組み込みプログラミング・ノウハウ入門（第35回）

何に対して何を検査するのかが重要

――LTSAのまとめと最新のLTSA

藤倉 俊幸

前回はLTSAによって作った動作モデルからイベント系列を網羅的に生成してテスト・ケースとすることを述べた．これで，要求分析からテストまでの工程で，LTSAのどのような使い方が可能か示すことができたので，今回は今までのまとめにしようと思っていた．

ところが，v3.0がリリースされてしまった．v3.0は，なんと時相論理式を使って検証ができる．時相論理式を知らなくともモデル検査ができるLTSAが，時相論理式を使ってもモデル検査できるようになってしまった．これは，かなりインパクトのあるバージョンアップである．そこで今回は，今までのまとめと，v3.0の簡単な紹介を行う．

1 LTSAで何ができるか

● ステート・マシンの合成ツールとしてのLTSA

LTSAは，イベントやアクションの順序をいくつかの「プロセス」と呼ぶ単位で指定すると，それぞれのプロセスを任意の順番で実行した場合，全体でどのような実行のしかた（実行パス）があるかを網羅的に作り出してくれる．つまり，すべての組み合わせを作り出すという力仕事をやってくれる．

この機能を利用すると，ステート・マシンの合成などができる．LTSAは，合成したステート・マシンの絵を見せてくれるので，RoseRTやRhapsodyなどのCASEツールを使用しているときに，アクティブ・オブジェクトを合成したいときなどは便利である．アクティブ・オブジェクトに対して仕様を追加する場合にも，新しい仕様をプロセス式で表現して，既存のステート・マシンと合成すれば新しいステート・マシンを得ることができる．人間が考えるよりは，速くて確実である．

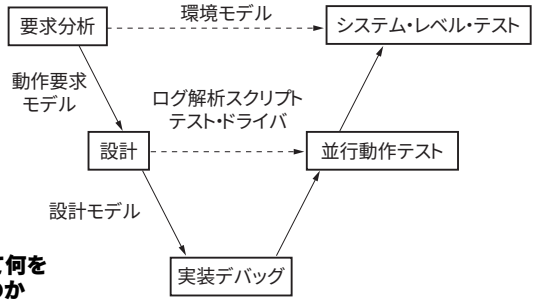

図1
何に対して何を検査するのか

● シーケンス図からもステート・マシンを作成できる

さらに，複数のシナリオを処理するステート・マシンを設計するときには，v2.3のMSCプラグインを使えば，ボタン一つで複数のシーケンス図からステート・マシンを自動生成してくれる．この連載で最初にLTSAを紹介したときに比べると便利になった．

しかし，これだけではモデル検査にはならない．合成したステート・マシンが正しいものかどうか検査して，はじめて仕事が完結する．人間が，合成したり仕様追加したりするよりは安全で確実であるが，追加する仕様の表現などに少しでもまちがいがあれば正しいステート・マシンを得ることはできない．つまり，合成が正しいことはまちがいないが，合成されたものが正しいかどうかはまだ保証されていない．だから，検査しなければならないのだ．

それでは，正しいことを確認する検査とは何だろうか．どうすればよいのだろうか．LTSAは，この部分もサポートしてくれる．

● 何に対して何を検査するのか

この検証の部分を使いこなすためには，何に対して何を検査するのかを明確にする必要がある（**図1**）．一般的には，設計を要求に対して検査する，実装を設計に対して検査すると考えるのが妥当である．もっと詳しく言うと，要求段階では，動作環境に対して動作要求を検査する．設計段階では，要求から受け取った動作要求に対して設計仕様を検査するということになる．

MDA（モデル・ドリブン・アーキテクチャ）では，要求仕様にプラットホームなどの実行環境情報を入れて設計仕様にすると説明されることが多い．しかし，この言い方だと一本道のような印象を受けるが，実際は動くモデルを作るだけでなく，それを検査するモデルも作りながら開発が進むのでレールを2本引かなければならないことになる．

● 検査式の記述法

v3.0以前のLTSAでは検査式としてプロパティを利用する．LTSAのプロパティは，動作を記述する動作モデルと同じように記述する．そして，記述した動きが仕様であると解釈される

開発局面ではプロパティとする．たとえば，要求段階で作成した動作モデルは，設計から見ると仕様なのでプロパティになる．さらに，二つのモデル間の動作に差があるかどうか確認するために，プロパティを交互に付け替えて検査することも可能である．

したがって，LTSA のモデルの書き方を習得すれば検査式も書けるようになる．これが，時相論理式ではそう簡単ではない．数学か情報かあるいは哲学関連の学部に行かないと，命題論理，述語論理，様相論理という系列を教えてくれない．時相論理は様相論理の仲間である．

● LTSA での検査法

LTSA ではプロパティを使って，動作を記述したモデルの動きがプロパティで指定した動きから逸脱しないことを検査できる．言い換えると，プロパティとして仕様を記述しておけば，仕様から外れた動作が起こるかどうかを検出できる．この検査は，Safety 検査と呼ばれる．検査としては，Safety のほかに liveness 検査と呼ばれるカテゴリがある．Safety が悪いことが起きないことを確認する検査であるのに対して，liveness 検査は指定したことが起きることを確認する検査である．liveness 検査には，指定した動作をすべてくまなく実行できるかどうかという大雑把（おおざっぱ）なものから，ある動作やイベントが起きたらある指定された動作を実行するかどうかという特定の条件下の応答性や，ある状態が特定のイベントが起こるまで継続するかなどを調べる検査までいろいろである．

LTSA のプロパティ検査は，プロパティとして指定したステート・マシンからはみ出す動作があると検出できるが，プロパティに指定された動作をすべて実行するかどうかは検査してくれない．別の言い方をすると，プロパティとして指定したステート・マシンのすべての状態を通ったかどうかなどは検査してくれない．

LTSA ではプロパティ検査のほかに progress 検査を実行できるが，これは上記の liveness 検査の中の大雑把な検査に対応するだけで，応答性などの検査はできない．時相論理式を使わないとこれらの検査はできない（ただし，v3.0 からはできるようになった）．

そこで，どうするかというと，プロパティにダミーのアクションを忍ばせて，プロパティとしてではなく普通のステート・マシンとして合成して progress 検査を行うなどのくふうが必要になる．

LTSA のモデルは，プロセスと呼ぶひとまとまりのイベントや動作の順序を記述したものを構成要素として，それらを組み合わせて作る．プロセス間で同じ名前のイベントや動作があると同期を取って実行される．名前が違うと勝手に実行される．ここで，実行と言うのは，ステート・マシンでいえば遷移することに対応する．同期を取って実行というのは，同時に遷移が起こるということである[注1]．

共有されているイベントは，同時に実行される．同時に実行するためには，それを共有しているすべてのプロセスが，それを実行可能な状態になっていなければならない．実行可能な状態にあるというのは，LTSA の図でいうと，その名前の矢印（イベント）が出ている丸（状態）にプロセスがいなければならない（アニメーションで赤くなる）ということである．そうでないとそのイベントの実行は禁止される．これが LTSA の動作原理である．

LTSA の動作原理を，タスク間の通信などに適用してモデルを作るとき，注意しなければならないのは，同期通信であるということと，ブロードキャストにならないようにするということである．要求レベルのモデルを作っている場合には，それほど問題にならないが，設計レベルのモデルの場合は，関数呼び出しやメッセージ通信を 2 点間の相互作用としてモデル化しなければならない．また，非同期通信を利用する場合は，キューイングするためのプロセスを用意しなければならない．

ただし，このようなことをやっていると状態数が増えてしまい，だんだん扱いにくいモデルになって，最後は状態爆発してしまう．状態爆発が起きると，そこまでの苦労は水の泡になる．そのため，LTSA でモデル化する対象は，アーキテクチャ・レベルにするのが安全である．詳細設計レベルのモデリングと検証は，コンポーネントに分割して行うようにすることで状態数を減らすようにする．

2 要求段階でのモデルの作成

● 環境モデルの作成

組み込みシステムの特徴は，実世界との物理的な相互作用があることである．実世界は，人間や物などで構成されている．物理的な相互作用というのは，ボタンが押されたり，温度を測ったり，物を動かしたりすることである．実世界を構成するものは，それぞれ独立に，並列に，好き勝手に動くことができる．どの程度好き勝手に動くのか知りたい場合は，一つ一つの構成物の動きを表現して，それらを並列合成して確認することができる．これを環境モデルと呼ぶことにする．

一つの構成物の動きの表現を一つの LTSA プロセスに対応させる．そして，それらを合成すると環境モデルになる．環境モデルは，これから開発しようとするシステムが置かれる環境を表現したものになる．オブジェクト指向のユース・ケース・モデルに害されていると，構成物とプロセスが一対一に対応していないと後ろめたい気がするかもしれないが，そのようなことにこだわる理由はまったくない．動きの仕様としてまとまっているほうが重要である．「一つの構成物の動きの表現を一つの LTSA プロセスに対応させる」という表現は，構成物間の一つの動きの表現を一つの LTSA プロセスに対応させるといったほうが，上級者向きである．

注1：以降，イベントやアクションのことを，めんどうなのでイベントと呼ぶことにする．また，同じ名前のイベントが複数のプロセスにある場合，イベントを共有しているということにする．