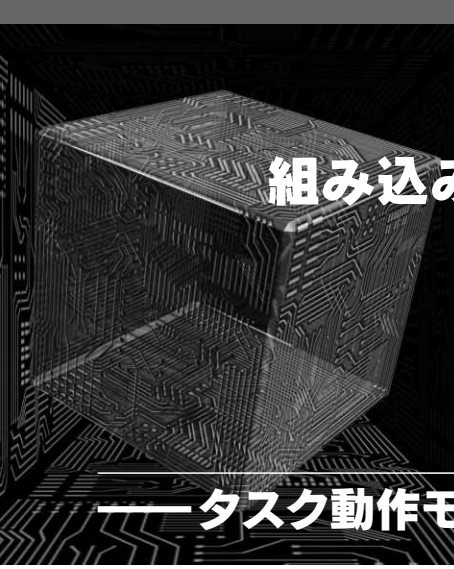


組み込みプログラミング・ノウハウ入門（第31回）

タスクの動きをモデル化する

――タスク動作モデルの作成（2）

藤倉 俊幸

前回は，RTOSのスケジューラをモデル化することで，タスク動作のモデル化を行った．RTOSを入れない場合に比べて，場合の数が減少して扱いやすくなること，タスク動作の全体を見られることを説明した．また，タスク・スイッチングの原因になるシステム・コールのモデル化方法についても説明した．この結果，話が詳細設計から実装のほうに向かってしまった．

今回は，システム・コールをマップする前の，もっと抽象度の高いタスク設計について説明する．使用するシステム・コールを具体的に検討する前に，「ここでスイッチングする」とか，「ここで待ちに入る」といったレベルで話を進める．いわば動きのアーキテクチャ設計である．抽象度が高いので，特定のRTOSを意識する必要はない．そのためRTOSを決める前でもタスク設計と検証ができる．また，スケジューラを複数導入すると並列分散システムのモデリングもできる．つまり，ハードウェアを決める前からタスク設計が行えるのだ．またさらに，CANのような「優先度付きネットワーク」のモデルも加えれば，通信路も含めたシステム全体のモデリングもできる．つまり，ハードウェアを選定する前からタスク設計（もうタスク設計とは言えない）を行えて，最適なハードウェア選定にも貢献できる．

従来のパッチワーク的ソフトウェア開発…建て増しで迷路と化した温泉旅館のようなソフトウェアは限界にきている．すべての問題をかたくなにソース・コードだけで解決しようとする人々の努力と忍耐と血と涙の結晶，たとえばサイクロマティック数[注1]が300近い関数などには，どのような物語があるのかを想像するだけならおもしろい．しかし，放って置くわけにもいかないし，いつまでそのようなことをしていられるのだろうか．

開発期間の短縮と開発コストの削減を迫られる中で，このような状態にならずに，組み込みシステムの品質や柔軟性を向上させる方法の一つは，これから述べるような，抽象度の高い設計とその検証手法の導入である．

注1：McCabe's cyclomatic complexity．プログラムの複雑さを静的に計測する数値．最適な値は10以下と言われている．
e＝プログラムに含まれる基本ブロックの数
n＝プログラムに含まれる分岐点と合流点の数
p＝自分自身を含めたサブルーチンの数
複雑度 $C = e - n + 2p$

1 タスク設計の簡単な例

● まず処理をタスクに分解する

例として，デバイスの入力を処理するタスクを考える．要求分析の結果，

●行うべき処理が三つ：act1，act2，act3

が抽出されたとする．そして，act2とact3は，act1終了後に実行しなければならないという順序制約も抽出されたとする．たとえば，act1がデバイスからの入力処理で，act2とact3は入力されたデータを利用する処理だとする．その場合，act2とact3はact1の終了を待つ必要がある．act2とact3の間には順序制約はないのでどちらを先に実行しても良いとする．このとき，どのようなタスク設計をすべきだろうか．必要な処理はすでにわかっている状況で，その処理をタスクにマップするオブジェクト-タスク・マッピングの問題と考えることもできる（**図1**）．

この要求仕様は，act1→act2→act3またはact1→act3→act2の順で処理を行う一つのタスクによって実現できる．ところが，act2とact3には別々のデバイスへの出力処理が含まれているので，処理の途中でI/O待ちが発生するとする．この場合，一つのタスクでは処理効率が悪い．act2から処理を始めて，act2がI/O待ちになったら，その間にact3の処理を始めてact3がI/O待ちになるころにはact2のI/Oが終了しているとCPUが遊ばなくてぐあいが良い．そこで，タスクを二つ用意する．この結果，たとえば，

```
TaskA = (act1->act2->TaskA).
TaskB = (act3->TaskB).
```

のようなタスク設計を行ったとする．ただし，これだけでは

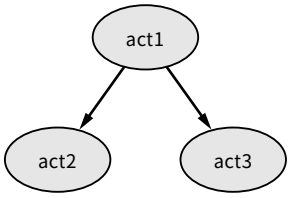


図1
ハッセ図で表した順序制約

KEYWORD ――タスク設計，LTSA，RTOS，順序制約

```
||System2=(RTOS||Tasks)@{TaskAAct,TaskBAct}.
```

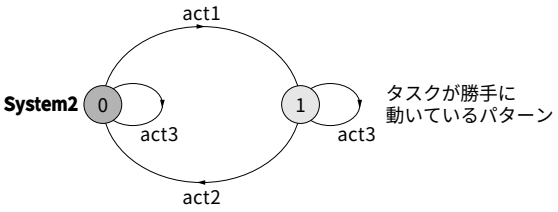
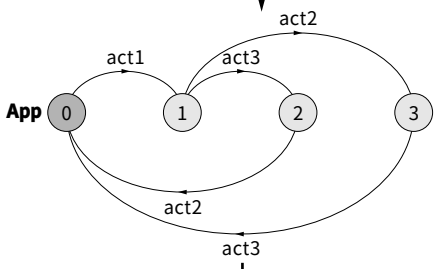


図2　アプリケーション・レベルの動作

```
// 要求としての順序制約
Ap1=(act1->act2->Ap1).
Ap2=(act1->act3->Ap2).
||App=(Ap1||Ap2).
```




```
// 要求仕様を守るための安全性条件
property AppSafe=Q0,
   Q0=(act1->Q1),
   Q1=(act3->Q2
      |act2->Q3),
   Q2=(act2->Q0),
   Q3=(act3->Q0).
```

図3　順序制約を満たす動作パターンと安全性条件

各タスクに処理を割り当てただけなのでタスク設計とは言えない．TaskA と TaskB の間に同期構造を入れて要求仕様を満たす必要がある．たとえば，同期構造を入れないと**図2**のような動作になってしまう．

● 次に順序制約を盛り込む

まず，満たさなければならない要求仕様，すなわち**図1**に示した順序制約を表現する．要求仕様から順序制約を抽出する方法は，この連載の中の動作モデル作成のときに説明した．まず，順序制約を順序対として表現する（**図3**の Ap1 と Ap2）．そして，それを並列合成する（**図3**の||App）．その並列合成した結果が，順序制約を満たす動作パターンになる．その動作パターンを安全性条件とする．LTSA の具体的表現は property AppSafe とする．

● タスクとしての動作パターンを生成する

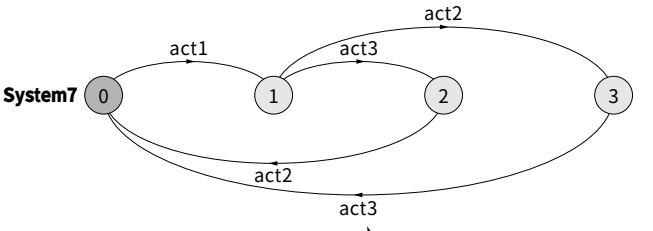
次に，前回作成した RTOS モデルと上記のタスク・モデルを結合してタスクとしての動作パターンを生成する．このときにはプライオリティを決めなくてはならないので，とりあえず TaskA を高プライオリティとする．タスク設計手順とその結果を**図4**に示す．結果は，八つの状態をもった動作パターンになった．

このステート・マシンを見ても制約を満たしているかわからないので，先に作っておいた AppSafe とリンクする．そうすると反例を網羅的に作ってくれる．また，反例として興味があるのはアプリケーション・レベルの動作なので @{TaskAAct, TaskBAct}を使用してミニマイズすると**図5**のようなアプリ

```
set TaskAAct2={act1,act2,act2b,w.act3}
set TaskBAct2={w.act1,act3,act3b}
TaskA2=(act1->w.act1->act2->aio->act2b->w.act3->TaskA2).
TaskB2=(w.act1->act3->bio->act3b->w.act3->TaskB2).
||Tasks2=(TaskA2||TaskB2).

RTOS=Rr,//1:A 2:B
Ww=({w.act3,r1}->Rw|{w.act1,r2}->WR),
Rw=({w1,aio}->Ww|{w.act1,r2}->Rr|TaskAAct2->Rw),
WR=({w.act3,r1}->Rr|{w2,bio}->Ww|TaskBAct2->WR),
Rr=({w1,aio}->WR|w2->Rw|{TaskAAct2,w.act1}->Rr).

||System6=(RTOS||Tasks2||AppSafe)@{TaskAAct2,TaskBAct2,aio,bio}.
||System7=(RTOS||Tasks2||AppSafe)@{act1,act2,act3}.
```



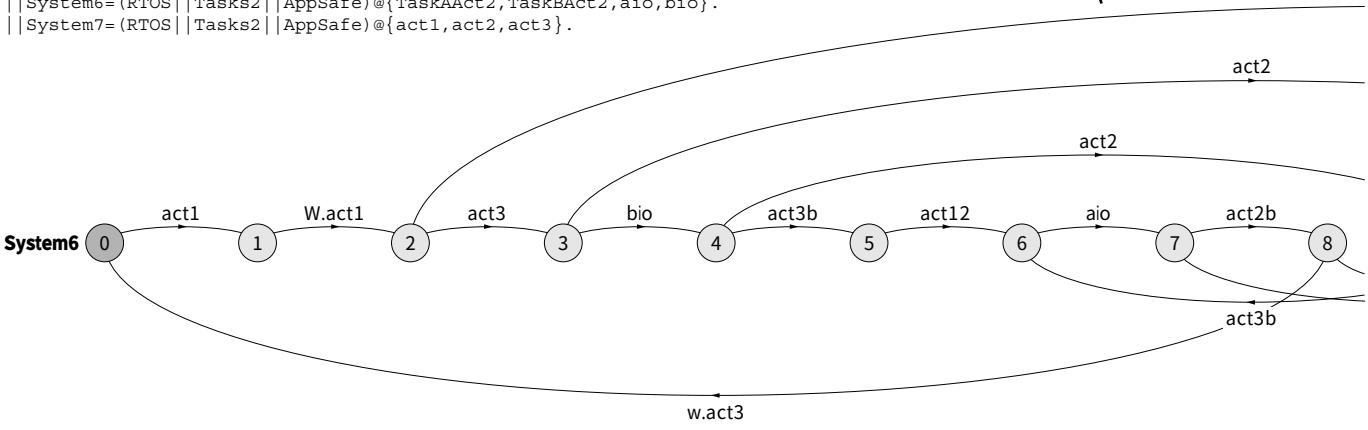


図6　アプローチ1の設計と検証