# マルチタスクソフトウェアの検証法

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

# 背景

- 多くの形式手法が提案されている。
  - いくらかの手法は実際の開発に適用可能なレベルになっている。
    - モデル検査手法、定理証明手法、etc.
- 形式手法と開発のギャップ。
- 開発で出現する概念や記述の取り扱い。
  - 概念や記述の形式化。
  - 形式手法のカスタマイズ。
- マルチタスクソフトウェアとモデル検査手法。

# 背景

- RTOSを用いたソフトウェア開発。
  - 並行処理の直接的な記述。
  - 動作やタイミングの解析が必要。
    - デッドロック、飢餓状態、競合状態、etc.
- モデル検査手法。
  - 動作やタイミングの検証。
  - 有限状態で特徴づけられる振る舞いを網羅的に探索。
    - 並行性、非決定性を持つ振る舞いの網羅的探索。
- モデル検査手法を用いてRTOS上のソフトウェアを検証する。

# 背景

- 対象：μITRON準拠したRTOS上で動作するソフトウェア。
- モデル検査ツールSpinを用いる。
  - 非同期プロセスを取り扱うことができる。
    - μITRONのタスクの概念に近い。
  - 仕様記述言語Promela
    - 優先度、セマフォ、Sleep/Wakeupなどのサービスコールに対応する概念が無い。
- タスクスケジューリングの取り扱い。
  - スケジューリングを無視する。
    - 広い振舞いを調べる。
    - 実際には起こりえない反例が多く生成される。
  - スケジューリングを模倣する。
    - 実装に近い動作で検証を行える。
    - スケジューリングの模倣で状態数が増える恐れ。

# 目的

- モデル検査ツールSpinでμITRONに基づいた優先度付きマルチタスクスケジューリングを扱えるようにする。
  - μITRONの振る舞いをSpin上で模倣する。
  - RTOS内部の計算方式をPromela/Spinで実装する。

  ＝μITRONのためのSpinのライブラリ。
- 状態爆発を回避する。

# 目的

検証対象

優先度:3(低)

```
while(1){
   wai_sem(0,P1);
    /* critical section */
   printf("P1¥n");
   sig_sem(0)}
```

優先度:2(中)

```
while(1){
 printf("P2¥n");}
```

優先度:1(高)

```
while(1){
   wai_sem(0,P3);
    printf("P3¥n");
progress: sig_sem(0);
    yield(P1)}
```

RTOSライブラリ(μITRON RTOSエミュレータ)

モデル検査ツールSpin

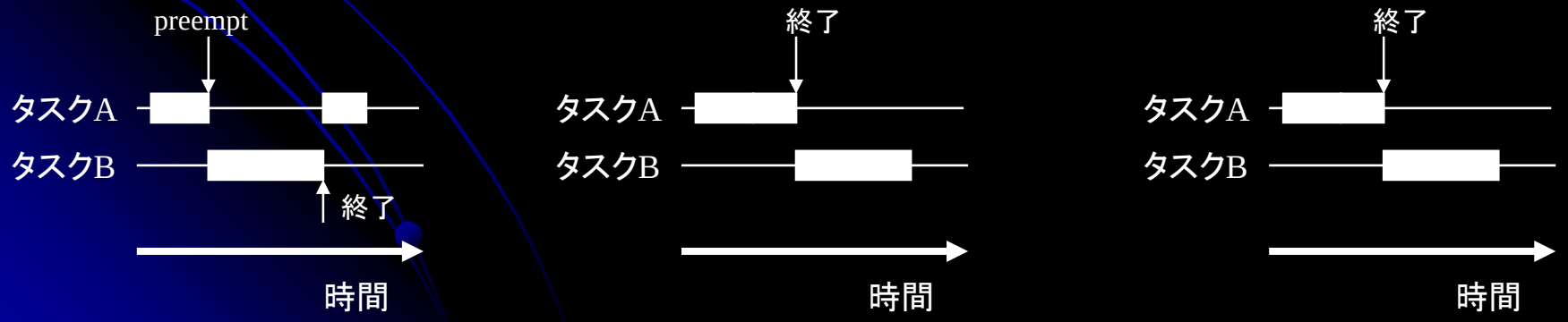
並行ソフトウェア
の問題の検出

反例の出力

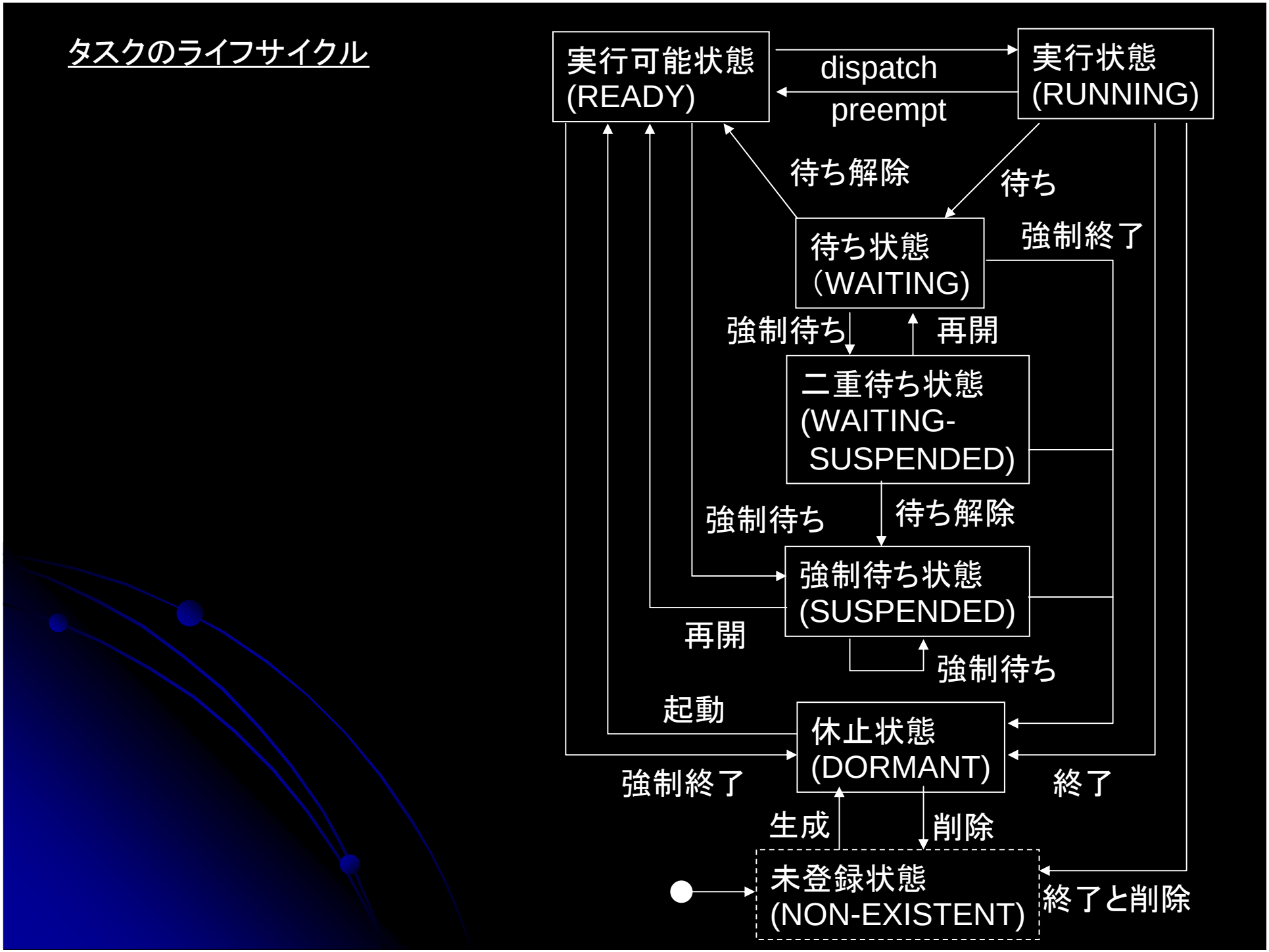
```
4:     high(3):[wai_sem(0 ,2);now.turn=top();]
P3     high(3):[printf('P3¥n')]
28     high(3):[sig_sem(0);now.turn=top();]
30     high(3):[yield(0)]
32 low(2):[wai_sem(0,0);now.turn=top();]
34     high(3):[wai_sem(0,2);now.turn=top();]
<<<<<START OF CYCLE>>>>>
P2
36:  mid(4):[printf('P2¥n')]
```

# 優先度付きマルチタスクOS

- タスクには優先度が割り当てることができる。
  - タスクAが実行状態の時、タスクBが実行可能になったとする。
    - タスクAの優先度＜タスクBの優先度、の場合。
      - タスクAはpreemptされて実行可能状態に、タスクBが実行状態になる。
    - タスクAの優先度＞タスクBの優先度、の場合。
      - タスクAが実行されつづける。
    - タスクAの優先度＝タスクBの優先度の場合。
      - タスクAが実行されつづける。FCFS(First Come First Service)

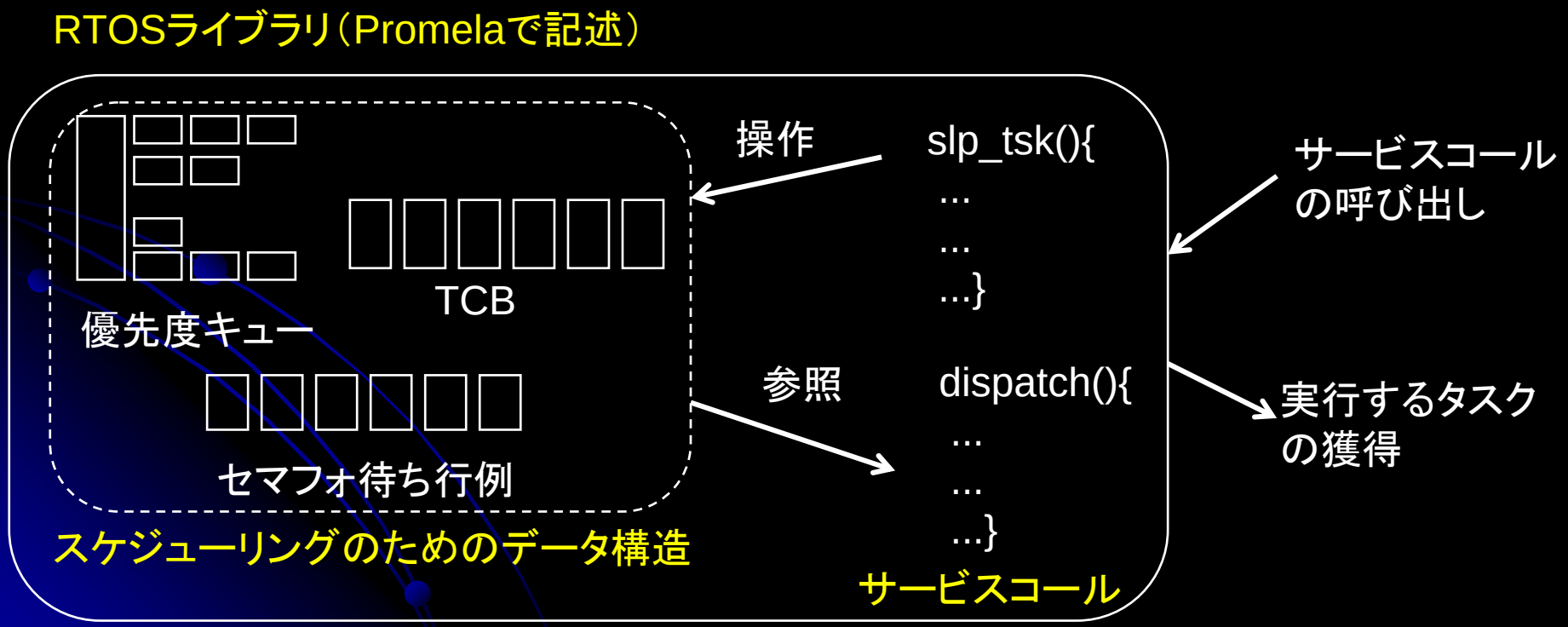


# タスクのライフサイクル



# RTOSライブラリ

- RTOSカーネルと同様の計算方法でタスクのスケジューリングを実現する。



# RTOSライブラリ

- RTOSカーネルと同様の計算方法でタスクのスケジューリングを実現する。

Promelaライブラリのヘッダの一部

```
#define TID int  /* タスク識別子の型 */
#define PRI int /* 優先度の型 */
/* 優先度キュー */
#define prio(x,y) ready[((x) * N_TASK) + (y)]
TID ready[N_PRIO_TASK];
/* タスクの状態 */
#define NOTEXIST        0
#define DORMANT         1
#define READY           2
#define RUN             3
#define WAIT_SLEEP      4
#define WAIT_WOBJ       5
#define WAIT_SLEEP_SUSPENDED    6
#define WAIT_WOBJ_SUSPENDED     7
#define SUSPENDED       8
```

```
typedef TCB{     /* タスクの管理に必要な情報.  */
  PRI tpriority; /* 優先度 */
  byte tstat;    /* 状態 */
  TID tid;      /* タスクID */
  byte actcnt;   /* キューされている起動要求回数 */
  byte wupcnt;   /* キューされている起床要求回数 */
  byte suscnt    /* キューされている強制待ち要求回数 */
};
TCB tsk_state[N_PRIO_TASK]; /* タスクの状態を管理する配列 */
typedef Semaphore{ /* セマフォの型 */
  int num;
  int max;
  TID block[N_PRIO_TASK] /* ブロックキュー */
};
Semaphore sem[N_SEM]; /* セマフォを格納する配列. */
```

# 使い方（交互実行）

## 交互実行

```
#define P1 1
#define P2 2
#define P3 3
proctype low() provided (turn == P1){
  do
  :: printf("P1¥n")-> wup_tsk(P2);
  od}
proctype high1() provided (turn == P2){
  do
  :: printf("P2¥n")-> wup_tsk(P3);
     slp_tsk(P2);
  od}
```

```
proctype high2() provided (turn == P3){
  do
  :: printf("P3¥n")->slp_tsk(P3);
  od}
init{
 ini();
 cre_tsk(1, P1);cre_tsk(2, P2); cre_tsk(2, P3);
 act_tsk(P1); act_tsk(P2); act_tsk(P3);
 run low();
 run high1();
 run high2()}
```

## シミュレーション結果

```
BEGIN0:initialization of the library     P2
    END
    BEGIN0:cre_tsk(2, 1)
    END
....
BEGIN0:act_tsk(1)
    END
....
```

```
BEGIN2:wup_tsk(3)
END
BEGIN2:slp_tsk(2)
END
   P3
   BEGIN3:slp_tsk(3)
   END
```

```
P3
        BEGIN3:slp_tsk(3)
        END
    P1
    BEGIN1:wup_tsk(2)
    END
....
```

# 使い方（デッドロックの検出）

**セマフォ使用によるデッドロック**

```
#define P1 1
#define P2 2
proctype intr (){
  rot_rdq(1)}
proctype prs1() provided (turn == P1){
   do
   :: wai_sem(1, P1) ->
      wai_sem(2, P1);
      printf("P1¥n");
      sig_sem(2);sig_sem(1)
   od}
```

```
proctype prs2() provided (turn == P2){
   do
   :: wai_sem(2, P2) ->
      wai_sem(1, P2);
      printf("P2¥n");
      sig_sem(1);sig_sem(2)
   od}
init{
 ini();
 cre_tsk(1, P1);cre_tsk(1, P2);
 cre_sem(1, 1);cre_sem(2, 1);
 act_tsk(P1);act_tsk(P2);
 run prs1(); run prs2(); run intr()}
```

# 使い方（優先度逆転の検出）

- 優先度逆転問題
  - High, Mid, Low
    - Lowが共有資源を獲得。リリースする前にHighにディスパッチ
    - Highが共有資源獲得しようとする。ブロックする。
    - Midにディスパッチ。
    - Midが動き続けてLowにはディスパッチされない。LowとHighは動くことができない。

```
#define P1  1
#define P2  2
#define P3  3
proctype low() provided (turn == P1) {
  do
  :: wai_sem(0,P1)->printf("P1¥n");
     sig_sem(0)
  od}
proctype mid() provided (turn == P2){
  do
  :: printf("P2¥n");
  od}
```

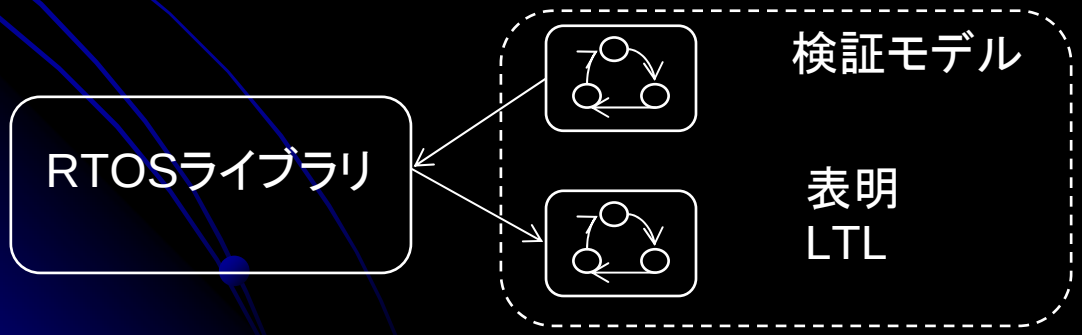
```
proctype high() provided (turn == P3){
  do
  :: wai_sem(0,P3);printf("P3¥n");
progress: sig_sem(0); yield(P1)
  od}
init{
  ini();
  cre_tsk(1,P1);
  cre_tsk(2,P2);
  cre_tsk(3,P3);
  cre_sem(0,1);
  sta_tsk(P1);
  sta_tsk(P2);
  sta_tsk(P3);
  run low(); run high(); run mid()}
```

# 状態爆発の回避

- 検証したいものはタスクの振舞い。
  - RTOS自体ではない。
- サービスコール内の処理を不可分操作にする。
  - 内部処理を直列に実行したものと同等になるように実装されているものとする。
  - オプションでインターリーブ可能とできる。
- 一時変数の取り扱い。
  - サービスコール内で使用する一時変数は、最後に初期状態に戻す。
- RTOSの模倣の部分では状態を消費しない。
  - 状態ベクトルのサイズは大きくなる。
    - 優先度キュー、TCB、etc...

# ライブラリの正しさ

- ライブラリが仕様どおり実現できているか検証したい。
  - RTOS自体が正しく実現できているかどうか検証するのと同じ。
- 仕様をモデル検査で使えるくらい形式化する必要がある。
  - いくらかの観点。
    - タスクライフサイクル、セマフォ、メールボックス、etc.
  - 検証モデルの作成。



# ライブラリの正しさ

- タスクのライフサイクルに関する検証モデルの作成。
  - 個々のサービスコール毎に以下を記述。
    - 呼び出すことができる状態
    - 呼び出した後の状態
  - それらをまとめて状態遷移モデルを作成。
    - キューイング数を表現するための変数を導入。
- エラーの検出。
  - 考慮されていなかった状態によるサービスコールの呼び出し。
  - キュー操作の誤り（enq/deqしていなかった。する必要ないのにしていた。）
  - エラーや操作のための条件が不正確だった。

```
[cre_tsk]
 pre: state == NOTEXIST
 act: Task::cre_tsk(p, task)
      actNnum = 0
      susNum = 0
      wupNum = 0
 post: self.state == DORMANT
```

```
[slp_tsk]
 pre: state==RUN && wupNum == 0
 post: WAIT_SLEEP

 pre: state == RUN && 0 < wupNum
 act: wupNum --
 post: state == READY
```

# 考察

- RTOSライブラリ自体では、状態数が増えない。
  - モデル検査で必要な状態数は、検証したいタスクの振舞い（状態数）に依存。
  - 状態ベクトルのサイズは大きくなる。
    - 消費メモリ＝状態ベクトルサイズ×状態数
- スケジューリングを無視した場合よりは、スケジューリングを考慮したほうが状態数は少ない。
  - RTOSライブラリを導入することにより、状態数を減らすことができる。
  - セマフォによるデッドロックの例（デッドロックを解消）。
    - スケジューリングを無視→状態数：36
    - スケジューリングを考慮→状態数：15

# 考察

- 状態ベクトルのサイズ。
  - 取り扱うタスク数、セマフォ数、メールボックス数などに依存。
  - それぞれの数に関してコンスタントに増加。
    - タスク数：1-12byte, 2-172byte, 3-216byte, ..., 12:624byte, ..., 30: 1404byte, ...
  - メモリ消費量へのインパクトは小さい。

# 考察

- ライブラリの正しさ。
  - ライブラリが仕様どおり実装されているか検査した。
  - 仕様をPromelaで書く必要がある。
    - μITRONの仕様は自然言語で書かれている。
    - 完全な正しさを保証するためには、仕様も形式化して厳密に決める必要がある。
      - 複数の要件を組み合わせて検証できる枠組みが必要。
      - タスクライフサイクル×セマフォ×メールボックス…
- μITRONの振舞い仕様。
  - ライブラリの正しさを保証することにより、RTOS実装のリファレンスとなる。
    - 設計フェーズで、ライブラリとの等価性を保証。
    - 設計した最適化法、メカニズムの正しさを確認。
  - 正しさの保証に用いた検証モデルは、RTOS実装の検証にも用いることができる。

# 考察

- 実開発への適用。
  - 本学の岸研究室が実開発への適用実験を行っている。
    - まずまずうまくいったと聞いています。。。
  - 数社に配布。
    - フィードバックはまだ来ていません。

# まとめ

- μITRON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を提案。
- 優先度つきスケジューリングにまつわる典型的な問題が検出可能。
- 状態爆発問題を回避。
  - モデル検査に必要な状態数は、検証対象の状態数に依存。
  - タスク数に関してコンスタントに状態ベクトルのサイズが増加。

# 今後の課題

- 時間の取り扱いについて。
  - 周期とタイミング検証の関係。
    - 周期実行に関する検証と解析手法→組み込みシステムシンポジウム2007で発表。
  - 他にもいろいろな典型的な問題がある。
    - 問題を整理して、解決策の提案を積み上げる必要がある。
    - マルチタスクソフトウェア検証セットの作成。
      - 一般的な枠組みがある？
- RTOSの仕様、実装法の検証。
  - 振舞い検証のための検証モデルの作成。
  - μITRON仕様の誤り、実装法の誤りを検証する。

# ライブラリの公開

- 名称: μIPRON
  - μITRON RTOS用ライブラリ for PROmela/spiN
- 以下のURLから辿れる場所に本体、マニュアル、サンプルが置いてあります。
  - http://aoki-www.jaist.ac.jp/~toshiaki/
  - 左のメニューのRTOSライブラリをクリック。
- パスワード制御しています。使ってみたい方は青木(toshiaki@jaist.ac.jp) まで。