<<もくじ：Formal Verification Memo

Alternating Bit Protocol (ABP)

ABP とは

反転ビットプロトコル (Alternating Bit Protocol; ABP) は、メッセージの消失が起こり得る通信経路を想定して設計されたプロトコルのうち、最も単純かつ最も古いもののひとつ。

送信者 (sender) は、コントロールビットと呼ばれる 1-bit 情報を各メッセージに付加します。メッセージに付けるコントロールビットは、その直前のメッセージについていたビットの値を反転したものとします。送信者は、メッセージを一つ送信したら、同じコントロールビットを持つ応答メッセージを待ちます。もし、一定期間内に応答がなかったり、間違ったコントロールビットの応答メッセージが返ってきたら、送信者は繰り返し同じメッセージを送信します。これは、ただしい応答を受け取るまで繰り返されます。

受信者 (receiver) は、受け取った全てのメッセージに対して応答メッセージを返します。その際、応答メッセージには、受信メッセージについていたのと同じコントロールビットを付けて返します。さらに、受信者は、コントロールビットの確認を行います。いま受信したメッセージのコントロールビットと、ひとつ前のメッセージのものが同じだった場合には、そのメッセージを破棄します。ひとつ前のメッセージと異なるコントロールビットを持つメッセージは捨てずに受け取ります。

ABP は、メッセージやその応答の損失がどのように生じた場合であっても、正しく動作することを意図して設計されています。ここで、「正しく動作する」とは、受信者者によって受け取られたメッセージ列が、送信者が送ったものと同一であり、メッセージの消失、重複、および順番の前後がないことを指します。

非常にシンプルなプロトコルであるのにもかかわらず、ABP の検証は一筋縄ではいきません。たとえば、ACK が遅れて送られてくるケースに、送信者は ACK が届かないものとして動作するかも知れないとか。

Spin を用いた検証

Promela によるモデル記述 (abp_promela)

```
#define MAX 5
#define ERRMAX 2 /* Error 頻発による non-progress を除外するため */



mtype = {msg, ack, nack, corrupt};

proctype sender(chan in, out)
{
  int nerr;
  byte dat, cbit_s, r;

  nerr = 0;
  dat = MAX - 1;
  cbit_s = 0;
```

```
  do
  :: dat = (dat + 1) % MAX; /* entity of Next message */
again:
    if
    :: out!msg(dat, cbit_s);
    :: (nerr <= ERRMAX) -> nerr = nerr + 1; out!corrupt(0, 0);
    :: (nerr <= ERRMAX) -> nerr = nerr + 1; /* lost */
    fi;
    if
    :: timeout          -> goto again
    :: in?corrupt(0, 0) -> goto again
    :: in?nack(r, 0)    -> goto again
    :: in?ack(r, 0) ->
       if
       :: (r == cbit_s) -> goto progress
       :: (r != cbit_s) -> goto again
       fi
    fi;
progress:
    cbit_s = 1 - cbit_s; /* alternate bit */
  od
}

proctype receiver(chan in, out)
{
  int nerr;
  byte d, c; /* received data entity and control bit */
  byte d_expect;
  byte c_expect;

  nerr = 0;
  d_expect = 0;
  c_expect = 0;

  do
  :: in?msg(d, c) ->
       if
       :: (c == c_expect) ->
         assert(d == d_expect);
```

```
progress:
         c_expect = 1 - c_expect;
         d_expect = (d_expect + 1) % MAX;
         if
         :: out!ack(c, 0);
         :: (nerr <= ERRMAX) -> nerr = nerr + 1; out!corrupt(0, 0);
         :: (nerr <= ERRMAX) -> nerr = nerr + 1; /* lost */
         fi
       :: (c != c_expect) ->
         if
         :: out!nack(c, 0); /* why nack!? */
         :: (nerr <= ERRMAX) -> nerr = nerr + 1; out!corrupt(0, 0);
         :: (nerr <= ERRMAX) -> nerr = nerr + 1; /* lost */
         fi
       fi
  :: in?corrupt(0, 0) ->
    if
    :: out!nack(c, 0);
    :: (nerr <= ERRMAX) -> nerr = nerr + 1; out!corrupt(0, 0);
    :: (nerr <= ERRMAX) -> nerr = nerr + 1; /* lost */
    fi
  od
}

init {
  chan msg_ch = [1] of {mtype, byte, byte};
  chan ack_ch = [1] of {mtype, byte, byte};

  atomic {
    run sender(ack_ch, msg_ch);
    run receiver(msg_ch, ack_ch);
  }
}
```

上のソースは、Basic Spin Manual にあるものとほとんど同じです。
前に述べた ABP の説明と異なる点は、以下の二つ：

- メッセージの消失 (lost) に加えて、壊れ (corrupt) のケースが増えている
- 受信側は、コントロールビットが期待と違ったときに、なぜか NACK 応答している

前者は、通信路におけるメッセージ消失だけでなく、メッセージの壊れ（データ化け）にも対処するための

対処です。メッセージの符号化に誤り検出符号を用い、受信側によるエラー検出を行うことに対応します。これは、ABP に対する自然な拡張といえるでしょう。

後者は、不思議ですね〜。ABP 違反なんじゃないでしょうか。

検証プログラムの実行

まずはコンパイル。

```
% spin -a abp_promela
% gcc -DNP -o pan pan.c
```

今回は non-progress cycle の検出を行いたいので、gcc に -DNP オプションをつけてコンパイルしています。

実行してみます。

```
% ./pan
warning: never claim + accept labels requires -a flag to fully verify
hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     - (not selected)
        invalid end states      - (disabled by never claim)

State-vector 72 byte, depth reached 828, errors: 0
     5717 states, stored
     5590 states, matched
    11307 transitions (= stored+matched)
        2 atomic steps
hash conflicts:        13 (resolved)

    5.044       memory usage (Mbyte)
```

アサーション違反は無いようです。ただし、今回用いたアサーションでは、なんらかの理由でひとつもメッセージが送信完了せずに無限にリトライを繰り返すようなエラーを検出できません。

このような non-progress サイクルを検出するためには、pan を -l オプション付きで実行します。（それに先だって、コンパイル時に -DNP を指定している必要がある）

```
% ./pan -l
pan: non-progress cycle (at depth 25)
pan: wrote abp_promela.trail

(Spin Version 5.1.4 -- 27 January 2008)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 72 byte, depth reached 51, errors: 1
      108 states, stored (199 visited)
      116 states, matched
      315 transitions (= visited+matched)
        2 atomic steps
hash conflicts:         0 (resolved)

    4.653       memory usage (Mbyte)
```

エラートレースを見てみます：

```
% spin -t -r -s abp_promela
spin: couldn't find claim (ignored)
 11:    proc  1 (sender) line  20 "abp_promela" Send msg,0,0    -> queue 1 (out)
 13:    proc  2 (receiver) line  51 "abp_promela" Recv msg,0,0  <- queue 1 (in)
 23:    proc  2 (receiver) line  60 "abp_promela" Send corrupt,0,0      -> queue 2 (out)
 25:    proc  1 (sender) line  26 "abp_promela" Recv corrupt,0,0        <- queue 2 (in)
  <<<<<START OF CYCLE>>>>>
 27:    proc  1 (sender) line  20 "abp_promela" Send msg,0,0    -> queue 1 (out)
 29:    proc  2 (receiver) line  51 "abp_promela" Recv msg,0,0  <- queue 1 (in)
 33:    proc  2 (receiver) line  65 "abp_promela" Send nack,0,0 -> queue 2 (out)
 35:    proc  1 (sender) line  27 "abp_promela" Recv nack,0,0   <- queue 2 (in)
spin: trail ends after 35 steps
#processes: 3
 35:    proc  2 (receiver) line  50 "abp_promela" (state 37)
 35:    proc  1 (sender) line  19 "abp_promela" (state 11)
 35:    proc  0 (:init:) line  87 "abp_promela" (state 4) <valid end state>
```

```
3 processes created
```

ひとたび ACK 応答が壊れてしまうと、あとはメッセージの再送と NACK 応答が際限なく繰り返されてしまっています。

メモ

上のコーディングでは、発生するエラーの回数に上限を設けています。なにも制限しないと、ひたすら通信路上のエラーが発生するせいで処理がすすまないことを示す non-progress cycle が大量発生します。

Basic Spin Manual には、このようなエラーサイクルを除外する方法が書かれているのですが、なんだか良くわからないし、うまくいきませんでした。

今回は、たった一度の通信路エラーによって non-progress cycle に陥ってしまうことが確認できれば十分なので、このような対処でいいかな。

このプロトコルの良くないところは、やはり、コントロールビットが期待値と違ったときに NACK を返しているところでしょう。

```
        :: out!nack(c, 0); /* why nack!? */
```

と書かれているところを

```
        :: out!ack(c, 0);
```

に直すと正しいプロトコルになる筈です。

あと、Spin/Promela における timeout の扱いについて、少し気になります。ABP の検証を難しくする要因のひとつに、タイムアウトがあると思います。Receiver が応答を返し、伝送路上での消失がなかったにも関わらず、Sender がタイムアウト判定してしまうケースが生じ得ます。

上のモデルでは、条件選択のなかに timeout から始まるシーケンスを設けてありますが、これが期待通りのモデリングになっているかどうか、若干不安です。Basic Spin Manual を読むと、「timeout キーワードは、システムがハングした状態からの脱出を Promela でモデル化するたののものなのです」とあり、「もうちょっと待てば応答がくるのに、一足早くタイムアウト検出してしまった」というケースが対象になっているのかどうか、よくわかりません。

ちょっと実験してみましょう:

```
proctype r(chan ch)
{
  int to_count;

  to_count = 0;

  do
  :: timeout -> printf("timeout: %d\n", to_count)
  :: ch?0 -> skip
  od
```

```
}

proctype s(chan ch)
{
  do
  :: ch!0
  od
}

init
{
  chan ch = [1] of {int};
  atomic {
    run s(ch);
    run r(ch);
  };
}
```

上のソースを toex という名前で作成します。そして、コンパイル→実行してみました。

```
% spin -a toex
% gcc -o pan pan.c
% ./pan

hint: this search is more efficient if pan.c is compiled -DSAFETY

(Spin Version 5.1.4 -- 27 January 2008)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 40 byte, depth reached 6, errors: 0
        6 states, stored
        4 states, matched
       10 transitions (= stored+matched)
        1 atomic steps
hash conflicts:         0 (resolved)
```

```
    4.653       memory usage (Mbyte)

unreached in proctype r
        line 8, state 3, "printf('timeout: %d\n',to_count)"
        line 11, state 9, "-end-"
        (2 of 9 states)
unreached in proctype s
        line 18, state 5, "-end-"
        (1 of 5 states)
unreached in proctype :init:
        (0 of 4 states)

pan: elapsed time 0 seconds
```

やはり、timeout は成立しないようですね。まあ、当然といえば当然のような気もします。
以下のように、若干小細工しても、同じでした。

```
proctype s(chan ch)
{
  int skip_ct;

  skip_ct = 0;

  do
  :: ch!0; skip_ct = 0;
  :: (skip_ct < 100) -> skip_ct++; skip
  od
}
```

つまり、上でモデル化された ABP において、タイムアウトになるのは、伝送路上で消失したケースのみとなります。

これが ABP のモデル化として適切なのかどうか、ちょっと良くわかりません。データリンク層のプロトコルだと思えば、隣接ノード間の転送レンテンシが予測できないなんて想定は必要ないのかもしれないし。

ちょっと保留。