2);
 7:             if (state instanceof Close) {
 8:                 break;
 9:             }
10:         }
11:     }
12: }
13: class Request2 extends Request {
14:     public State nextHeader() {
15:         return new Header1_0();
16:     }
17:     public State nextHeader1_1() {
18:         return new Header1_1();
19:     }
20:     public State stateTrans(ServerFunc2 serverFunc2) {
21:         String message = serverFunc2.Request2Msg();
22:         return handle(serverFunc2, message);
23:     }
24:     public State handle(ServerFunc2 serverFunc2, String message) {
25:         Pattern p1 = Pattern.compile("GET (/\\w+)+ HTTP/1.1\r\n");
26:         Matcher m1 = p1.matcher(message);
27:         if (m1.matches()) {
28:             String uri = m1.group(1);
29:             serverFunc2.Request2Func(uri);
30:             return nextHeader1_1();
31:         }
32:         else {
33:             return super.handle(serverFunc, message);
34:         }
35:     }
36: }
```

図 12: April++によって自動生成された HTTP/1.0 と HTTP/1.1 の差分のサーバ用クラス定義の一部

5.3 Func() メソッド

Func() メソッドは状態遷移時のアクションを定義するメソッドであり，プログラマが定義する．例えば Web サーバであれば，GET メッセージで指定されたファイルを開いたり，そのファイルの中身を読み込むといった処理を行う．このような処理は，Func() メソッドの中で定義しておく．このメソッドは，handle() メソッドの中で行われるメッセージの解析の結果に応じて呼び出される．

5.4 モジュール化を実現する機構

プロトコルの拡張に対応するための機構である状態の親子関係と状態遷移のオーバーライドは，オブジェクト指向言語におけるクラスの継承と，メソッドのオーバーライドによって実現している．図 12 の例では，Request2 状態 (図 9) のクラスが，Request 状態 (図 6) のクラスを継承し，handle() メソッドの中でメッセージのマッチングに失敗した時に，親クラスの handle() メソッドを呼び出している (33 行目)．これにより，次に親状態で定義されているメッセージのマッチングと状態遷移を試すことになる．

また，図 11 の 14〜16 行目，図 12 の 14〜16 行目，17〜19 行目は状態遷移を行うメソッドであり，図 12

表 1: プロトコル定義の記述行数の比較

	April の記述行数	April++の記述行数
HTTP/1.0	123	125
SMTP	65	76
POP3	75	98

表 2: プロトコル拡張に対する追加行数の比較

	April		April++	
	追加箇所	追加行数	追加箇所	追加行数
HTTP/1.0 → HTTP/1.1	6	94	1	113
POP3 → APOP 対応 POP3	3	10	1	18

の 14〜16 行目で図 11 の 14〜16 行目のメソッドをオーバーライドし，別の状態に遷移するよう変更している．このようにして親状態の状態遷移を変更することによって，状態遷移のオーバーライドを実現している．この実装法によって，プロトコル定義だけでなく自動生成されるクラス定義に関しても，拡張前のクラス定義を変更することはない．

6 実験

6.1 記述性

April++言語によるプロトコルの記述力を検証するため，HTTP，SMTP，POP3 のプロトコル定義を記述した．HTTP/1.0 は GET，HEAD，POST の 3 つのみに対応した単純化したもので，SMTP と POP3 は最低限のコマンドのみ定義している．HTTP/1.0，SMTP，POP3 を April++で記述したものと April で記述したものとの行数の比較を表 1 に示す．また，拡張性に関する評価を行うため，HTTP/1.0 を HTTP/1.1 に拡張した場合と，POP3 に APOP コマンドを用いた認証を拡張した場合の記述について，追加行数と追加箇所を April と比較したものを表 2 に示す．

HTTP/1.0 の記述行数にはほとんど差がなかったが，SMTP，POP3 の記述では，拡張 April のほうが若干行数が多くなった．これは，クライアントが 1 つのメッセージを送信するたびにサーバから応答が返ってくる場合，April ではそれを 1 つの状態の中で定義できるためである．例えば POP の認証状態では，クライアントがサーバに対して `USER ユーザ名\r\n` というメッセージを送信し，それに対してサーバは `+OK\r\n` というメッセージで応答する．このような場合，April ではこの 2 つのメッセージのやり取りを 1 つの状態で定義できるのに対し，April++では，クライアントが USER コマンドを送る状態とサーバが応答する状態を分けて記述しなくてはならない．これは，クライアントからのリクエストとサーバの応答を 1 つの状態の中で記述できるような構文糖衣を用意すれば解決できる．

プロトコルの拡張については，April では元の記述に差分のコードを追加しているのに対し，April++では状態の親子関係と状態遷移のオーバーライドの記述をしているため，若干記述行数が増えている．しかし記述行数の増加量は，どちらも April で記述した場合の 10%増程度におさえられている．重要なのは，April では拡張に伴う差分の記述が随所に散らばっているのに対し，April++では差分の記述が 1 箇所にまとめられ，別のファイルの中で定義することができた点である．April++を用いてプロトコルの定義を記述する際には，差分の記述が 1 箇所にまとめられており，それによって記述する行数が極端に増加することもなく，適切なモジュール化ができている．

現状の April++では，IMAP[7] のような非同期的な通信を行うプロトコルの記述が難しい．これは，IMAP が複数のメッセージを非同期的に処理するため，同時に処理している全てのメッセージに対し，その返答が受理できるように仕様を記述しなければならないためである．そのため状態数が増加し，現状の April++ ではプロトコルの自然な定義が難しい．これは，複数のメッセージが同時に処理されること

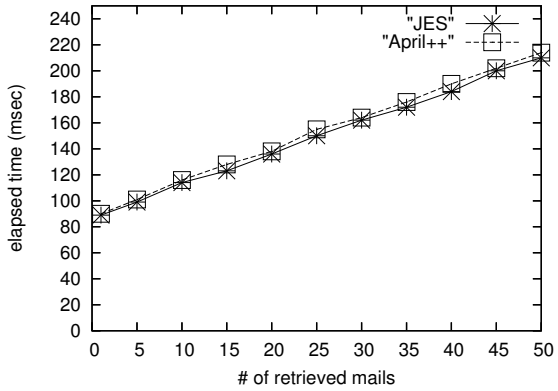


図 13: メール受信時間の比較

を表すプリミティブを持たないためであり，新たなプリミティブを導入すれば解決できると考えられる．

しかし，アプリケーション層プロトコルの中でも IMAP のように非同期的な通信を行うプロトコルは稀である．実際，April++言語を用いて HTTP，SMTP，POP 等の主要なアプリケーション層プロトコルの定義が自然に記述することができ，April++は実用的なプロトコルの定義を記述するのに十分な記述力があることが確認できた．

6.2 実行時性能

April++によって自動生成されたクラス定義の実行時オーバーヘッドを調べるため，Java で実装された POP サーバである JES[8] と，April++を用いて実装した POP サーバとの性能比較を行った．サーバ/クライアントには Linux 2.6.8 の稼働している Pentium4 3.0GHz プロセッサ，メモリ 1GB を搭載した PC/AT 互換機を用いた．サーバとクライアントは，Gigabit Ether によって接続されている．

図 13 に，実験結果を示す．横軸は受信したメールの個数，縦軸はメールの受信に要する時間 (msec) を表す．メールの大きさは 1 通 4Kbyte とした．April++を用いて実装したサーバのオーバーヘッドは，最大でも JES の実行時間の 4%程度に抑えられた．April++が対象とするアプリケーション層プロトコルではネットワークによる通信遅延が生じるため，インターネット環境では POP 以外のアプリケーションでも April++によるオーバーヘッドはごく小さいと期待できる．

7 関連研究

April++はプロトコルの定義からプロトコル処理コードを自動生成することによって，プログラマの負担を軽減する．このような言語システムは，これまでにもいくつか提案されている．

April[6] は本論文で提案している April++のアイデアのベースとなるものである．しかし，たびたび指摘している通り，April はモジュール化の機構を持たないため，アプリケーション層プロトコルの更新や改訂が行われるたびにプロトコルの定義全体を変更する必要がる．April++はこの問題を解決し，拡張性が高く保守性に優れたシステムとなっている．

MSPL[9] は April++と同様，アプリケーション層プロトコルの仕様記述からプロトコル処理部を自動生成するシステムである．しかし，MSPL はプロトコルを状態遷移として捉えておらず，メッセージのフォーマットのみを記述する．そのため，サーバ/クライアントの状態管理，状態遷移に関するコードは自動生成されず，すべてプログラマが記述する．

Prolac[10] は，静的に型付けされたオブジェクト指向のプロトコル記述言語であり，プロトコルスタックの記述に特化している．これは April++ とは異なり，正規表現に基づいたメッセージを記述することができないので，その記述力に問題がある．トランスポート層以下のレイヤが対象のシステムには，Prolac の他に，Promela++[11]，PacketTypes[12]，Preccs[13] などがある．しかし，これらはいずれも April++ とは対象とするレイヤが異なるため，アプリケーション層プロトコルを記述することができない．

多くのプロトコル記述言語は，LOTOS[14]，E-LOTOS[15]，Estelle[16]，SDL[17] に代表されるように，通信プロトコルの形式的な仕様検証を目的としている．そのそれぞれの言語は，LOTOS とその拡張である E-LOTOS はプロセス代数，SDL は有限状態機械といったように，検証のための理論に基づいている．したがって，これらの言語を用いるには，背後にある理論を理解する必要があり，一般的な利用者にとって使いこなすのが困難である．April++は，形式的な検証を目的とはせず，プロトコルの拡張にも対応した実用的なコードの自動生成を目的とした一種のプログラミング支援ツールであり，理論的な予備知識は一切必要としない．

8 おわりに

本論文では，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拡張性に優れたプロトコル処理コード生成系，April++を提案した．April++ではプロトコルの定義から，プロトコルに従ってメッセージの送受信を行うコードを自動生成する．また，April++言語による状態遷移記述は拡張性に優れており，頻繁に行われるアプリケーション層プロトコルの更新にも，差分のみを記述することで容易に対応できる．April++によるプロトコル定義の例として，HTTP/1.0 の定義を示した．また，拡張性の高さを示す例として，HTTP/1.0 に HTTP/1.1 を拡張した際の定義を示した．

April++を用いて多くのプロトコルを定義することができると共に，様々なプロトコルの拡張にも対応でき，適用範囲は十分に広い．実際 HTTP，SMTP，POP3 等の主要なアプリケーション層プロトコルを定義し，プロトコル処理部のコードを自動生成することができた．また，状態の親子関係と状態遷移のオーバーライドという 2 つの機構を用いることで，プロトコルの定義，生成されるコード共に拡張前の記述に手を加えることなくプロトコルの更新に対応でき，拡張性を実現することができている．

現状の April++は，IMAP のように非同期的な通信を行うプロトコルの記述には適していない．このようなプロトコルも自然に記述できるような拡張が今後の課題である．

参考文献

[1] Fielding, R.: Hypertext Transfer Protocol – HTTP/1.0, Internet Request for Comments (RFC) 1945 (1996).

[2] Fielding, R., Gettys, J., Mogul, J., Frystyk, H. and Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1, Internet Request for Comments (RFC) 2068 (1997).

[3] Postel, J.B.: Simple Mail Transfer Protocol, Internet Request for Comments (RFC) 821 (1982).

[4] Freed, N. and Rose, M.: SMTP Service Extensions, Internet Request for Comments (RFC) 1869 (1995).

[5] Myers, J. and Rose, M.: Post Office Protocol – Version 3, Internet Request for Comments (RFC) 1939 (1996).

[6] 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, Vol. 44, No. SIG2, pp.25–36 (2003).

[7] Crispin, M.: Internet Message Access Protocol Version 4rev1, Internet Request for Comments (RFC) (2060) 1996.

[8] Daugherty, E.: Java Email Server (JES). http://www.ericdaugherty.com/java/mailserver/index.html.

[9] Douglas, M.A.L. and Chan, P.K.: A Protocol Language Approach to Generating Client-Server Software. Technical Report, Department of Computer Science, Florida Insitute of Technology (2000). CS-2000-2.

[10] Kohler, E., Kaashoek, M.F. and Montgomery, D.R.: A Readable TCP in Prolac Programing Language, *Proc. ACM SIGCOMM*, pp.3–13 (1999).

[11] Basu, A., Hayden, M., Morrisett, G. and von Eicken, T.: A Language-Based Approach to Protocol Construction, *Proc. ACM SIGPLAN Workshop on Domain Specific Languages* (1997).

[12] McCann, P.J. and Chandra, S.: Packet Types: Abstract Specification of Network Protocol Messages, *Proc. ACM SIGCOMM*, pp.321–333 (2000).

[13] 服部健太, 数馬陽一. 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, 情報処理学会第 54 回プログラミング研究会 (2005).

[14] ISO 8807: Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behavior (1989).

[15] ISO/IEC 15437:2001: Information Technology – E-LOTOS (2001).

[16] Amer, P.D., Sethi, A.S., Fecko, M. and Uyar, M.: Formal Design and Testing of Army Communication Protocols Based on Estelle, *Proc. 1st ARL/ATIRP Conf*, pp.107–114 (1997).

[17] ITU-T Recommendation Z.100: Specification and description language (SDL) (1999).