

述語列のリアルタイム収束性に基づく リスポンシブプロトコルの検証法

川島健一 角田良明 菊野亨

大阪大学 基礎工学部 情報工学科

〒 560 大阪府豊中市待兼山町 1-1

あらまし 情報通信システムの大規模化に伴って通信プロトコルの高信頼性やリアルタイム性が強く要求されるようになっている。プロセス間の同期のずれにより異常状態に陥ってもリアルタイムに正常状態に回復するための機能を備えているプロトコルをリスポンシブプロトコルと呼ぶ。述語列のリアルタイム収束性とは、プロトコルの各状態を述語で表して(プロトコルの動作に対応する)述語の系列を考えると、正常状態を表す述語にある系列長の範囲内で収束することを言う。本稿では、この述語列の収束性に基づいたリスポンシブプロトコルの検証法を提案する。本検証法ではまず、任意の異常状態に陥っても有限時間内に正常状態に回復するかどうかを検証する。次に、正常状態に回復するまでの時間を計算して、それがあらかじめ決められている時間より短いかどうかを検証する。

Verification of Responsive Protocols Based on Real-Time Convergence of Sequences of Predicates

Kenichi Kawashima, Yoshiaki Kakuda and Tohru Kikuno

Department of Information and Computer Sciences

Faculty of Engineering Sciences, Osaka University

1-1, Machikaneyama-cho, Toyonaka-shi, Osaka 560, Japan

{kawasima, kakuda, kikuno}@ics.osaka-u.ac.jp

Abstract As communication systems become large and complicated, high reliability and performance in presence of faults for communication protocol are required. Communication protocols that can make real-time recovery from any abnormal state are called responsive protocols. This paper proposes a new verification method of responsive protocols. In this method, first, it is verified whether the protocol reaches to a normal state from any abnormal state within a finite time, and then it is verified whether the time required to reach a normal state from any abnormal state is shorter than a predetermined time.

1 まえがき

情報通信システムの大規模化に伴い、システムの障害が起きてもシステムの構成要素間の処理をリアルタイムに続行する必要性が高まっている。このためには、通信プロトコルのリアルタイム性を考慮した高信頼化が不可欠である。リアルタイム高信頼化機能をもつシステムを一般にリスポンシブシステム[4]という。文献[6, 7]では、リスポンシブシステム技術を通信用プロトコルに応用したリスポンシブプロトコル (responsive protocol) を新たに提案している。このリスポンシブプロトコルでは、伝送路の遅延、メッセージの転送エラー、プロセスの一時的な故障などを原因とするプロセス間の同期のずれにより異常状態に陥ったときに、リアルタイムに正常状態へ回復するための機能が必要であることを示している。

異常状態から正常状態への回復を例外処理のルーチンを用いずに実行するプロトコルは自己安定プロトコル (self-stabilizing protocol)[1] と呼ばれ、既にいくつかのプロトコルが提案されている。自己安定プロトコルの利点としては、プロセス間の同期がずれた場合に例外処理ルーチンを用いずに解決できる点や、プロセスの初期化をネットワークのグローバルな状態を把握せずに行なえる点がある。しかし、自己安定プロトコルでは一般に、信頼性の向上のみが考慮されており、リアルタイムな回復は要求されない。リスポンシブプロトコルでは、このような自己安定性とリアルタイム性の両者を満足しなければならない。そこで、本稿では、自己安定性とリアルタイム性の両方の性質を持つ通信プロトコルであるリスポンシブプロトコルの検証法について議論する。プロトコル検証とは、プロトコル仕様がサービス仕様を正しく実現していることを証明することである。従って、サービス仕様として、自己安定性とリアルタイム性が満たされているとした場合のプロトコル検証がリスポンシブプロトコルの検証となる。

G.Gouda により異常状態から正常状態に至る過程の状態で成り立つ述語の系列の収束性を数学的に証明することにより自己安定性の検証を行なう方法[1]が提案されている。しかし、この方法ではリアルタイム性の検証はできない。文献[5]では、変数の値が有限と限定された場合のリスポンシブプロトコルの自動検証法を提案している。本稿では、この方法を一般化し、変数が無限の値をとることを許した場合の検証法を提案する。本検証法では、プロトコルの状態を述語で表し、任意の状態を表す述語からのすべての述語系列を求めてその述語の収束性を調べることで自己安定性の検証を行う。また、その述語列から異常状態から正常状態に回復するのに要する時間を求めることによりリアルタイム性の検証を行う。

以下では、2. で本論文で扱う拡張有限状態機械に基づく通信プロトコルとそれに関連する正常状態や異常状態などの諸定義を与える。3. ではリスポンシブプロトコルの必要

性と従来の検証法について述べる。4. では述語列のリアルタイム収束性に基づく検証法を提案する。5. では提案した検証法を自己安定双方向ハンドシェイクプロトコルに適用した検証例を示す。最後に、6. で本研究の結果と今後の課題について述べる。

2 通信プロトコル

2.1 通信プロトコルのモデル化

通信プロトコルは、プロセスとチャネルで構成される通信システムでのメッセージのやりとりを表す通信規約である。本節では、各プロセスを以下に定義する拡張有限状態機械でモデル化し、プロセス間のチャネルを FIFO キューで表すことにより通信プロトコルを定義する。

定義 1 プロセス P をモデル化した拡張有限状態機械 PM を次のような 7 項組 $(S, S_I, V, V_I, B, T, \delta)$ で定義する。

$S = \{S_1, \dots, S_n\}$: プロセス P のとり得る状態の有限集合。

S_I : プロセス P の初期状態で S の要素の 1 つ。

$V = \{v_1, \dots, v_m\}$: プロセス P の保持する変数の有限集合。

$V_I = \{v_{I1}, \dots, v_{Im}\}$: 各プロセス変数 v_i の初期値の集合。

$B = \{B_1, \dots, B_o\}$: 真偽値をとる述語の集合。

$T = \{T_1, \dots, T_p\}$: 命令の集合。各命令 T_k は受信命令 ($+message$) と送信命令 ($-message$) のいずれかとプロセス変数の計算を表す。

$\delta = S \times B \times T \rightarrow S$: プロセス P の状態遷移関数。状態 $S_i (\in S)$ で $B_j (\in B)$ が真で、かつ、命令 $T_k (\in T)$ が実行できるとき、遷移 δ は S_i で実行可能であるという。

定義 2 プロセス間のチャネル C は容量に制限のある FIFO キューで表される。以下では、チャネル C の容量を $|C|$ と書く。また、チャネル上の各メッセージは有限個のメッセージ変数の系列 $(mv_1, \dots, mv_m)$ で表わされる。

定義 3 定義 1, 2 に基づいて通信プロトコルを次のような 2 項組 (P, C) で定義する。

$P = \{PM_i | 1 \leq i \leq n\}$: 各 PM_i はプロセス P_i をモデル化した拡張有限状態機械であり、 n はプロセス数を表す。

$C = \{C_{ij} | i \neq j, 1 \leq i, j \leq n\}$: 各 C_{ij} はプロセス P_i と P_j の間のチャネルを表す。ただし、 $|C_{ij}| = 0$ のときは、 P_i から P_j へのチャネルが存在しないことを表す。

2.2 通信プロトコルのグローバル状態

本節では、通信プロトコルの状態を表すためにグローバル状態を定義する。また、グローバル状態を初期状態から到達可能な正常状態と到達不可能な異常状態に分類する。

定義 4 通信プロトコルのグローバル状態は、次の (1), (2) に定義するプロセスのローカル状態 PS とチャンネルのローカル状態 CS で表される。

- (1) プロセスのローカル状態 $PS = \langle S_i; pred \rangle$

この PS によって、プロセス変数に関する述語 $pred$ が成り立つような状態 S_i にプロセスがあることを表す。ただし、 $\langle S_i; true \rangle$ はプロセス変数には無関係にプロセスの状態が S_i であることを表すものとする。

- (2) チャンネルのローカル状態 $CS = \langle message_1: pred_1, \dots, message_m: pred_m \rangle$

この CS によって、チャンネル内の各メッセージ $message_i$ のメッセージ変数が $pred_i$ を満たすチャンネル状態であることを表す。ここで、 m はチャンネル内のメッセージ数 ($1 \leq m \leq |C|$) を表す。また、 $message: true$ はメッセージ変数に制約のない任意のメッセージを表し、 $CS = \langle \rangle$ はチャンネルが空であることを表すものとする。

各プロセス P_i のローカル状態を PS_i 、各チャンネル C_{ij} のローカル状態を CS_{ij} とすると、通信プロトコルのグローバル状態 $GS = (PS_1, \dots, PS_n, CS_{12}, \dots, CS_{nn-1})$ と表せる。

定義 5 グローバル状態の中で次の 2 つの条件 $C1, C2$ を満たすものを初期グローバル状態 (GS_0) もしくは、単に初期状態と呼ぶ。

条件 $C1 : \forall i [PS_i = \langle S_i; v_1 = v_{1I} \wedge \dots \wedge v_m = v_{mI} \rangle]$

条件 $C2 : \forall i, j (i \neq j) [CS_{ij} = \langle \rangle]$

グローバル状態 GS_i で、あるプロセスが $T_j (j \in T)$ により遷移 δ を実行可能で、遷移後のグローバル状態が GS_j である時、グローバル状態 GS_i からグローバル状態 GS_j へ遷移可能であると言う。

定義 6 正常グローバル状態を以下の (1), (2) のように再帰的に定義する。

- (1) 初期状態 GS_0 は正常グローバル状態である。
(2) 正常グローバル状態から遷移可能なすべての状態は正常グローバル状態である。

正常グローバル状態に含まれないすべてのグローバル状態を異常グローバル状態と定義する。以下では、簡単化のために正常グローバル状態と異常グローバル状態を、それぞれ、正常状態と異常状態と呼ぶ。

3 リスポンシブプロトコル

3.1 リスポンシブプロトコルの必要性

通信プロトコルは正常時にはプロトコル設計者の要求通りに正しく動作するように設計されており、正常状態から異常状態に遷移するとは考えない。従って、正常状態から異常状態に陥ったときの動作は保証されておらず、一度異常状態に陥ってしまうとそれ以後は正しく動作しない。

正常状態から異常状態に陥る原因としては、メッセージの転送エラーや遅延、プロセスの一時的故障、初期設定の誤りなどによりプロセス間の同期がくずれることがあげられる。

多くの通信プロトコルでは専用の例外処理ルーチンを用いてプロセス間の同期のずれを検出し、異常状態に陥っている場合、強制的に初期状態に戻すことにより正常状態に回復していた。この方法では、あらゆるエラーに対応するために多くの例外処理ルーチンを用意せねばならず、設計にコストがかかるという欠点がある。また、初期状態に強制的に戻してしまうので、異常状態に陥るまでに行なった処理が無駄になってしまう。

例外処理ルーチンを用いず、有限時間内に異常状態から正常状態に回復することが保証されている通信プロトコルは自己安定プロトコル [1, 2] と呼ばれる。これまでに双方向ハンドシェイクプロトコルやウィンドウプロトコルなどの実用的な通信プロトコルに自己安定性を持たせた自己安定プロトコルが提案されている。

しかし、それらのプロトコルの多くは故障から回復する機能としての自己安定性のみ考慮されており、正常状態に回復するまでの時間については考慮されていない。そこで、自己安定性を持ち、しかも、リアルタイムな回復が保証されているプロトコル (リスポンシブプロトコル [6] と呼ぶ) の開発が必要となってきた。

3.2 従来のリスポンシブプロトコルの検証法

リスポンシブプロトコルの検証では、そのプロトコルがリアルタイムで正常状態へと回復する性質 (リスポンシブ性と呼ぶ) を満たしているかどうかの検証が必要となってくる。

通常のリスポンシブ性の検証では、与えられた通信プロトコルが自己安定性を満たしているかどうかの検証が初めに行なわれる。その後、そのプロトコルが要求される時間内に正常状態へと回復するか否か (つまり、リアルタイム性をもつかどうか) の検証が行なわれる。

自己安定性の検証方法の提案としては G.Gouda らによる異常状態から正常状態に至る過程の状態で成り立つ述語の収束性判定を利用した方法がある。また、文献 [5] では、変数の値が有限の値をとる場合のリスポンシブプロトコル

の自動検証法が提案されている。

4 検証法

本稿では、文献 [5] の検証法を一般化し、変数が無限の値をとることを許した場合のレスポンスプロトコルの検証法を提案する。具体的には、まず、異常状態から正常状態への遷移系列をすべて列挙することにより異常状態を表す述語から正常状態を表す述語への述語系列の収束性、つまり、自己安定性の検証をする。次に、その述語系列に対し、異常状態に陥ってから正常状態に回復するまでに要する時間の最大値を求め、その値と要求されている時間との比較を行なうことによりリアルタイム性を検証する。

4.1 概略

検証に対する入力として次の (1) ~ (4) が与えられる。

- (1) プロトコルの仕様。仕様は拡張有限状態機械モデルに基づいて記述されている。
- (2) 正常状態の集合 $GS_{normal} = \{GS_{n1}, \dots, GS_{nm}\}$ 。本検証法では、正常状態を厳密にかつ簡潔に述語で表すことが重要である。
- (3) 要求される時間 R 。どの異常状態に陥っても R 時間以内に正常状態に回復する時、そのプロトコルはリアルタイム性を持つとする。
- (4) 一回の遷移に必要な時間 t 。プロトコル内の各遷移は時間 t 以内で実行されるとする。

検証手順の概要 (図 1 参照) を以下に示す。ただし、ステップ 1~3 の具体的な方法はそれぞれ 4.2~4.4 で示す。

[ステップ 1] 〈異常状態の列挙〉... 異常状態をすべて求める。実際には、対象とする異常状態だけを的確に求めることは困難なので、本稿では異常状態を含む任意の状態をすべて求める。

[ステップ 2] 〈自己安定性の検証〉... 各異常状態からの遷移系列をすべて求めて、状態遷移図を作成する。これらのすべての遷移系列がそれぞれある正常状態に到達可能であれば、自己安定性をもつと判定できる。この判定法については紙面の都合上省略する。本稿では状態遷移図の作成法のみを示す。

[ステップ 3] 〈リアルタイム性の検証〉... 作成した状態遷移図を利用して各異常状態から正常状態まで達するまでの時間 (回復時間と呼ぶ) の最大値を求め、それが R を越えるかどうかの検証を行なう。これらのすべての遷移系列の回

復時間が R 以下であれば、リアルタイム性を持つと判定できる。

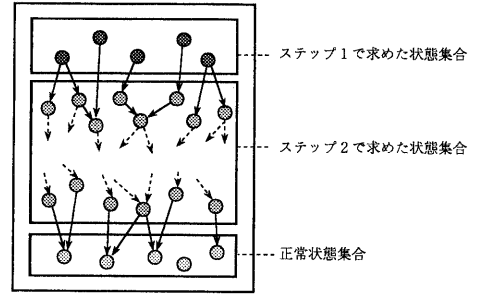


図 1 状態遷移図の作成

4.2 異常状態の列挙

(1) ローカル状態の列挙 ... プロセスとチャンネルごとにすべてのローカル状態の集合を以下のように求める。

- (i) プロセス P_i のローカル状態集合 PSI_i ... P_i のすべての状態 S_{ij} のローカル状態 $\langle S_{ij}:true \rangle$ を集合 PSI_i に加え、 $PSI_i = \{\langle S_{i1}:true \rangle, \dots, \langle S_{im}:true \rangle\}$ とする。
- (ii) チャンネル C_{ij} のローカル状態集合 CSI_{ij} ... チャンネル内のメッセージ数が $k (0 \leq k \leq |C_{ij}|)$ であるチャンネル C_{ij} のローカル状態を集合 CSI_{ij} に加える。ただし、各メッセージは $\langle mes:true \rangle$ とする。 $CSI_{ij} = \{\langle \rangle, \langle mes_1:true \rangle, \dots, \langle mes_1:true, \dots, mes_{|C_{ij}|}:true \rangle\}$ とする。

(2) 異常状態の列挙 ... (1) で求めたプロセスとチャンネルの各ローカル状態集合 $PSI_1, \dots, PSI_n, CSI_{12}, \dots, CSI_{nn-1}$ から状態を 1 つずつとり出して、グローバル状態 $gs = (PSI_1, \dots, PSI_n, CSI_{12}, \dots, CSI_{nn-1})$ をすべて求める。ここで、各 $PS_i \in PSI_i$, 各 $CS_{ij} \in CSI_{ij}$ とする。

4.3 自己安定性の検証

異常状態の集合より状態遷移図を作成し、自己安定性の検証を行なう。状態遷移図の作成方法について以下に述べる。

4.2 で求めたすべての異常状態を含む状態遷移図をまず作成し、次に、図中の各状態 (gs) から遷移可能な状態 (gs') を順次付加していくことにより状態遷移図を作成する。

状態 gs から遷移可能な状態 gs' の構成方法を以下に示す。今、状態 gs のプロセス P_i のローカル状態を $\langle S_j:pred_j \rangle$ とし、プロセス P_i の S_j における状態遷移関数を $S_j \times B_k \times T_i \rightarrow S_m$ と仮定する。このとき、以降の処理 (1) ~ (5) を各 $P_i (1 \leq i \leq n)$ について行なう。

- (1) gs が正常状態であれば、その状態からの遷移は実行しない。 gs が異常状態であれば、プロトコル内のプロセス P_i について、実行可能な遷移を調べ、その遷移の実行後のグローバル状態 gs' を以下の手順で求める。ただし、チャンネルの容量を越える遷移は考えない。
- (2) もし、 $pred_j$ の値域と B_k の値域の間に重なりがあれば、 gs' における P_i のローカル状態を $\langle S_m : pred_j \wedge B_k \rangle$ とする。ただし、受信命令や内部計算によってプロセス変数の値が変化して、それまで成立していた述語が成り立たなくなる場合、その述語を取り除く。
- (3) gs' の P_i 以外のプロセスの状態は gs の状態と同じにする。ただし、プロセス P_i の変数を含んだ述語が他のプロセス $P_l (l \neq i)$ のローカル状態に含まれており、かつ、その変数の値が P_i の遷移により変化した場合、そのプロセスのローカル状態の述語を書き換える。
- (4) gs' のチャンネルの状態は gs の状態と同じにする。ただし、チャンネル内のメッセージの述語にプロセス P_i の変数が含まれており、しかも、変数の値が遷移により変化した場合、その述語を書き換える。一方、送信命令を実行した場合、メッセージにそのメッセージ変数間で成り立つ述語を付加したものをチャンネル状態の最後に加える。受信命令を実行した場合には、チャンネルの先頭のメッセージを取り除く。
- (5) 最終的にできた gs' を状態遷移図に加える。 P_i のローカル状態 $\langle S_j : pred_j \rangle$ で、複数の実行可能な遷移がある場合には、すべての遷移可能な状態 gs' を求める。

状態遷移図中のすべての状態から正常状態に到達するまで、上述の操作を繰り返す。

4.4 リアルタイム性の検証

この節では、4.3 で作られた状態遷移図に基づいて、異常状態から正常状態へ至る遷移系列の長さ（以下、遷移数と呼ぶ）を求める方法と、遷移数から計算される回復時間を利用してリアルタイム性を検証する方法を示す。

状態遷移図中の各状態 $gs_i (1 \leq i \leq m)$ から正常状態に到達するまでの遷移数は以下の (1),(2) に示すように再帰的に求めることができる。ただし、 $succ(gs_i, T_j)$ は gs_i において $T_j (\in T)$ を実行することにより遷移可能な状態を表す。

- (1) gs_i のすべての $succ(gs_i, T_j)$ が正常状態であれば、 gs_i の遷移数を 1 とする。
- (2) 正常状態に属さない状態 $succ(gs_i, T_j)$ があれば、それらの状態の遷移数の最大値に 1 を加えたものを gs_i の遷移数とする。

上記の方法で求めた遷移数の計算においては、2 つ以上のプロセスの遷移の同時実行が仮定されていない。しかし、実際のプロトコルでは幾つかのプロセスが並列に動作している。並列動作を考慮したリアルタイム性の検証を行なうためには、並列に実行可能な遷移を同時に行なったときの遷移数を求める必要がある。このような並列性を考慮した遷移数はマルチプロセッサに対するタスクスケジューリングアルゴリズムを用いて求めることができる [5]。

最後に、並列性を考慮した遷移数に t を掛けて回復時間 R_i を求め、 R_i と R の大小を比較する。もし、任意の gs_i について $R_i \leq R$ を満たせば、そのプロトコルはリアルタイム性を満たすと判定する。

状態遷移図が与えられたときに並列性を考慮した遷移数の最大値を求めるアルゴリズムを図 2 に示す。

```

procedure find_max_step({ $gs_1, \dots, gs_m$ }:SET of state);
var sstep1, ..., sstepm : integer;
    pstep1, ..., pstepm : integer;
    max_step : integer;
begin
  for i := 1 to m do
    sstepi := 0;
    for i := 1 to m do
      step( $gs_i$ );
    for i := 1 to m do
      pstepi := parallel_step( $gs_i$ );
      max_step := max{pstepi | 1 ≤ i ≤ m};
    return(max_step);
  end;

  procedure step( $gs_i$ );
  begin
    if sstepi > 0 then
      exit;
    if  $\forall succ(gs_i, T_j) \in GS_{normal}$  then
      sstepi := 1
    else
      begin
        for  $\forall succ(gs_i, T_j) \notin GS_{normal}$  do
          step(succ( $gs_i, T_j$ ));
          sstepi := max{sstepk |  $gs_k = succ(gs_i, T_j)$ } + 1;
        end
      end;
  end;

  function parallel_step( $gs_i$ ):integer;
  begin
    タスクスケジューリングアルゴリズムを用いて並列性を
    考慮した遷移数 (pstepi) を求める;
    return(pstepi);
  end;

```

図 2 遷移数の計算

5 検証例

本章では、自己安定双方向ハンドシェイクプロトコル [1] を例にとって提案した検証法を説明する。

5.1 双方向ハンドシェイクプロトコル

双方向ハンドシェイクプロトコルは、2 つのプロセス P_1, P_2 の間のコネクションを確立するためのプロトコル

である。プロセス P_1 がコネクションを確立するためにプロセス P_2 に *set* を送る。 *set* を受けとった P_2 はコネクションを確立するのであれば *acpt* を、確立しないのであれば *rjct* を P_1 に返す。 P_1 は返事が *rjct* であればコネクションを確立せず、 *acpt* であればコネクションを確立する。コネクションを切るときは P_1 が *reset* が送り、それを受けとった P_2 は無条件に *ack* を返す。 P_1 が *ack* を受けとった時点でコネクションが切られる。

自己安定性を持つ双方向ハンドシェイクプロトコルは、G.Gouda より提案されている。このプロトコルでは、 P_1 が *set* や *reset* を P_2 に送るときこれらのメッセージに識別番号 (P_1 の変数 *cp* の値) を付けて送る。 P_2 は受けとったメッセージに対する返事を返すときに、受けとったメッセージの識別番号をそのまま付けて返す。 P_1 は P_2 からのメッセージを受けとったときに、その識別番号を見てそれが正しいメッセージかどうかの判定を行なっている。

この自己安定双方向ハンドシェイクプロトコルのプロセス P_1, P_2 の拡張有限状態機械モデルに基づく仕様を以下に示す。

(1) プロセス P_1 の仕様。

$S_1 = \{close, open, wait_1, wait_2, done_1, done_2\}$, $S_{1I} = close$, $V_1 = \{cp, t, u\}$, $V_{1I} = \{0, \perp, \perp\}$, 状態遷移関数 δ_1 は表 1 に示す。

S_1 の要素 *close* はコネクションが閉じている状態、 *open* はコネクションが確立されている状態、 *wait₁*, *wait₂* はコネクションが確立するのを待っている状態、 *done₁*, *done₂* はコネクションが閉じるのを待っている状態を表す。次に V_1 の要素 *cp* はメッセージにつける識別番号の値を保持する変数を表す。 $\perp$ は未定義を表す。状態遷移関数 δ_1 中の $TO(pred)$ は、 *pred* を満たす状態があらかじめ決められた時間より長く続くとき、かつ、そのときに限り真になる述語である。

(2) プロセス P_2 の仕様。

$S_2 = \{close, open, ready\}$, $S_{2I} = close$, $V_2 = \{v, w\}$, $V_{2I} = \{\perp, \perp\}$, 状態遷移関数 δ_2 は表 2 に示す。

S_2 の要素 *close* はコネクションが閉じている状態、 *open* はコネクションが確立されている状態、 *ready* はメッセージの応答を返すのを待っている状態を表す。

次に、正常状態の集合 GS_{normal} を次のように与える。

$$GS_{normal} = \{ \\ \langle \langle close: true \rangle, \langle close: true \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle wait_1: true \rangle, \langle close: true \rangle, \\ \langle \langle rqst, num \rangle: rqst = set \wedge num = cp \rangle, \langle \rangle \rangle, \\ \langle \langle wait_1: true \rangle, \langle ready: w = cp \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle wait_1: true \rangle, \langle close: true \rangle, \\ \langle \rangle, \langle \langle rply, num \rangle: rply = rjct \wedge num = cp \rangle \rangle, \\ \}$$

$$\langle \langle wait_1: true \rangle, \langle open: true \rangle, \\ \langle \rangle, \langle \langle rply, num \rangle: rply = acpt \wedge num = cp \rangle \rangle, \\ \langle \langle wait_2: t = rjct \wedge u = cp \rangle, \langle close: true \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle wait_2: t = acpt \wedge u = cp \rangle, \langle open: true \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle open: true \rangle, \langle open: true \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle done_1: true \rangle, \langle open: true \rangle, \\ \langle \langle rqst, num \rangle: rqst = reset \wedge num = cp \rangle, \langle \rangle \rangle, \\ \langle \langle done_1: true \rangle, \langle ready: w = cp \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle done_1: true \rangle, \langle close: true \rangle, \\ \langle \rangle, \langle \langle rply, num \rangle: rply = ack \wedge num = cp \rangle \rangle, \\ \langle \langle done_2: t = ack \wedge u = cp \rangle, \langle close: true \rangle, \langle \rangle, \langle \rangle \rangle \\ \}$$

例えば、状態 $\langle \langle open: true \rangle, \langle open: true \rangle, \langle \rangle, \langle \rangle \rangle$ は 2 つのプロセス P_1, P_2 間にコネクションが確立されている状態を表す。この状態では、プロセス P_1, P_2 の状態は共に *open* で、かつ、チャネルはすべて空である。しかし、プロトコル変数に関する述語は *true* なので、各変数 (*cp, t, u, v, w*) は任意の値が許される。つまり、1 つの述語で実際には無限個のプロトコル状態を表している。

5.2 検証例

自己安定双方向ハンドシェイクプロトコルがリスボンプロトコルであるか否かを検証する例を以下に示す。

[ステップ 1] 4.2 で示した方法で異常状態の集合を求める。ここでは、各チャネルの容量を 1 と仮定する。

(1) プロセス P_1, P_2 とチャネル C_{12}, C_{21} の任意の状態を表すローカル状態の集合は次の様に求まる。

$$PSI_1 = \{ \langle \langle close: true \rangle, \langle open: true \rangle, \langle wait_1: true \rangle, \\ \langle wait_2: true \rangle, \langle done_1: true \rangle, \langle done_2: true \rangle \rangle, \\ PSI_2 = \{ \langle \langle close: true \rangle, \langle open: true \rangle, \langle ready: true \rangle \rangle, \\ CSI_1 = \{ \langle \rangle, \langle \langle rqst, num \rangle: true \rangle \}, \\ CSI_2 = \{ \langle \rangle, \langle \langle rply, num \rangle: true \rangle \}. \\ \}$$

(2) 求めたすべての異常状態の集合を次に示す。

$$\langle \langle close: true \rangle, \langle close: true \rangle, \langle \rangle, \langle \rangle \rangle, \\ \langle \langle close: true \rangle, \langle close: true \rangle, \langle \rangle, \langle \langle rply, num \rangle: true \rangle \rangle, \\ \vdots \\ \langle \langle done_2: true \rangle, \langle ready: true \rangle, \langle \langle rqst, num \rangle: true \rangle, \langle \rangle \rangle, \\ \langle \langle done_2: true \rangle, \langle ready: true \rangle, \\ \langle \langle rqst, num \rangle: true \rangle, \langle \langle rply, num \rangle: true \rangle \rangle. \\ \}$$

これより全部で $6 \times 3 \times 2 \times 2 = 72$ 個のグローバル状態が定義できる。

[ステップ 2] 異常状態の集合から状態遷移図を作成する。

図 3 に状態 $gs_1 = (\langle \langle wait_1: true \rangle, \langle open: true \rangle, \langle \langle rqst, num \rangle: true \rangle, \langle \langle rply, num \rangle: true \rangle \rangle)$ から正常状態 (gs_{14}) への遷移系列の一例を示す。

状態 gs_1 では、 P_1 による $\langle \langle rqst, num \rangle$ の受信と P_2 による $\langle \langle rply, num \rangle$ の受信の 2 つの遷移が実行可能である。図 3 で

は後者を省略している。\$(rqst, num)\$を受信すると、\$P_1\$は状態は\$wait_2\$になり、チャンネル\$C_{21}\$が空になって状態\$gs_2\$に達する。続いて状態\$gs_2\$では、\$P_1\$による3つの遷移が考えられるが、図3には状態遷移関数(表1参照)の\$wait_2\$の2つ目の遷移を行なった場合を示している。\$P_1\$は状態\$close\$に遷移し、しかも、遷移に関する述語は\$(t = rjct \wedge cp = u)\$なので、\$P_1\$の遷移後のローカル状態は\$(close: t = rjct \wedge cp = u)\$となり、状態\$gs_3\$に達する。状態\$gs_4\$以降の状態も同様に求まり、最終的に正常状態\$(gs_{14})\$に達する。

```

gs1 : ((wait1: true), (open: true),
      ((rqst, num): true), ((reply, num): true))
gs2 : ((wait2: true), (open: true), ((rqst, num): true), ())
gs3 : ((close: t = rjct ∧ cp = u), (open: true),
      ((rqst, num): true), ())
gs4 : ((close: t = rjct ∧ cp = u), (ready: true), (), ())
gs5 : ((close: t = rjct ∧ cp = u), (open: v = set),
      (), ((reply, num): reply = acpt))
gs6 : ((wait1: t = rjct ∧ cp = u), (open: v = set),
      ((rqst, num): rqst = set ∧ num = cp),
      ((reply, num): reply = acpt))
gs7 : ((wait2: t = acpt), (open: v = set),
      ((rqst, num): rqst = set ∧ num = cp), ())
gs8 : ((open: t = acpt ∧ cp = u), (open: v = set),
      ((rqst, num): rqst = set ∧ num = cp), ())
gs9 : ((open: t = acpt ∧ cp = u), (ready: v = set ∧ w = cp), (), ())
gs10 : ((open: t = acpt ∧ cp = u), (close: v = set ∧ w = cp),
      (), ((reply, num): reply = rjct ∧ num = cp))
gs11 : ((done1: t = acpt ∧ cp = u), (close: v = set ∧ w = cp),
      ((rqst, num): rqst = reset ∧ num = cp),
      ((reply, num): reply = rjct ∧ num = cp))
gs12 : ((done1: t = rjct ∧ num = cp), (close: v = set ∧ w = cp),
      ((rqst, num): rqst = reset ∧ num = cp), ())
gs13 : ((done1: t = rjct ∧ num = cp), (ready: v = reset ∧ w = cp),
      (), ())
gs14 : ((done1: t = rjct ∧ num = cp), (close: v = reset ∧ w = cp),
      (), ((reply, num): reply = ack ∧ num = cp))

```

図3 遷移系列の一例

[ステップ3] 並列性を考慮した遷移数を求めリアルタイム性を判定する。

状態\$gs_1\$から正常状態に至る遷移系列\$(gs_1 \rightarrow \dots \rightarrow gs_{14})\$の遷移数は13である。しかし、図3の遷移系列において\$gs_2\$から\$gs_3\$へと\$gs_3\$から\$gs_4\$への2つの遷移は並列に実行可能である。同様に並列に実行可能な遷移を陽に抽出すると、\$gs_1\$から正常状態に至るまでの遷移系列は次のようになる。ここで、\$\Rightarrow\$は並列に実行可能な遷移を表す。

$gs_1 \Rightarrow gs_2 \Rightarrow gs_4 \Rightarrow gs_6 \Rightarrow gs_7 \Rightarrow gs_9 \Rightarrow gs_{11} \Rightarrow gs_{13} \Rightarrow gs_{14}$
従って、遷移数は8となる。

最後に、遷移数8に時間\$t\$を掛けて得られる回復時間\$R_1(=8t)\$と\$R\$の大小比較を行なう。すべての遷移系列について\$R_1 \leq R\$が成り立てば、5.1の双方向ハンドシェイクプロトコルはリスポンシブプロトコルであると判定できる。

6 あとがき

本稿では、通信プロトコルを拡張有限状態機械でモデル化した場合のリスポンシブ通信プロトコルの検証法を提案した。また、双方向ハンドシェイクプロトコルがリスポンシブプロトコルであるか否かの検証例を示した。提案した検証法の特徴は以下の通りである。

- (1) 正常状態を述語を用いて、厳密にかつ簡潔に述語に表現し、それらの状態の集合を入力として与えることができれば、通信プロトコルの自己安定性の自動検証が可能になる。
- (2) プロトコル検証のためのグローバル状態遷移図の作成とマルチプロセッサシステムのためのタスクスケジューリングアルゴリズムを応用して、異常状態に陥ってから正常状態に回復するまでに要する時間を求めることにより、通信プロトコルのリアルタイム性の自動検証が可能である。

今後の課題としては、実用的な各種通信プロトコルへの本検証法の適用がある。

参考文献

- [1] M. G. Gouda and N. J. Multari: "Stabilizing communication protocols", IEEE Trans. on Computers, Vol.40, No.9, pp.448-458, (April 1991).
- [2] N. J. Multari: "Toward a theory for self-stabilizing protocols", Ph.D. Dissertation, University of Texas, Austin (1989).
- [3] M. T. Liu: "Protocol engineering", Advances in Computers, 29, pp.79-195, (1989).
- [4] M. Malek: "Responsive systems(A challenge for the nineties)", Proc. EUROMICRO'90, 16th Symp. on Microprocessing and Microprogramming 30, pp.9-16, (1990).
- [5] Y. Kakuda and T. Kikuno: "Verification of responsiveness for communication protocols", IEICE Japan, Tech. Group Paper FTS91-57, CPSY91-58 (Dec. 1991).
- [6] Y. Kakuda, T. Kikuno and H. Saito: "A new design method of responsive protocols for communication systems", Int'l Workshop on Responsive Computer Systems, Golfe-Juan, France, (1991).
- [7] 角田, 菊野: "リスポンシブシステムと通信プロトコル", 実時間ワークショップ (March 1992).
- [8] 福村, 稲垣: "オートマトン・形式言語理論と計算論", 岩波書店 (1982).

表 1 プロセス P_1 の状態遷移関数 δ_1

状態	述語	命令	次の状態
$\langle close \rangle$		$-(set, cp)$	$\langle wait_1 \rangle$
$\langle close \rangle$		$+(rply, num)$	$\langle close \rangle$
$\langle open \rangle$		$-(reset, cp)$	$\langle done_1 \rangle$
$\langle open \rangle$		$+(rply, num)$	$\langle open \rangle$
$\langle wait_1 \rangle$	$TO(C_{pq} = 0 \wedge C_{qp} = 0)$	$-(set, cp)$	$\langle wait_1 \rangle$
$\langle wait_1 \rangle$		$+(rply, num), t := rply, u := num$	$\langle wait_2 \rangle$
$\langle wait_2 \rangle$	$t = acpt \wedge cp = u$		$\langle open \rangle$
$\langle wait_2 \rangle$	$t = rjct \wedge cp = u$	$cp := cp + 1$	$\langle close \rangle$
$\langle wait_2 \rangle$	$t = ack \wedge cp = u \vee cp \neq u$		$\langle wait_1 \rangle$
$\langle done_1 \rangle$	$TO(C_{pq} = 0 \wedge C_{qp} = 0)$	$-(reset, cp)$	$\langle done_1 \rangle$
$\langle done_1 \rangle$		$+(rply, num), t := rply, u := num$	$\langle done_2 \rangle$
$\langle done_2 \rangle$	$t = ack \wedge cp = u$	$cp := cp + 1$	$\langle close \rangle$
$\langle done_2 \rangle$	$t \neq ack \vee cp \neq u$		$\langle done_1 \rangle$

表 2 プロセス P_2 の状態遷移関数 δ_2

状態	述語	命令	次の状態
$\langle close \rangle$		$+(rqst, num), v := rqst, w := num$	$\langle ready \rangle$
$\langle open \rangle$		$+(rqst, num), v := rqst, w := num$	$\langle ready \rangle$
$\langle ready \rangle$	$v = set$	$-(acpt, w)$	$\langle open \rangle$
$\langle ready \rangle$	$v = set$	$-(rjct, w)$	$\langle close \rangle$
$\langle ready \rangle$	$v = reset$	$-(ack, w)$	$\langle close \rangle$