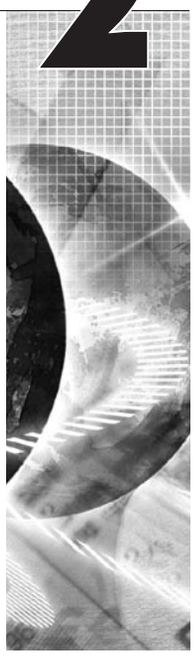
2

形式的手法による高信頼性組み込みソフトウェア開発



abstract

組み込みソフトウェアには高い信頼性が要求されるため，それを実現することができる形式的手法に対する期待が高まっている．形式的手法にはさまざまなものがあり，古くから研究されてきた．そこで本稿では，まず，形式的手法の概要について紹介する．そして，それらの中でも組み込みソフトウェア開発に有望と思われるモデル検査手法に焦点を当てて，その詳細を具体例を示しながら解説する．

青木 利晃
北陸先端科学技術大学院大学
安心電子社会研究センター
toshiaki@jaist.ac.jp

形式的手法の概要

組み込みソフトウェアは通信端末，家電機器，自動車などさまざまな製品に組み込まれており，今日の社会生活に欠かせない構成要素となっている．そのため，組み込みソフトウェアの誤りは日常生活や経済活動を混乱させ，莫大な時間的，金銭的損失を引き起こす可能性があり，高い信頼性を実現しなければならない．高信頼性を実現する1つのアプローチは，ソフトウェア開発において，数学を基礎とした言語やツールを用いて，対象となるソフトウェアを記述し検証すること，すなわち，形式的手法を用いることである．

形式的手法にはさまざまなものがあり，古くは，1960年代に提案されたプログラム検証手法がある．フローチャートや手続きプログラムに成立して欲しい表明を導入して，それが本当に成立することを証明するのである．その後，厳密に仕様を記述するための形式的仕様記述，検証技法であるモデル検査手法や定理証明手法などが提案されてきた．

◆形式的仕様記述

形式的仕様記述では，数学的に裏付けのある概念に基づいた言語を用いて厳密に仕様を記述する．Z, VDM, Bなどが代表的な手法である．Zは集合論に基づいて仕様を記述するための言語であり，集合，関係，関数，および，その上の制約を書くための記法が揃えられている．また，仕様をモジュール化するためのスキーマと呼ばれる概念も導入されている．このような言語を用いると，対象の仕様を厳密に解釈する必要が生じ，結果として仕様の理解や品質が向上することになる．また，記述した仕様からプログラムやテストケースを生成する手法，および，仕様の正しさをモデル検査手法や定理証明手法を用いて検証する手法なども提案されている．

◆モデル検査手法

モデル検査手法は，有限状態で特徴づけられる振る舞いを網羅的に探索し，不具合を発見するものである．不具合が発見された場合，反例，すなわち，そのような状況へ導く動作列を出力する．モデル検査手法では，自動的に検査できる反面，網羅的に探索するため，非常に多くの状態の数を探索しなければならなくなる状態爆発問題が生じる可能性がある．そのため，多くのモデル検査手法の実装では，効率的なデータ構造や探索最適化手法を用いている．モデル検査手法を実装したさまざまなツールがあり，Spin, SMV, LTSAが代表的なものである．

◆定理証明手法

定理証明手法では，述語論理などの形式的体系を用いた証明を行う．そのため，本質的に無限の状態を含むものを取り扱うことができるが，自動化が困難である．ゲーデルの不完全性定理では，算術計算を含む述語論理では，完全な体系（すべての正しい事実が証明できる体系）がないことが証明されている．一方で，Presburger Arithmeticのように，算術計算の範囲を限定することにより完全で決定可能（自動的に計算可能）なアルゴリズムが提案されている．定理証明手法を実装したシステムには，大別して，なるべく証明を自動化することを目指したものと，対話的な証明を支援するものがある．代表的なものとして，前者には，ACL2, Eves, LP, 後者には，Isabell, HOL, Coqなどがある．

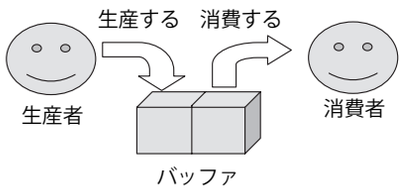


図-1　生産者－消費者問題

本稿では，これらの形式的手法のうちモデル検査手法に焦点を当てて解説をする．モデル検査手法は，組み込みソフトウェア開発で重要なタイミングの問題や共有リソース管理，並行性といった概念を取り扱えるからである．さらに，他の手法と比べて，前提となる知識の量が少なく，短期間の訓練で使えるようになるため，産業界への形式的手法導入のファーストステップとして適していると考えている．

モデル検査手法

以下では，モデル検査ツールSpinを用いて解説する．Spin[1]はAT&Tベル研究所により提案されたツールである．検証する振る舞いはPromelaと呼ばれる仕様記述言語により，チャネルを通じてメッセージを送受信しながら動作する並行プロセスを記述する．検査する性質は，ラベル，表明，性質オートマトンで記述する．また，時相論理の1つであるLTL（Linear Temporal Logic）により直感的かつ柔軟に性質を記述し検査することもできる．時相論理とは，時間経過により状況が変化する世界を扱う論理であり，さまざまな種類のものが提案されている．その1つであるLTLでは，∧，∨，¬，→ のような命題論理で扱う演算子のほかに，□, ◇, X, Uといった時間経過による状況の変化を指定できる．「□P」と書くと，時間経過しても常にPが成立するという意味になる．同様に，「◇P」はいつかはPが成立する，「X P」は次の時間でPが成立する，「P_1 U P_2」はP_2が成立するまでP_1が成立するという意味である．これらを組み合わせて，「□(P → ◇Q)」のような性質を記述する．この論理式は，時間経過のどの時点においてもPが成立するとそこから先にいつかはQが成立するという意味である．

例として，**図-1**のようなバッファが2つある生産者－消費者問題を考える．生産者はバッファがいっぱいになるまでアイテムを生産でき，消費者はバッファが空になるまでアイテムを消費できる．生産者と消費者の振る舞いをPromelaで記述したものを**図-2**に示す．この記述では，2つのプロセスproducerとconsumerが作成され，それぞれ，生産者と消費者を表現している．また，バッファの状況は32bit整数型の変数cntで表現している．0の時がバッファが空，1の時がバッファが1つだけ埋まっている，2の時が一杯である．それぞれのプロセスの振る舞いはブラケットの中に書かれており，バッファの状況を見ながら生産するか消費するかを決めている．プロセスproducerはcntが2未満，すなわち，バッファが空か1つだけ埋まっている場合は，1つアイテムを生産してcntをインクリメントする．cntが2以上の時は，2未満になるまで，条件（cnt<2）のところでブロックする．プロセスconsumerは，バッファが0より大きい場合，すなわち，バッファが1だけ埋まっているか，一杯の場合は，1つアイテムを消費してcntをデクリメントする．cntが0以下の時は，同様に，条件（0<cnt）が成立するまでブロックする．ここで，バッファ溢れが起きないことを，cntの値が0以上，2以下になっているかどうか調べることにより検証してみる．Promelaでは，assertという宣言を用いてその時点で成立すべき表明を記述する．図-2では，それぞれのプロセスの，インクリメント，および，デクリメントされた後に，cntの値が0以上，2以下であるという表明を入れてある．

```
int cnt = 0;

active proctype producer(){
 loop:
  (cnt < 2) ->
  /* 1つアイテムを生産 */
  cnt++;
  assert(0 <= cnt && cnt <= 2);
  goto loop
}

active proctype consumer(){
 loop:
  (0 < cnt) ->
  /* 1つアイテムを消費 */
  cnt--;
  assert(0 <= cnt && cnt <= 2);
  goto loop
}
```

図-2　生産者－消費者問題のPromelaによる記述

この記述では，さまざまな実行順序が考えられる．たとえば，

1：プロセスproducerが1つアイテムを生産
2：プロセスproducerが1つアイテムを生産
3：プロセスproducerが条件のところでブロック
4：プロセスconsumerが1つアイテムを消費
5：プロセスproducerのブロックが解除され1つアイテムを生産
...

```
Starting producer with pid 0
Starting producer with pid 1
Starting consumer with pid 2
Starting consumer with pid 3
  1:    proc  1 (producer) line   5 "mutex2.spin" (state 1)     [((cnt<2))]
  2:    proc  1 (producer) line   7 "mutex2.spin" (state 2)     [cnt = (cnt+1)]
  3:    proc  3 (consumer) line  14 "mutex2.spin" (state 1)     [((0<cnt))]
  4:    proc  2 (consumer) line  14 "mutex2.spin" (state 1)     [((0<cnt))]
  5:    proc  3 (consumer) line  16 "mutex2.spin" (state 2)     [cnt = (cnt-1)]
  6:    proc  2 (consumer) line  16 "mutex2.spin" (state 2)     [cnt = (cnt-1)]
spin: trail ends after 6 steps
#processes: 4
                cnt = -1
  6:    proc  3 (consumer) line  17 "mutex2.spin" (state 3)
  6:    proc  2 (consumer) line  17 "mutex2.spin" (state 3)
  6:    proc  1 (producer) line   8 "mutex2.spin" (state 3)
  6:    proc  0 (producer) line   5 "mutex2.spin" (state 1)
4 processes created
```

図-3　出力された反例

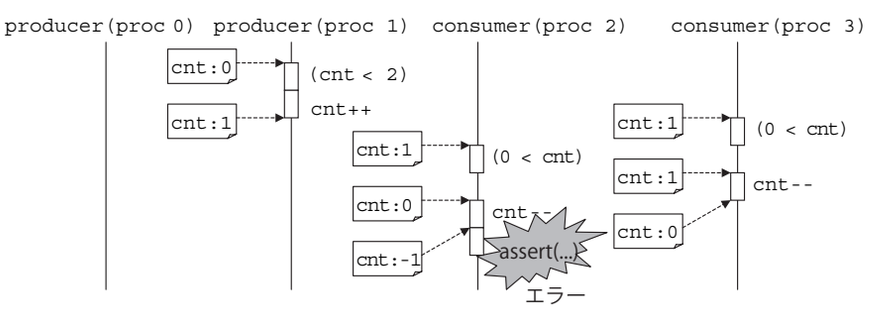


図-4　図-3の反例が示す振る舞い

1：プロセスproducerが1つアイテムを生産
2：プロセスconsumerが1つアイテムを消費
3：プロセスproducerが1つアイテムを生産
4：プロセスconsumerが1つアイテムを消費
...

などである．モデル検査では，このような考えられ得るすべての実行順序で，指定した性質(この例では表明)が成立するかどうかを調べる．図-2の例では，これらのプロセスは無限に動き続けるが，変数cntの取り得る値は有限個であり，実行時点の情報を加味しても有限個の実行パターンしか存在しない．よって，それらを網羅的に調べることにより，表明が指定した時点で常に成立するかどうか検証できるのである．実際にSpinによりモデル検査を実行してみると，エラーは報告されず，どのように実行したとしても，表明が成立することが保証された．この検査は，ほぼ，一瞬で終了する．

次に，生産者と消費者を，それぞれ2つずつ，計4つのプロセスを作成してみる．図-2のPromela記述のプロセスproducerとconsumerのactive proctype ...のところを，active[2] proctype ...と変更するのである．これにより，producerとconsumerのプロセスは2つずつ作成される．ここで，同様にSpinによりモデル検査を実行すると，エラーが報告される．表明が成立しない実行順序が存在するのである．このようなエラーが発見された場合反例が出力され，それを調べることによりエラーが発生する状況が明らかになる．この例で出力された反例を**図-3**に，それが意味する振る舞いを**図-4**に示す．プロセスにはIDが割り当てられており，proc 0，proc 1がproducerのプロセス，proc 2，proc 3がconsumerのプロセスである．まず，producerの1つのプロセスがcntをインクリメントする．そして，consumerの1つのプロセスが条件(0<cnt)をパスする．その直後に，実行権がconsumerのもう1つのプロセスに移り，その条件(0<cnt)をパスする．この時点で，consumerの2つのプロセスが両方とも条件をパスしたのである．そして，それぞれのプロセスがcntをデクリメントして，最終的に，cntの値が-1になり，表明が破られる．

このように，モデル検査では，非決定性や並行性を含む振る舞いにおいて，考えられるすべての場合を網羅的に探索して，指定した性質が成立するかどうかを自動的に調べる．**図-5**にモデル検査を用いた振る舞いの一般的な検証法についてまとめた．まず，検証対象の振る舞いにおいてエラーが報告されなければ，指定した性質が成立しないことはあり得ず，安心して，その振る舞いに基づいて実装などを行うことができる．エラーがあると反例が出力されそれを解析することによりエラーの原因を発見する．エラーを修正すると，再度，モデル検査を実行し，エラーがなくなるまで，モデル検査の実行と記述の修正を繰り返す．

組み込みソフトウェアの検証への応用

これまでに紹介した代表的なモデル検査ツールであるSpinは，比較的，組み込みソフトウェアの検証への応用が容易であると考えている．組み込みソフトウェアはマルチタスクを用いて実装することが多いが，その振る舞いがSpinの並行プロセスと似ているからである．しかし

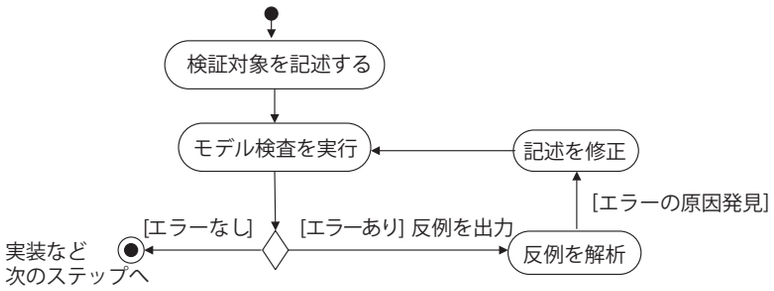


図-5　モデル検査による振る舞いの検証法

```
#define N 2
int count = 0;
producer(){
  while (TRUE){
  produce_item();
  if (count == N) sleep();
  enter_item();
  count=count+1;
  if (count == 1) wakeup(consumer);
  }
}

consumer(){
  while(TRUE){
    if (count == 0) sleep();
    remove_item();
    count = count - 1;
    if (count == N - 1) wakeup(producer);
    consume_item()
  }
}
```

図-6　資源管理の例(Cのプログラム)

ながら，Promelaで記述できること，および，探索できる状態空間は限られており，直接的に設計書やプログラムの検証を行うことは困難である．よって，検証したい性質がSpinで調べられるように工夫をしてPromelaで記述する必要がある．以下で，それらのいくつかについて紹介する．

◆競合状態の検出

図-6のCのプログラムを考える．これは，タネンバウムの教科書に載っている，sleep/wakeupによる資源管理の例である（文献2）p.76参照）．これまでに紹介した生産者－消費者問題と同様，2つのバッファがある生産者－消費者問題であり，組み込みソフトウェア開発でも頻繁に出現する典型的な資源管理のメカニズムである．関数sleepは自タスクをスリープさせ，関数wakeupは指定したタスクを起床させる．正しく見えるプログラムであるが，実は，デッドロックしてしまう可能性がある．さて，原因が分かるであろうか？ Spinで検証してみる．まず，Promelaで記述する．問題は，sleep/wakeupで

```
#define N 2
byte count = 0;
bool p_flag = true;
bool c_flag = true;

inline sleep_p(){p_flag = false}
inline sleep_c(){c_flag = false}
inline wakeup_p(){p_flag = true}
inline wakeup_c(){c_flag = true}

active proctype producer() provided (p_flag){
 again:
   if
   :: (count == N) -> sleep_p()
   :: else
   fi;
   /* put item in buffer */
   count = count + 1;
   if
   :: count == 1 -> wakeup_c()
   :: else
   fi;
   goto again
}

active proctype consumer() provided (c_flag){
 again:
   if
   :: (count == 0) -> sleep_c()
   :: else
   fi;
   /* remove item from buffer */
   count = count - 1;
   if
   :: (count == N -1) -> wakeup_p()
   :: else
   fi;
   goto again
}
```

図-7　図-6のCプログラムのPromelaによる表現

ある．Promelaは，そのようなプリミティブは持っていないので，存在するものを組み合わせて実現する必要がある．Promelaでは「provided（条件）」をプロセスの宣言に追加することができる．これは，条件が成立する時のみ，そのプロセスが実行可能であり，それ以外は実行されない．この仕組みを用いてsleep/wakeup を実現する．図-6のCのプログラムの振る舞いをPromelaで実現したものを**図-7**に示す．inline sleep_p()は，プ

```
Starting producer with pid 0
Starting consumer with pid 1
  1:    proc  1 (consumer) line  29 "sw.spin" (state 1) [((count==0))]
  2:    proc  0 (producer) line  15 "sw.spin" (state 4) [else]
  3:    proc  0 (producer) line  18 "sw.spin" (state 7) [count = (count+1)]
  4:    proc  0 (producer) line  20 "sw.spin" (state 8) [((count==1))]
  5:    proc  0 (producer) line   9 "sw.spin" (state 9) [c_flag = 1]
  6:    proc  1 (consumer) line   7 "sw.spin" (state 2) [c_flag = 0]
  7:    proc  0 (producer) line  15 "sw.spin" (state 4) [else]
  8:    proc  0 (producer) line  18 "sw.spin" (state 7) [count = (count+1)]
  9:    proc  0 (producer) line  21 "sw.spin" (state 11)        [else]
 10:    proc  0 (producer) line  14 "sw.spin" (state 1) [((count==2))]
 11:    proc  0 (producer) line   6 "sw.spin" (state 2) [p_flag = 0]
spin: trail ends after 11 steps
#processes: 2
                count = 2
                p_flag = 0
                c_flag = 0
 11:    proc  1 (consumer) line  33 "sw.spin" (state 7)
 11:    proc  0 (producer) line  18 "sw.spin" (state 7)
```

図-8　出力された反例

ロセス記述のsleep_pをインライン展開するものである．if ... fiは条件分岐であり，::の直後にあるのが条件で，elseは，列挙されている条件のいずれにもマッチしない場合に選択される．詳細は説明しないが，図-6のCのプログラムを実現していることが読み取れると思う．この記述をSpinにより検証すると，デッドロックを検出する．**図-8**に出力された反例を示す．まず，consumerがcountを参照して，バッファが空なので，sleepしようとする．しかしながら，条件（count == 0）をパスしただけで，sleepを実行する前に実行権がproducerに移っている．producerでは，バッファは空なのでアイテムを生産して，consumerを起床させようとしているが，consumerはまだスリープしておらず，意味のないwakeupの実行になっている．そして，consumerに実行権が移り，ここで，sleepが実行されスリープ状態になる．その後，producerがアイテムを生産し続けて，バッファが一杯になるとsleepする．この時点で，consumerとproducerの両方ともスリープ状態になってしまい，デッドロックになるのである．これは，競合状態が発生する典型的な例である．一見正しそうなプログラムも，モデル検査を実行すると誤りを含む場合がよくある．人間は，正常な実行パスは発見するのが得意であるが，このようなすべての場合を調べるのは不得意であり，それが得意なモデル検査ツールなどにまかせるべきである．

教科書にはセマフォを用いた解決策も書かれている（文献2）p.78参照）．**図-9**にそのプログラムを示す．この解決策では3つのセマフォを使用する．fullはアイテムが入っているバッファの数，emptyは空のバッファの数を，それぞれ数える．mutexは排他制御のために用いている．**図-10**はPromelaによる実現である．セマフォの

```
#define N 2
typedef int semaphore;
semaphore mutex = 1;
semaphore emp = N;
semahphore ful = 0;
producer(){
  int item;
  while(TRUE){
    produce_item(&item);
    down(emp);
    down(mutex);
    enter_item(item);
    up(mutex);
    up(ful);}
}

consumer(){
  int item;
  while (TRUE){
    down(ful);
    down(mutex);
    remove_item(&item);
    up(mutex);
    up(emp);
    consume_item(item);
  }
}
```

図-9　セマフォによる解決策（Cのプログラム）

down操作は，値が0より大きい場合はデクリメントして，そうでない場合は0より大きくなるまでブロックするように実現している．up操作は，値をインクリメントするだけである．また，この記述では，バッファが溢れないことを確認するため，変数cntを導入して，生産する時にインクリメント，消費する時にデクリメントしている．そして，バッファ溢れが発生しないことを確認する表明を追加した．モデル検査を実行すると，デッドロックは発生せず，バッファ溢れも発生しないことが確認できる．

```
#define N 2

#define semaphore byte

semaphore mutex = 1;
semaphore emp = N;
semaphore ful = 0;
int cnt = 0;
inline up(x){
  x++
}

inline down(x){
 d_step{(0 < x) -> x--}
}

inline put(){
  cnt ++;
  assert( cnt <= N)
}

inline remove(){
  cnt --;
  assert( 0 <= cnt )
}

active proctype producer(){
 again:
   down(emp);
   down(mutex);
   put();
   up(mutex);
   up(ful);
   goto again
}

active proctype consumer(){
 again:
   down(ful);
   down(mutex);
   remove();
   up(mutex);
   up(emp);
   goto again
}
```

図-10　図-9のCプログラムのPromelaによる表現

```
#include "rtoslib.spin"

#define P1  1
#define P2  2
#define P3  3

proctype low() provided (turn == P1) {
  do
  :: wai_sem(0,P1);
     /* critical section */
    printf("P1\n");
    sig_sem(0)
  od
}

proctype mid() provided (turn == P2){
  do
  :: printf("P2\n");
  od
}

proctype high() provided (turn == P3){
  do
  :: wai_sem(0,P3);
    printf("P3\n");
progress: sig_sem(0);
    yield(P1)
  od
}

init{
  ini();

  cre_tsk(3,P1);
  cre_tsk(2,P2);
  cre_tsk(1,P3);
  cre_sem(0,1);

  act_tsk(P1);
  act_tsk(P2);
  act_tsk(P3);

  run low();
  run high();
  run mid()
}
```

図-11　優先度逆転問題

◆スケジューリングの取り扱い

組み込みソフトウェアではμITRON[3]のような優先度付きマルチタスクが扱えるRTOS（Real-Time Operating Systems）を使用する場合が多い．そのようなタスクの振る舞いをSpinで検証する場合，RTOSで実現されている複雑なスケジューリングをPromela記述で実現しなければならない．図-7で実現したsleep/wakeupや図-10 のセマフォは，単純であったため，それらの実現は容易であった．しかしながら，μITRONのようなRTOSでは優先度キューやさまざまな情報を用いて，タスクの実行を制御している．そのような振る舞いを単純なPromelaを用いて実現できるのか，さらには，実現できたとして，深刻な状態爆発問題を引き起こさないか心配である．そこで，μITRONのスケジューリングをシミュレートするライブラリをPromelaで作成した[4]．このライブラリでは，μITRON4.0の仕様に基づいて，そのCによる実装であるTOPPERSのソースコードを参照しながら，優先度キューやTCB（Task Control Block）などのデータ構造とそれらの操作を実現した．そして，cre_tsk（タスク生成），act_tsk（タスクの起動要求），slp_tsk（タスクを起床待ち状態へ移行）といったタスク管理のサービスコールや，wai_sem（セマフォ資源獲得），sig_sem（セマフォ資源返却）といったタスク同期・通信のサービスコールを実現した．実装したサービスコールは約30で，Promela記述のサイズは約1,200行であった．その結果，μITRON のタスクの振る舞いをSpinでシミュレートすることができた．このライブラリを用いて優先度逆転問題が生じる振る舞いを記述したものを**図-11**に示す．

```
...
4:      high(3):[wai_sem(0 ,2);now.turn=top();]
P3      high(3):[printf('P3\n')]
28      high(3):[sig_sem(0);now.turn=top();]
30      high(3):[yield(0)]
32 low(2):[wai_sem(0,0);now.turn=top();]
34      high(3):[wai_sem(0,2);now.turn=top();]
<<<<<START OF CYCLE>>>>>
P2
36:  mid(4):[printf('P2\n')]
...
```

図-12　優先度逆転問題の反例

優先度が一番低いlow, 中間のmid, 一番高いhighの3つのタスクが存在し，lowタスクとhighが資源を共有している．優先度逆転は次の実行順で発生する．

1：lowタスクが資源(wai_sem)を獲得する．
2：highタスクに実行権が移る．
3：highタスクが資源を獲得しようとするが，lowが獲得しているためブロック．
4：midタスクが動作し続ける．

Spinによるモデル検査を実行すると，この優先度逆転問題を検出できる．出力された反例を**図-12**に示す．図-11のyield(P1)は一時的に実行権をlowにゆずる命令である．一時的に実行権をタスクlowにゆずり，lowがセマフォの資源を獲得し，実行権がhighに移り，その後，midの実行が続いている様子が分かる．また，優先度逆転問題はPCP（Priority Ceiling Protocol）などを用いれば解決できることが知られている．PCPを実装して優先度逆転問題が発生しないことも確認できた．優先度逆転問題は，NASAの火星探査機Mars Pathfinderのシステムでも発生した問題である．解決策は知られているが，検出が困難な問題と言える．このような問題も，モデル検査技術を用いることにより検出，解析が可能である．

普及へむけて

形式的手法にはさまざまなものがあることは最初に述べた．その中でもモデル検査手法は習得するのが容易で，かつ，使い始めてすぐにメリットが感じられ，比較的導入がしやすい技術であると考えている．モデル検査手法では有限状態で特徴づけられる振る舞いに限定されているので，そのうち限界が来るであろう．その限界を感じて，その他の手法や，より高度な理論に挑戦してみるのも良いと思う．一方で，モデル検査ツールの使い方を習得した技術者から，どのように組み込みソフトウェア開発に応用すればよいか分からないという声も聞く．その解決策は，ノウハウの蓄積であろう．ノウハウは企業の武器であり，努力の賜であるから，公開はされないと思う．地道に企業内でノウハウの蓄積活動をする以外にはないであろう．ひょっとすると海外の企業でそのような活動がされているかもしれない，という危機感を最近持っている．日本の国際競争力を維持，向上させていくためには，日本でも積極的にこのような技術に挑戦していくことが必要ではないだろうか．

本稿の後半で紹介したSpinは以下のURLからダウンロードでき，フリーで使うことができる．実際に動作するPromela記述をいくらか紹介してきたので，実際に打ち込んでツールを動作させ，モデル検査手法を体感してほしい．

〔http://spinroot.com/〕

参考文献

1）Holzmann, G. J.: The Spin Model Checker Primer and Reference Manual, addison-wesley(2003).
2）A. S. タネンバウム, A. S. ウッドハル：オペレーティングシステム―設計と理論およびMINIXによる実装 第2版，ピアソン・エデュケーション.
3)坂村　健：μITRON4.0標準ガイドブック，パーソナルメディア(2001).
4)青木利晃，片山卓也：RTOSに基づいたソフトウェアのためのモデル検査ライブラリ，組込みソフトウェアシンポジウム2005, pp.56-63(2005).

（平成18年4月3日受付）