第16章 LTSAによる動作モデルの検証

他人のモデルの
バグ探しから始めよう

本章では，これまで何度か出てきたLTSAについて，基本的な使い方を解説する．

1 モデル検査機LTSA

● LTSAによるモデル検査入門

ステート・マシンの生成/検査ツールであるLTSA (Labelled Transition System Analyser)は，おそらく一番簡単に扱えるモデル検査機である．プログラムはWebサイトからダウンロードできるし，英語ながら書籍もある[(1)]．また，本を買わなくとも書籍の内容をまとめたスライドをLTSAのWebサイト[(2)]からダウンロードできる．さらに，このスライドについては日本語化されたものが存在している[(3)]．したがって入門にちょうど良い．しかし，残念ながら実際に組み込み開発に利用したという話はあまり聞かない．

● 設計情報のみでテスト可能

モデル検査の良いところは，

1）ソース・コードがなくてもテストできること
2）網羅的にテストするのでテスト・ケースを作らなくても済む

である．つまり，設計情報があれば，それだけで網羅的にテストできてしまう．だから，書籍や論文に載っているサンプル程度の情報で，それにどのような問題が潜んでいるか検査できるのである．そのようなことで，モデル検査用のツールをダウンロードすると，まず他人が作ったプログラムを検査してみたくなる．

● LTSAは十分実用になる

モデル検査は，形式的手法の一つであり，アカデミックな人たちの間では常にホットな話題である．しかし，LTSA以外のツール，たとえばSPINやNuSMV，UPPAALなどの報告はあってもLTSAに関するものはほとんどない．アカデミックな人たちには簡単すぎておもしろくないのかもしれない．参考文献(1)は約350ページあるが，数学的なエッセンスはAppendix Cのわずか7ページのみである．LTSAがアカデミック受けしない理由として考えられるのは，各種の制限があること，たとえば最近までLTSAは時相論理式を使って検査条件を記述できなかったことや，ローカル変数しか使えないことなどだろうか．つまり，モデル検査能力やモデル記述力がほかのツールに比べると少し劣る．

しかし，時相論理式は読み書きできるようになるまでにけっこうな壁があるので最初はなくても良いし，LTSAで物足りなくなったときにそのありがたみがわかるので，LTSAをマスタしてからでも遅くはない．

それに，モデル検査で複雑なことをしようとすれば，すぐに状態爆発が起こってしまう．したがって，モデルはなるべく簡単なほうが良い．つまりどうせ複雑なモデルは書けないので，モデル記述力はLTSAで十分であるとしておく．

● LTSAの利点

LTSAの良いところは，簡単なだけでなくステート・マシンを合成してくれることである．合成したステート・マシンを絵にして見せてくれるので，検証だけでなく設計工程の中でも利用できるのである．

そこで本章では，LTSAを使ってモデル検査を実際に行う．そしてその後，さらにSPINを使ってモデル検査の概要を説明する．本章で紹介するリストはCQ出版社のWebサイトからダウンロードできるので，各自試してみてほしい[注1]．

注1：http://www.cqpub.co.jp/interface/techi/techi.html

見本

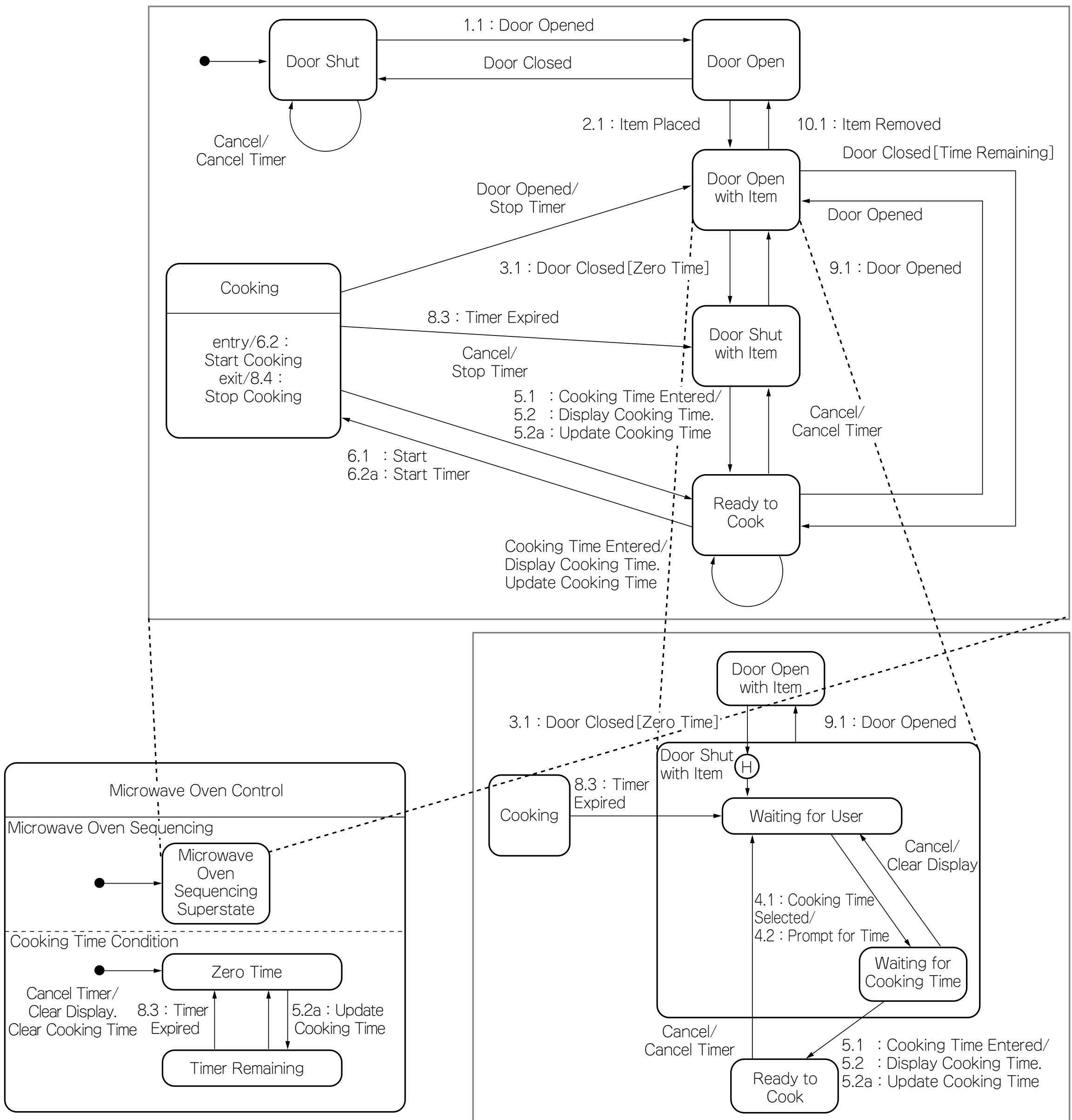

図1　Gomaaの電子レンジ制御のステート・マシン

2 LTSAで電子レンジのモデルを検査する

● LTSAのインストールと実行

それでは実際にLTSAを使って，Gomaaのプロダクト・ラインの本[(4)]に出てくる電子レンジの制御に対してモデル検査をしてみよう．**図1**がGomaaの本に掲載されているステート・マシンである．並列に動く二つのステート・マシン，Microwave Oven Sequencing と Cooking Time Conditionからできている．この図からLTSA用のモデルを作成する．

まだLTSAを持っていない人は，

`http://www.dse.doc.ic.ac.uk/ltsa/`

からLTSAをダウンロードする必要がある．アーカイブを解凍すると`readme.txt`が出てくるのでそれに従う．標準のままだとJavaのJRE1.3.0を利用するようになっているので，もうJRE1.3.0を持っていない人は1.4などを利用するように`ltsa.bat`を変更する必要がある．それには以下のようにする．

`"C:¥Program Files¥Java¥j2re1.4.2_03¥`

見本

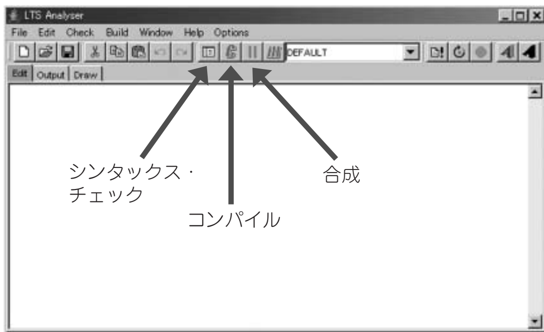

図2　LTSA起動画面

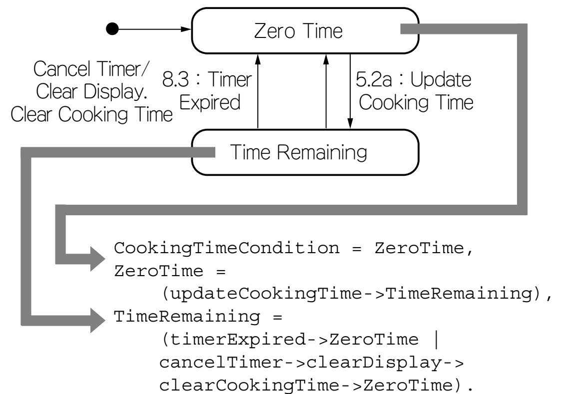

図3　Cooking Time ConditionのLTSAへの変換

```
bin¥javaw.exe" -Xmx128m -jar
"C:¥Program Files¥Concurrency¥
                    ltsa2.2¥lib¥ltsa.jar"
```

ltsa2.2の部分はバージョンによって変わってくるので，適宜読み替えてほしい．これでltsa.batをダブルクリックすればLTSAが**図2**のように立ち上がる[注2]．Editタグが選択された状態になっているので，その下の空白領域にステート・マシンを定義する．UMLツールのようにグラフィカルな入力はできないので，テキスト形式でプロセス代数の式を入力する．プロセス代数というと腰が引けてしまうが，LTSAの場合は意外と簡単である．

● ステート・マシンの入力と実行（その1）

それでは二つのステート・マシンのうち簡単なほうのCooking Time Conditionで説明する．まずステート・マシン名を入力する．この場合は，Cooking TimeConditionとする．このステート・マシンには二つの状態，Zero TimeとTime Remainingがあり初期状態がZero Timeなので，次のような骨格を作る．

```
CookingTimeCondition = ZeroTime,
ZeroTime =
TimeRemaining =
```

次に各状態について，この場合はZeroTimeとTimeRemainingから出ていく遷移関係を書き込んでいく．Zero TimeでUpdate Cooking Timeイベントを受け取るとTime Remainingに遷移するので，

```
ZeroTime = (updateCookingTime
                  -> Time-Remaining),
```

とする．なお表記には規約があり，

- 状態名とステート・マシン名は大文字で始める
- イベントやアクションは小文字で始める
- 最後の“,”は継続の意味

である．ZeroTime状態から出ていく遷移はこれだけなので，次にTimeRemaining状態に進む．この状態には出ていく遷移が二つある．遷移が複数ある場合は“|”で区切って入力する．つまり，次のような構造にして，

```
TimeRemaining =（ アクション列
          ->遷移先の状態名 | … | … ）.
```

のそれぞれにアクション列と遷移先の状態を書き込めばよい．遷移の一つはTimer Expiredイベントを受け取ってZero Timeに遷移するものなので，timer Expired->ZeroTimeとなる．もう一つは，Cancel Timerイベントを受け取って，Clear Display処理とClear Cooking Time処理を行ってZero Time状態に遷移するので，

```
cancelTimer->clearDisplay
          ->clearCookingTime->ZeroTime
```

となる．LTSAでは入力と処理や出力は区別しない．この二つの遷移を入力すると，

```
TimeRemaining =
   (timerExpired->ZeroTime |
    cancelTimer->clearDisplay->
    clearCookingTime->ZeroTime).
```

となる．最後の“.”はステート・マシン定義の終了を意味する．最終的には**図3**のようになる．この入力が

注2：参考までに，編集部の環境（Sun Java 1.4.2，展開後のltsa2.2ディレクトリにいる状態）ではltsa.batを以下のように変更すると動作した．

```
"C:¥Program Files¥Java¥j2re1.4.2_04¥bin¥javaw.exe" -Xmx128m -jar lib¥ltsa.jar
```

見本

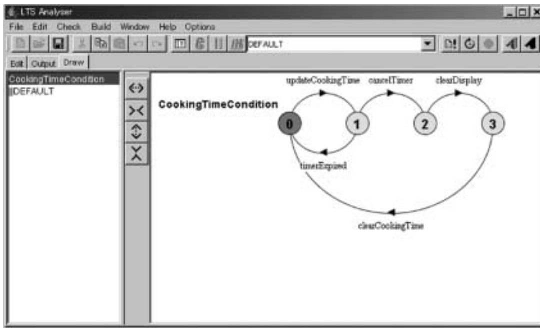

図4　生成されたステート・マシン

終了したら，シンタックス・チェック（メニューの「Check」→「Safety」），コンパイル（メニューの「Build」→「Compile」）を行ってからDrawタブを選択する．そしてCookingTimeConditionを選択すると，生成されたステート・マシンを見ることができる（**図4**）．

つまり，ステート・マシンが先に与えられているときにLTSAモデルを作る場合は，最初に状態をリストアップしてからそこから出ていく遷移を一つずつ書き込めば良い．

● ステート・マシンの入力と実行（その2）

もう一方のステート・マシンMicrowave Oven Sequencingは少し複雑で，Door Open with Item状態が内部状態をもっていること，ヒストリ・ポイントをもっていること，それからCooking状態にエントリ・アクションとエグジット・アクションがあることに注意して変換する必要がある．少しめんどうだが平坦なステート・マシンに変換しなければならない．最終的なMicrowave Oven Sequencingのステート・マシン定義は**図5**のようになる．

並列に動く二つのステート・マシンが定義できた．今度は，この二つを並列化オペレータ“||”で結合してMicrowave Oven Controlステート・マシンを次の式で生成する．

```
||MicrowaveOvenControl=
    (OvenSeq||CookingTimeCondition).
```

実際に合成（メニューの「Build」→「Compose」）を行った場合は，Outputタグが自動的に選択されて，

```
Composition:
MicrowaveOvenControl
   = OvenSeq || CookingTimeCondition
State Space:
 23 * 4 = 2 ** 7
Composing...
  potential DEADLOCK
-- States: 89 Transitions: 149 （略）
Composed in 79ms
```

と表示される．potential DEADLOCKと表示されるので，合成中にデッドロックが検出されていることがわかる．実際にどのようなデッドロックが発生しているのかを調べるには，メニューの「Check」→「Safety」を実行する．すると，

```
OvenSeq = DoorShut,
DoorShut = (doorOpen->DoorOpen|cancel->cancelTimer->DoorShut),
DoorOpen = (doorClose->DoorShut|itemPlaced->DoorOpenWithItem),
DoorOpenWithItem = ( itemRemove->DoorOpen |
    doorCloseZeroTime->DoorShutWithItem_WaitingForUser |
    doorCloseTimeRemain->ReadytoCook),
DoorOpenWithItem2 = ( itemRemove->DoorOpen|doorCloseZeroTime
    ->DoorShutWithItem_WaitingForCookingTime |
    doorCloseTimeRemain->ReadytoCook),
DoorShutWithItem_WaitingForUser = ( cookingTimeSelected->promptForTime
    ->DoorShutWithItem_WaitingForCookingTime |
    doorOpen->DoorOpenWithItem),
DoorShutWithItem_WaitingForCookingTime =
     (cookingTimeEntered->displayCookingTime->updateCookingTime->ReadytoCook |
    cancel->clearDisplay->DoorShutWithItem_WaitingForUser |
    doorOpen->DoorOpenWithItem2),
ReadytoCook = ( start->startTimer->Cooking |
    cookingTimeEntered->displayCookingTime->updateCookingTime->ReadytoCook |
    cancel->cancelTimer->DoorShutWithItem_WaitingForUser |
    doorOpen->DoorOpenWithItem),
Cooking = ( startCooking->DoCooking ),
DoCooking = ( doorOpen->stopCooking->stopTimer->DoorOpenWithItem |
    timerExpired->stopCooking->DoorShutWithItem_WaitingForUser |
    cancel->stopCooking->stopTimer->ReadytoCook).
```

図5　Microwave Oven Sequencingのステート・マシン定義

見本

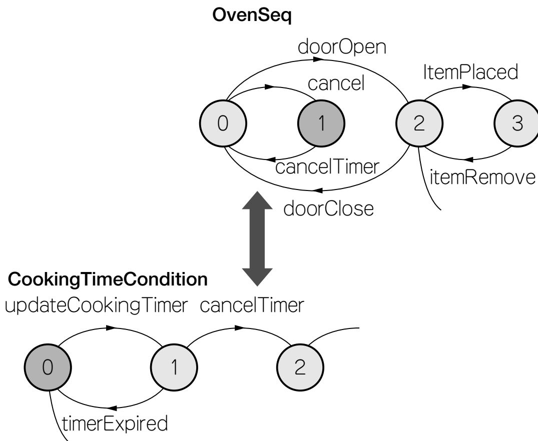

図6　cancelの次の処理が矛盾

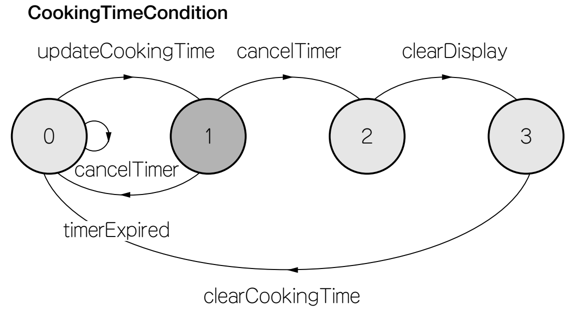

図7　cancelTimer読み飛ばし

```
 Trace to DEADLOCK:
     cancel
```

と表示される．つまり，「いきなりcancelを行うとデッドロックが発生する」ことがわかる（図6）．

このデッドロックは排他制御における互いを待ち合うデッドロックではない．むしろ，その先の処理が未定義あるいは仕様の漏れ，矛盾に相当する．この場合にcancelを実行したとしよう．OvenSeqステート・マシンではcancelTimerを実行しようとするが，CookingTimeConditionステート・マシンでcancelTimerするためにはその前にupdateCookingTimeを実行しなければならない．するとOvenSeqステート・マシンと同期してcancelTimerの実行ができない．そのため矛盾が生じている．タイマを使用しないとき，つまりCookingTimeConditionステート・マシンがZeroTime状態のときは，cancelボタンを使えなくするか，cancelTimerを読み飛ばすかしなければならない（図7）．

そこで，変更が少なくて済むcancelTimerの読み飛ばしでこのデッドロックを切り抜けることにする．

しかし，再度「Check」→「Safety」を実行すると，次のような別のデッドロックが報告される．

```
Trace to DEADLOCK:
doorOpen itenPlaced doorCloseZeroTime
cookingTimeSelected promptForTime
                                 cancel
```

この場合は，図8の⑥のcancelでトリガされる遷移の中でClear Displayを実行しようとしてデッドロックしている．Clear DisplayはCookingTimeConditionの処理だが，やはりタイマのスタート前のZeroTime状態のためClearDisplayが実行できない．したがってこれも読み飛ばす必要がある．そして，再度「Check」→「Safety」を実行すると，また次のような別のデッドロックが報告される．

```
Trace to DEADLOCK:
doorOpen itenPlaced
    doorCloseTimeRemain
cookingTimeEntered displayCookingTime
updateCookingTime cookingTimeEntered
    displayCookingTime
```

この場合は，タイマ作動中にタイマ値を変更するとデッドロックする．これを修正して，再度「Check」→「Safety」を行うと，次のデッドロックが報告される．このデッドロックは今までのものとは違い，並行性に関するものである．

```
Trace to DEADLOCK:
doorOpen itenPlaced
    doorCloseTimeRemain
start startTimer startCooking doorOpen
stopCooking timerEvent
decrementCookingTime finished
```

電子レンジで加熱中にドアを開けると加熱を停止する．一方，設定したタイマがタイム・アウトしても加熱を停止する．この二つがきわどいタイミングで起きたときにデッドロックするのである．タイミングに関するバグは一般のテストでは発見することが難しく，たいていは市場に出てから，クレームとして報告される．

見本

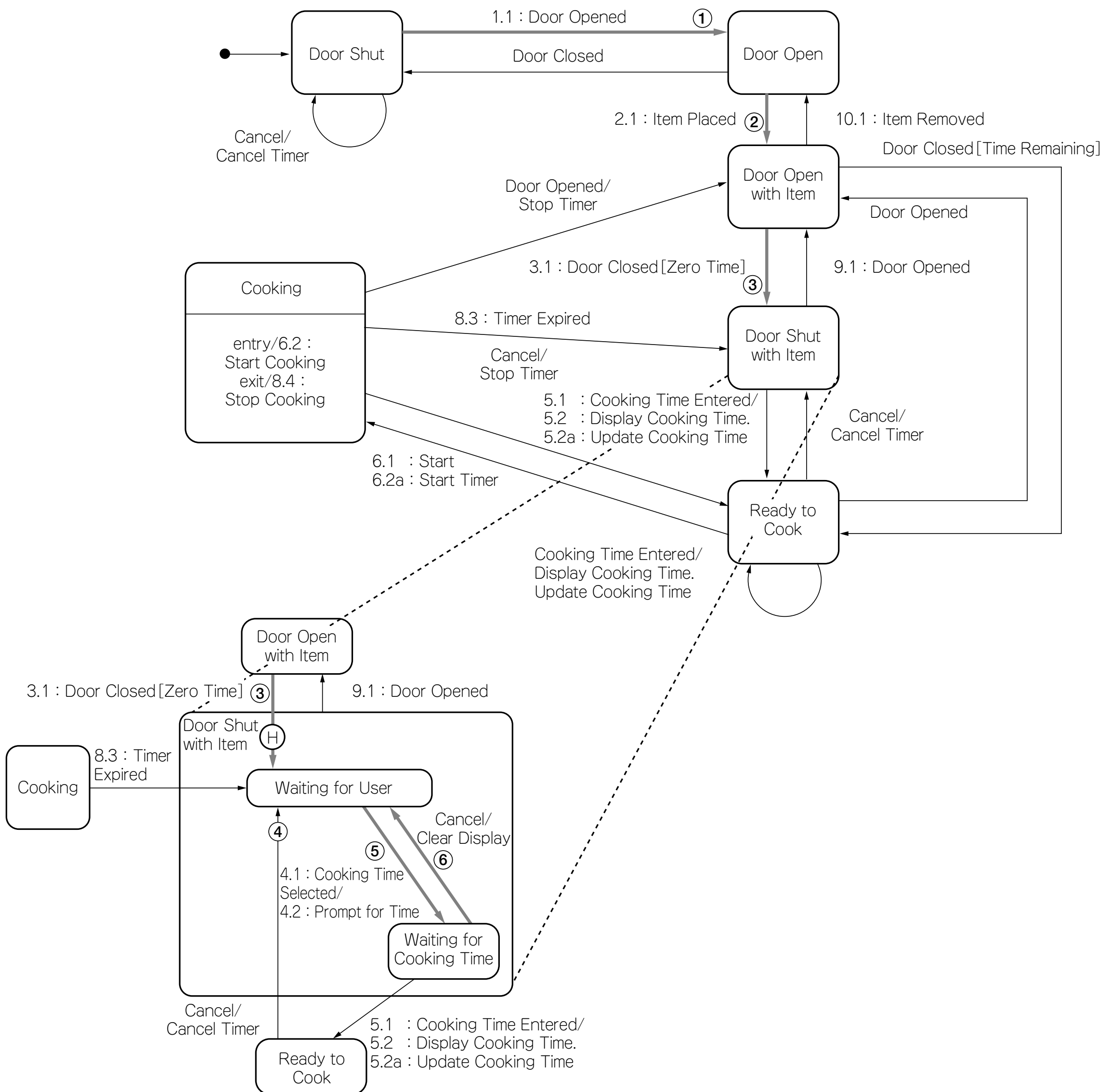

図8　第二のデッドロック

このデッドロックは**図9**に示すように，処理が⑦のStop Cookingまで終わってStop Timerを行う前に，タイマが

```
timerEvent->decrementCookingTime
                          ->finished
```

で終了してしまう．そうすると，もはやStop Timerが作動しなくなるのである．この場合は，今までのようにStop Timerを読み飛ばしただけではだめで，ステート・マシンが`DoorOpenWithItem`状態と`DoorShutWithItem`状態の間で矛盾を起こしてしまう．これを解決するには排他制御を行わなければならない．たとえば，Stop Cookingの前へのタイマ割り込み禁止にして，Stop Timerの後で割り込み解除などを行えばよい．

● モデルだけで検査できてしまうことにもっと驚いてほしい

ここで挙げた例はいずれもソース・コードに落とす前に，モデル検査を用いるだけで済む．設計情報を使うだけで，設計の問題点や実装時に注意しなければならない点の存在が示され，どのような解決策があるのかと試行錯誤できるということだ．ソース・コードに落としてから，そのソース・コードをレビューしても

見本

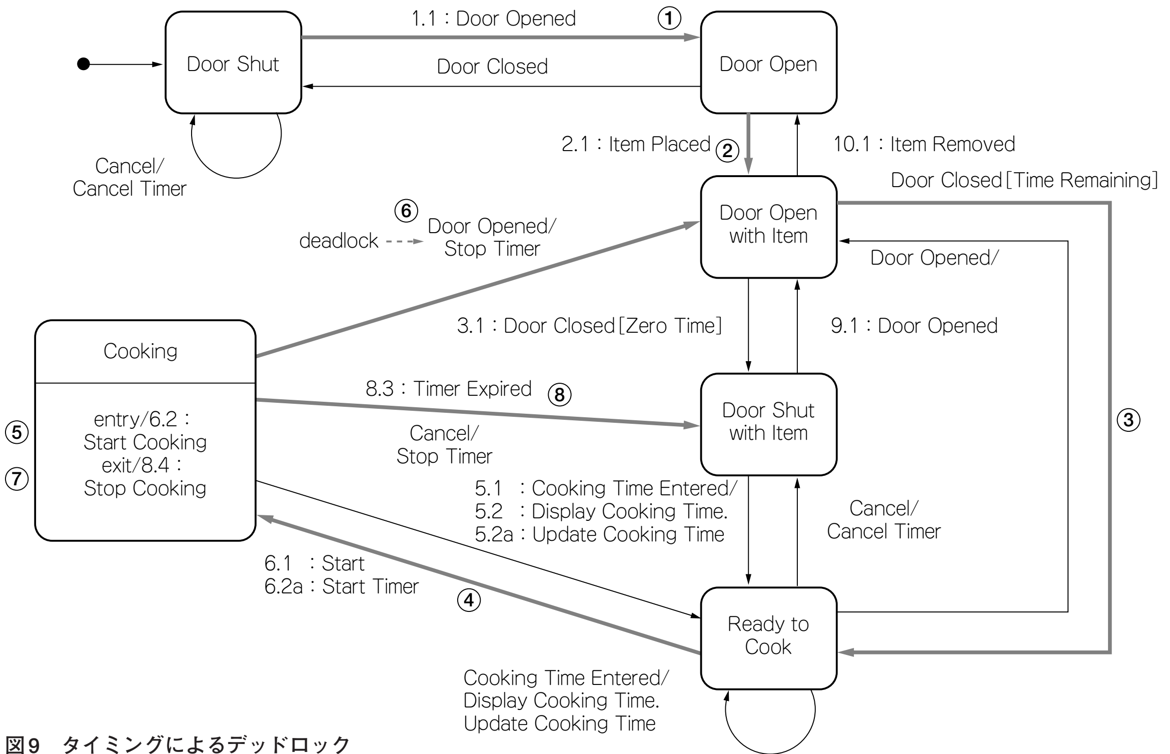

図9　タイミングによるデッドロック

発見できるかどうかわからないような問題点を，モデル検査では網羅的に検査することであぶり出してくれる．このような便利なものが存在するのだから，使わないと損である．

組み込み技術者は，実際にモノが動くことに非常な価値観を感じていることが多い．そのため，早くコーディングしてターゲット・ボードの上で動くところを見てみたいと思ってしまう．しかし，実際に動かすことができるテスト・ケースは全体から見ればほんのわずかでしかない．それにもかかわらず，動いたことで安心してしまうことが多い．

● LTSAを用いた演習で設計スキルを向上する

筆者は最近のコンサルティング作業ではLTSAを使用することにしている．LTSAはアニメーション・モードがあって実際にボタンなどの操作を行っているような動きや，外部イベントを発生させるような操作を行うことができる．したがって，それらの操作によって作成したステート・マシンがどのように動くのかを観察することができる．この点が組み込み技術者に評判が良く，LTSAを導入しやすい．

アニメーションでは満足のいく動作を行うようになっても，モデル検査を行うとほとんどの場合にはデッドロックが発生する．一つのデッドロックをクリアすると，また別のデッドロックが発生する．この繰り返しの過程が良い設計を行うための訓練に最適である．すぐコーディングしたがるエンジニアでも，デッドロックが発生することがわかっている設計に基づいてコーディングする気にはならないようである．簡単な演習を2～3回行うとコツを掴む人が出てくる．そのような人はどんどん設計スキルを上げていく．

その反面，どうしてもデッドロックをとれない人も出てくる．モデルがどんどん複雑になっていってしまうのだ．ステート・マシンが肥大化していき，そのうち状態数が2^{30}くらいになり，モデルをコンパイルすることもできなくなってしまう．普通のパソコン環境なら2^{20}（約100万）あたりが限界である．結局，デッドロックを取り除くことができない人は，設計作業には向いていないと判断せざるを得ないのである．そして，設計スキルの高い人が作成した検査済みのモデルに従って詳細設計以降の作業を担当してもらうことになる．LTSAを導入することはある意味で選別であり，過酷である．

● 設計に向いている人と向かない人の違い

LTSAを使用した設計スキルと，今までの組み込み

見本

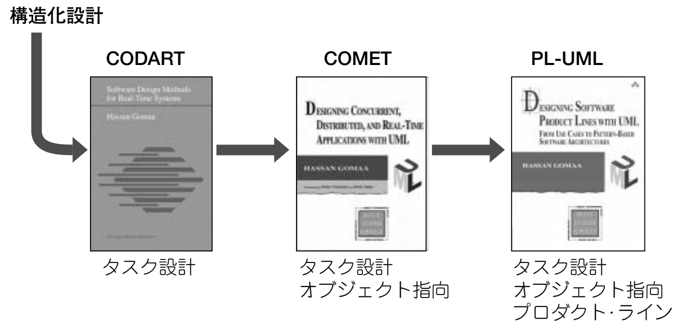

図10
開発手法の変遷の例　●アイデアしだいでどの開発手法の中にでもLTSAを組み込むことができる

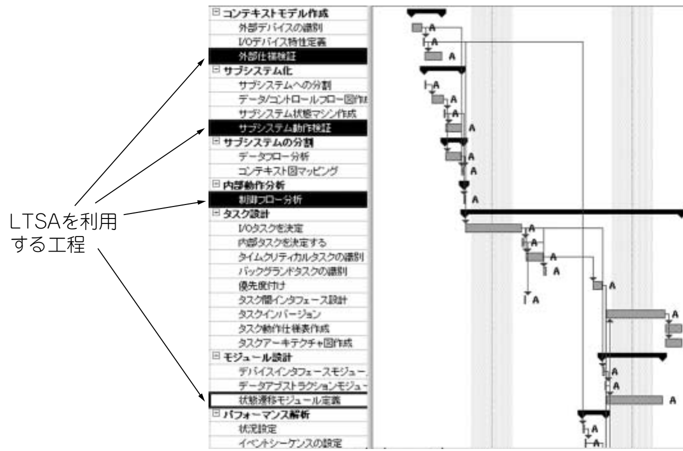

図11
CODARTの場合

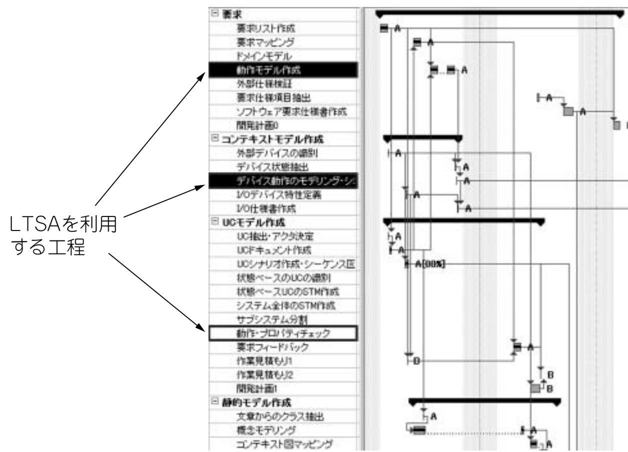

図12
COMETの場合

開発経験年数やハードウェアの知識があるなどということとは，あまり相関がない．設計をソース・コードに落として，実際に走らせてソフトウェアの良し悪しを判断していただけではほとんど見えない設計の良し悪しを，モデル検査は示してくれる．

　ここで言っている意味での設計に向かない人には，共通のパターンがある．すぐに詳細に入ってしまうのである．一度も全体を見ようとしない．向かない人は，全体を分割して，分割したものの間のインターフェースを考えて矛盾がないことを確認してから，分割したものの中身を考えるということをしない．ある部分を分割したらその中を先に作ってしまおうとするのであ

見本

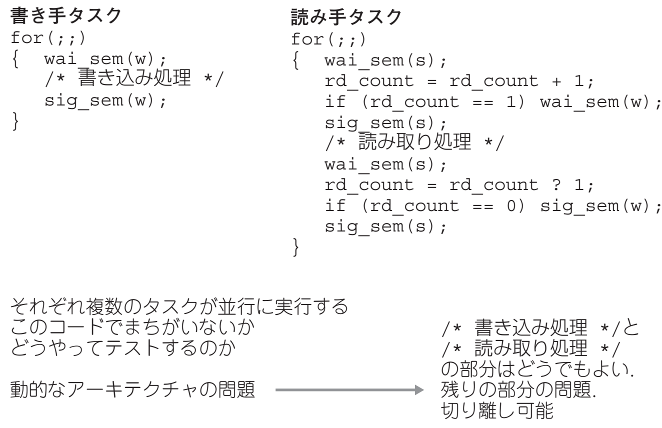

図13
読み手優先排他制御

る．確かに設計に向いている人でも，システムのある特定の部分を切り出して，その中を先に作ることはある．

しかし，向いている人は，次に作ったものの中と外を分ける作業を行う．向いていない人はそれをせずに別の部分を作り始める．つまり，分割しても作業手順を分割しただけで，ソフトウェアの構造を作ってはいないのである．向いている人と向いていない人もUMLで描くと同じような箱と線の図を作るが，向いていない人の箱と線は表面だけである．UMLで表現できないソース・コード・レベルやタイミング・レベルでつながっているのだ．

3 いつLTSAを使うか

他人のモデルでLTSAの有効性を認識したら，実際の仕事に使ってみたくなる．どのような使い方をするかは，現在の開発手法がどのようなものかに依存するが，ふるまいに関することであれば基本的に利用することができる．使用する開発手法は，開発するものの特性などによって変わるが，筆者の経験では**図10**に示したCODARTかCOMETのどちらかになることが多い．これは，オブジェクト指向を使っているか（COMET），使っていないか（CODART）に対応する．

どちらの場合であっても，上流工程で利用することにかわりはない（**図11**，**図12**）．先に挙げた「読み手優先プロトコル」のような制御変数が出てくる前の段階で利用する．また，LTSAで作成したモデルをそのまま設計に落としたりするには，つまりMDA的な使い方をするにはオブジェクト指向ベースの開発手法のほうが向いているようである．具体的な使い方については第17章に説明する．

4 SPINを使ったモデル検査

● SPINで読み手優先プロトコルを検査する

今度はLTSAではなく，SPINというモデル検査の世界ではメジャーなツールでモデル検査をしてみよう．

図13が実際に検査したものである．これは，「読み手が待っている間は，書き手は書き込み処理を行えない」という排他制御の実現方法を示したものである．この図の情報だけで排他制御の検査ができてしまうのである．

SPINによる検査の結果，次の事項が指摘された．

例では“読み手優先”の具体的な定義を，

(1) (`rd_count > 0`)の間，書き手タスクがクリティカル・セクションに入ることはない

としている．

この場合，読み手タスクがセマフォ`w`をリリースするまで書き手タスクの処理をブロックする構造になっているので，`rd_count`が読み込み処理中のタスク数を表現していることになる．すると，「もし読み込み処理中のタスクがいるのにもかかわらず，書き手がクリティカル・セクションに入る」ようなことがあれば，優先順位が守られていないことになる．そこで，上で定義した“読み手優先”がつねに正しいかどうかを検査したところ，以下の瞬間では“読み手優先”を満たしていないことがわかった．

(2) 読み手タスクが`rd_count = rd_count + 1`の文を実行した直後（次の`if`文の実行直前），書き手タスクがセマフォ`w`を獲得した場合

見本

つまり，rd_count が読み込み処理中のタスク数を表していないことがあるのだ．だからもし，rd_count を読み込み処理中のタスク数だと考えコードを再利用した場合，バグにつながる可能性がある．

さらに次のような改善方法も提案されている．

```
for (;;)
{    wai_sem(s);
     if (rd_count == 0) wai_sem(w);
     rd_count = rd_count + 1;
     sig_sem(s);
     ...
}
```

このようにプロトコルを変更すれば(1)の定義とプロトコルの動作が矛盾しない．

作者のもともとの意図は，rd_count は読み出しを待っているタスクの数であるが，読み手優先の定義と(1)のような形でリンクすることは考えていなかった．指摘されたように変更すればrd_count の意味が明確になるが，クリティカル・セクションが大きくなってしまうため，どちらを優先させるかは設計によって判断するべきである．

モデル検査は，すべての可能な実行パターンをしらみつぶしに検査してくれるので，(2)のような微妙なタイミングでも見逃しはしない．SPINでは，(1)の条件を，

(3) □(write → !(rd_count > 0))

のように表現する．これが時相論理式である．「書き込みのときは常にrd_count > 0ではない」，という意味である．□が“常に”を表している(残念ながらLTSAではこのような個別の条件を検査することはできない)．

時相論理は，大学の情報関係の学科でしか教えてくれないのでなじみがない．したがって，いきなりこのようなモデル検査を行おうとするとたいへんである．正しく検査できたのかどうかの判断が難しい．また，モデル検査ではすぐに状態数が天文学的数字になるので，ツールがメモリ不足で動かなくなる．この例のように，ソース・コードのレベルで記述された設計情報から必要な部分のみを切り出すには経験がいる．

＊　　　＊

海外では，学部の授業でLTSAを使った演習を行うと聞いたことがある．組み込みソフトウェア開発は日本の強みとは言うものの，相変わらずソース・コードをいじくり回す寝業(ねわざ)師と，人海戦術のテスト・チームが支えている．ソース・コードにしてから工数をかけてもそれでは手遅れだ．

参考文献

(1) Jeff Magee，Jeff Kramer，Concurrency；State Models & Java Programs，Wiley，1999.

(2) http://edutool.com/ltsa/

(3) http://www.graco.c.u-tokyo.ac.jp/~tamai/concurrency.html

(4) Hassan Gomaa；*Designing Software Product Lines with UML*，Addison Wesley，2004.

見本