April 5, 2010

RT0900
Computer Science 8 pages

Research Report

A Method for Verifying Behaviors in SysML Models

Shinichi Hirose

IBM Research - Tokyo
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

SysML モデルにおける振る舞いの検証方法

A Method for Verifying Behaviors in SysML Models

広瀬 紳一
日本アイ・ビー・エム株式会社 東京基礎研究所

はじめに

対象がソフトウェアであるかハードウェアであるかを問わず、多くの開発プロジェクトで（UML もしくはそれを拡張した）モデルが利用されるようになってきている。これは、ソフトウェア開発でのモデル駆動開発、といった最終生成物への変換を意図したものはもちろん、そうでない場合でも、一般的な効果として、記述の曖昧さを低減できるという効果が評価されているためだと考えられる。また、モデルに記述された内容（たとえ不完全なものでも）に基づいてシミュレーションをおこない、モデルの記述が正確であるか、また、モデルのもととなった仕様書に誤りがないか、等をチェックすることは、後工程で問題が発見されておきる手戻りのコストをおさえるための有効な手段である。

ソフトウェアやハードウェアが複合的に組み合わされたものを対象とするモデリング言語に SysML[1]がある。SysML では、対象物の構造をブロック定義図(Block Definition Diagram; BDD)という固有の図を用いて表わす。ブロックは、SysML の基本的な要素のクラスで、製品や部品、ソフトウェアモジュールをあらわすものである。また、あるブロックに注目したときの、その部品（子ブロック）同士の関係を内部ブロック図(Internal Block Diagram; IBD)と名づけられた図を用いて表現する。図 1~2 に各々の図の例を示す。

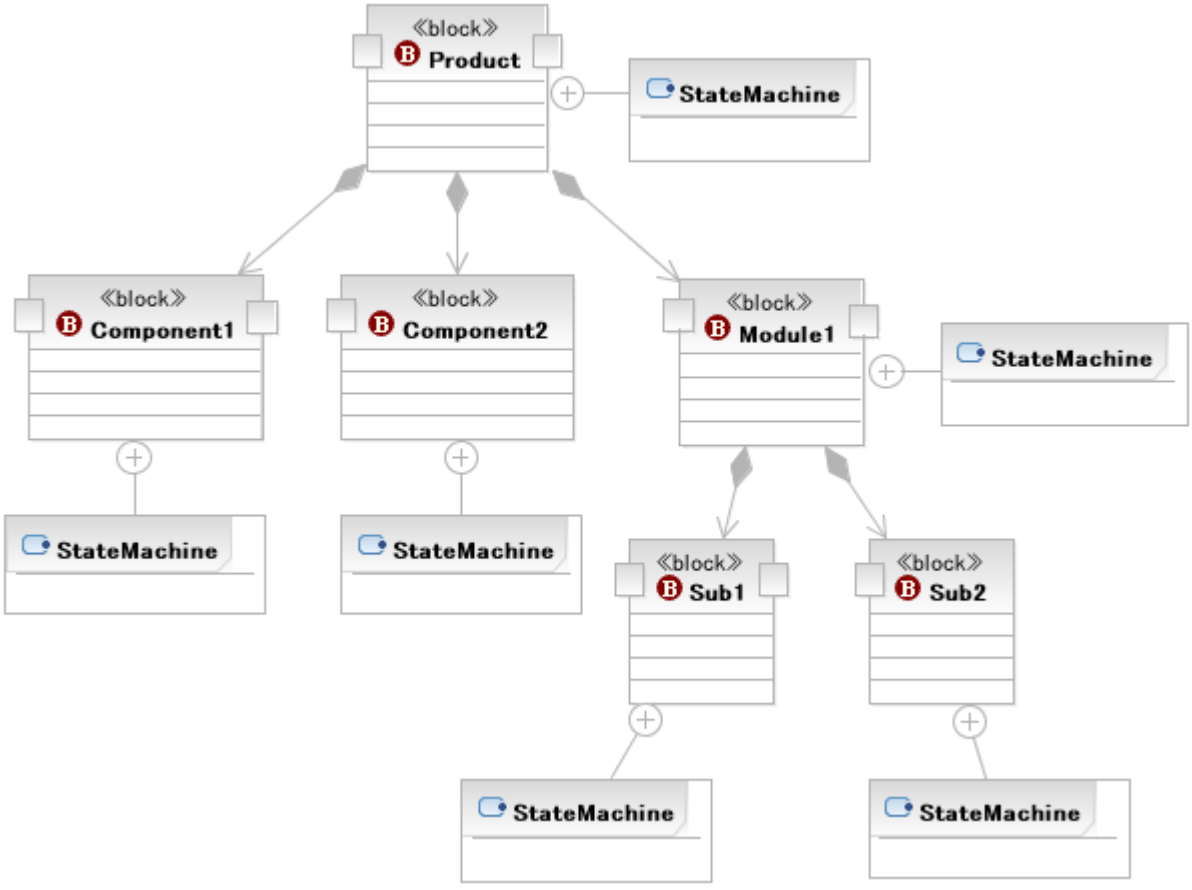

図1. ブロック定義図

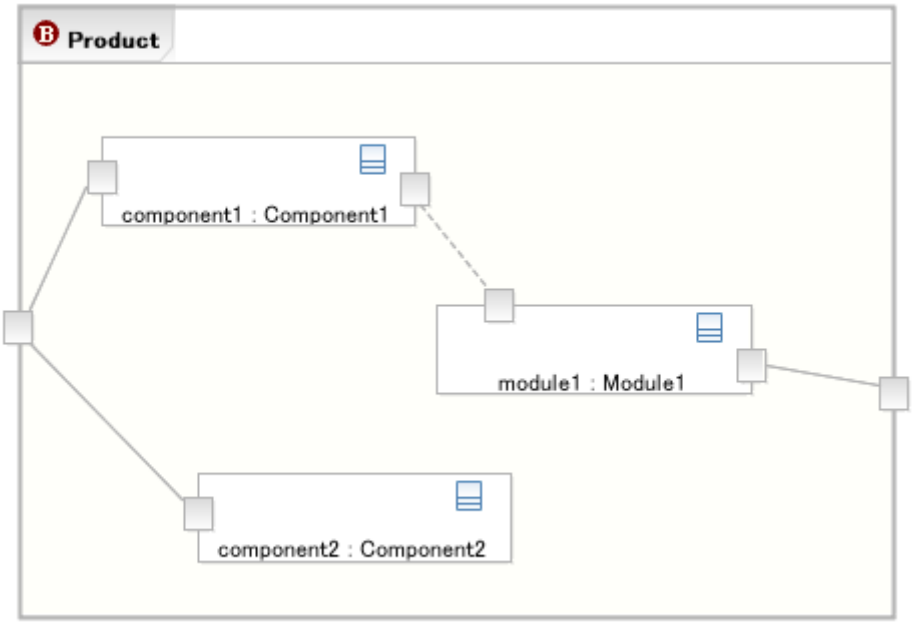

図2. 内部ブロック図

図1は、トップレベルのブロック(Product)が、3つの部品(Component1, Component2, および Module1)から構成されており、Module1 は、さらに2つの部品(Sub1 および Sub2)に分解されることを表現している。また、ここでは、各ブロックに、その主たる振る舞いとして状態機械が付属している。図2には、Product ブロックの各部品が、どのように信号のやりとりをするのかを、ポート(ブロックの辺上の小さな四角形で表現される)とポート間を結ぶコネクターを用いて表現されている。

SysML を用いた設計手法のひとつに、「段階的な詳細化」がある。これは、設計対象物をその機能を保ったままサブシステムの組み合わせに分解し、さらに、各サブシステムを下位のサブシステムや部品の組み合わせとして実現する、というようにして設計をすすめるものである。具体的には、まず、ブロック定義図での木の根となるトップレベルのブロックを置き、もともとの仕様書で規定されたモデリング対象物の性質を、ブロックの属性や振る舞いとして記述する。次に、そのブロックを実現するための部品やモジュールをあらわすブロックを子ブロックとして置き、さらに、その子ブロックにも属性や振る舞いを定義する。そして、それらが協調して動作することによって、親ブロックと同じ性質を示すことが確認できれば、子ブロックへの分解は完了、ということになる。同様のステップを繰り返して、実現可能性が自明なブロックにまで分解できれば、システム・レベルでのモデリングが完成する。通常、ブロック全体の主たる振る舞いの表現として状態機械図が用いられる。そして、状態機械の遷移に伴う動作や状態への出入りに伴う動作はアクティビティー図を用いて表現する。ただし、ブロックの主たる振る舞いが非常に単純な場合には、単体のアクティビティー図のみで十分な場合もある。

以上の議論より、ここでは SysML モデルの振る舞いの検証とは、親ブロックと、その部品である子ブロック群について、親ブロックの振る舞いと、子ブロック群を親ブロックの内部ブロック図に従って組み合わせたものの振る舞いが等価であることを、すべての親ブロックについてチェックすることである、というこだと考えることができる。本報告では、このような検証を実行方法について検討する。

検証の前提

与えられた SysML モデルが、その振る舞いに関して全体として整合性があることを確認するための手法について考える。ここで、全体の整合性とは、すべての子ブロックへの分解について、親子間での振る舞いの等価性が保たれているという意味である。また、不整合が発見された場合には、その事例を報告することもおこなう。検証の対象となるモデルには、図1に示すような、振る舞いが定義された SysML ブロックのコンポジション関連による木構造があり、部品を持つブロックでは、その内部構造(図2)が記述されていることを前提としている。

振る舞いが等価であることを確認するためには、モデル実行系を利用し、テストケースを用いたテストをおこなって、各テストケースに対する反応が同じであることをチェックするものとする。したがって、テストケースにあらわれない刺激に対する反応の等価性は検査されないことに注意が必要である。テストに用いるテストケースやその集まりであるテストスイートを作成する方法は、人手によるものや状態機械を走査して適切と考えられるものを自動的に生成する方式が各

種提案されており、テストスイートはそれらの手法を用いて、あらかじめ準備されているものとする。具体的には、テストケースは、ブロック(のインスタンス)に送付されたときに、その振る舞いを活性化させて所定の動作をおこなわせるためのイベント列である。

出力としては、不整合が生じた場合には、そのときに着目していたブロックの集合と、ひとつ以上のパスしなかったテストケースとなり、全体としては、その集合、つまり

```
（検査結果）＝｛（不整合事例）｝
（不整合事例）＝［｛（ブロック）｝，｛（テストケース）｝］
```

である(ただし、{…}は集合、[…]はタプルを表わす)。よって結果が空集合であれば、整合性が確認されたということになる。

一般のモデル実行系では、モデルに含まれる要素のすべてを対象に実行がおこなわれる。ところが、ここで注意が必要なのは、SysML モデルでは、親ブロックと子ブロック群は全く同じものを表現しているという点で、したがって、モデルの実行をおこなうにあたっては、親子関係にあるブロックの振る舞いが同時に出現するようなモデル実行の構成には意味がない。これを言い替えると、「コンポジット関連の木の根を含む部分木のすべての葉」が意味のある部品のセットということになる。よって、モデリングされた製品をの振る舞いを検証するためには、適切な部品のセットを見出し、その複合体が所定の動作をすることを確認することが必要でとなる。次節で、これを実施するようにモデル実行系を制御するアルゴリズムを検討する。

検証手法

モデルの開発環境として、Rational Software Modeler (および、SysML モデリングのための拡張機能 EmbeddedPlus SysML Toolkit)と、その上でのモデル実行のためのアドオンである MEX[2]を用いた場合の、検証の詳細な手順を述べる。なお、これらの開発ツールはあくまで例として用いているものであり、検証実行のアルゴリズム自体がこれらに依存しているものではない。

必要となるモデル実行系の機能を以下に列挙する。

- ブロックのインスタンス生成
- 振る舞いのインスタンス生成
- 生成されたインスタンス群に対して、イベントを送付する

これらの機能を利用してモデルの検証を実行するために、図3に示すように、テストケースを読み取って入力を供給するブロック(EventGenerator)と、出力を記録するブロック(EventLogger)を、検査対象のブロック(下図の例では Product)に結合したもの、および、これらを下記のアルゴリズムにしたがって制御する機構を用いる。なお、図3では、トップレベルのブロックしか描かれていないが、図3は、内部ブロック図であり、その他の子ブロックは、ブロック Product の内部に(Product の内部ブロック図である図2に示されているような形で)存在していることに注意されたい。

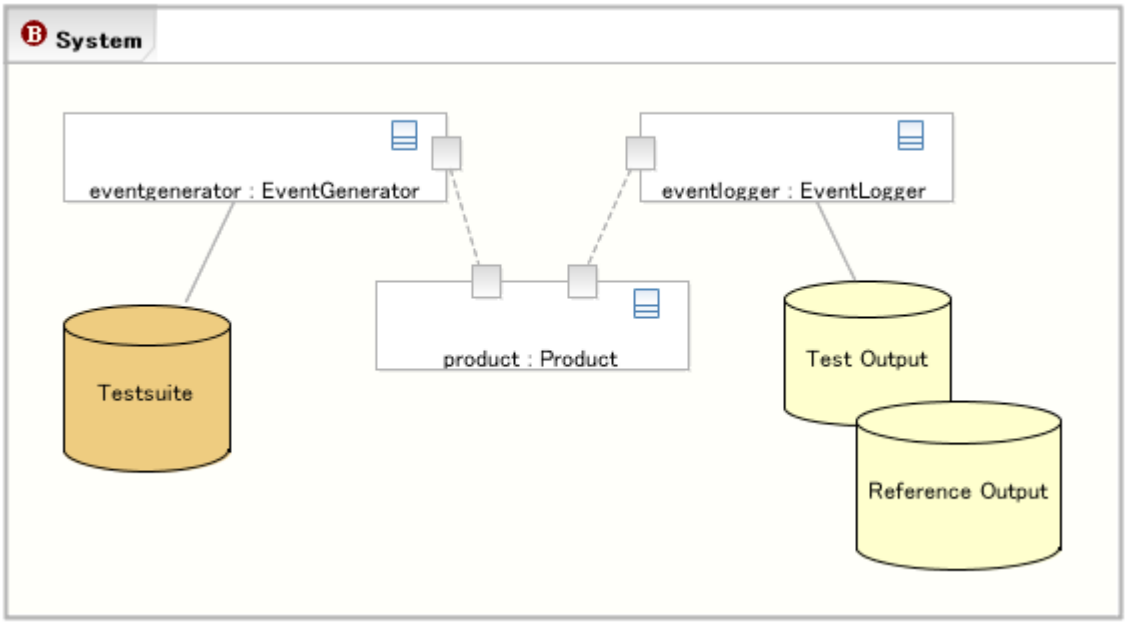

図3.

テスト結果を得るためのアルゴリズムを以下に示す。

1. 初期化：全ての子を持つブロックに”未検査”のマークをつける。また、不整合事例をクリアする。
2. 木の根のブロックに対して、テストケースにしたがってイベントを送付し、出力イベント列を保管する（これを参照出力と呼ぶ）。このブロックの未検査マークをクリアする
3. 木を深さ優先で探索し、未検査マークのついたブロックを得る。そのようなブロックがなければ不整合事例を値として終了する。
4. 得られたブロックの検査をおこなう。
   4.1. 初期化：検査対象のブロックとその祖先に”要展開”マークをつける。その他のブロックの要展開マークはクリアする。
   4.2. 木の根のブロック（必ず要展開マークがついている）を展開アルゴリズム（後述）によって展開する。これは再帰的なアルゴリズムである。
   4.3. 展開によって得られた、ブロックのインスタンスと、振る舞いオブジェクトのインスタンスに対してテストを実行し、各テストケースに対する出力イベント列を参照出力と比較する。
   4.4. 結果が一致しなければ、現在のブロックのインスタンスとテストケースの情報を不整合事例に追加する。また、検査対象のブロックのすべての子孫については、（誤ったブロックの分解が含まれるため）テストを実行する意味がないので、未検査マークをクリアする。
5. 検査したブロックの未検査マークをクリアし、3.に戻る。

テストに必要なブロックと振る舞いオブジェクトのインスタンスを生成するための展開アルゴリズムを以下に示す。

1. 与えられたブロックのインスタンスを生成する。
2. ブロックに要展開マークがついていなければ、振る舞いオブジェクトのインスタンスを生成し、5.へ。
3. すべての子ブロックに対して、この展開アルゴリズムを再帰的に適用し、インスタンスを生成する。
4. ブロックの内部ブロック図に記述されたポート間のコネクターのインスタンスを生成し、子ブロックのインスタンス同士もしくは、対象ブロックのインスタンスと子ブロックのインスタンスを結合する。
5. （1.で生成した）対象ブロックのインスタンスを値として戻る。

ここまで述べてきた検証方法においては、木の根のブロックの振る舞いとの比較をおこなっているが、この応用として、ある部分木についてのみの検証をおこなうことも可能である。具体的には、その部分木の根のブロックについて、それ

が展開されないようなブロック展開がなされたときに、そのブロックの各テストケースでの、入力イベント列と出力イベント列を記憶し、前者をその部分木用のテストケース、後者を参照出力として用いればよい。

検証の実行例

簡単な例を用いて、モデルの検証がおこなわれる様子を説明する。ここで使用するサンプルのモデルは、図 4 のブロック定義図を持つ二桁の十進カウンターである。

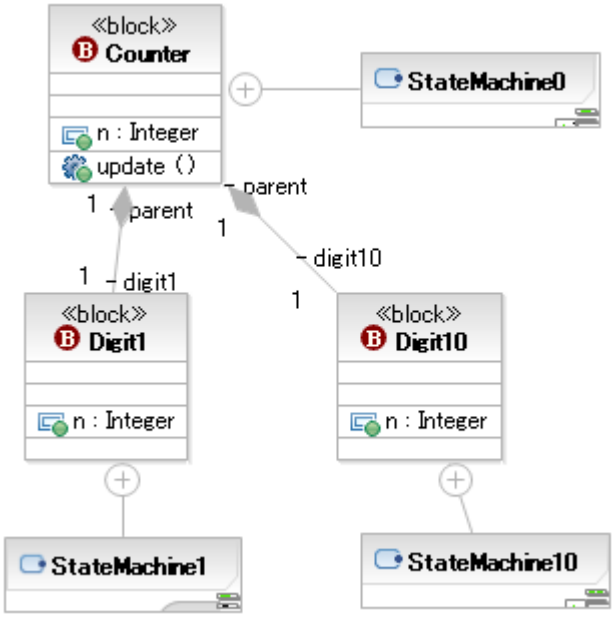

図4.

トップレベルの Counter ブロックは、ブロックプロパティーとして n:Integer を持ち、その初期値は 0 である。また、ポート port を通してイベントを受け取ることができる。さらに、ブロックの振る舞いとして、StateMachine0 が指定されている。この StateMachine0 は、状態機械であり、図 5 のような遷移が定義されている。

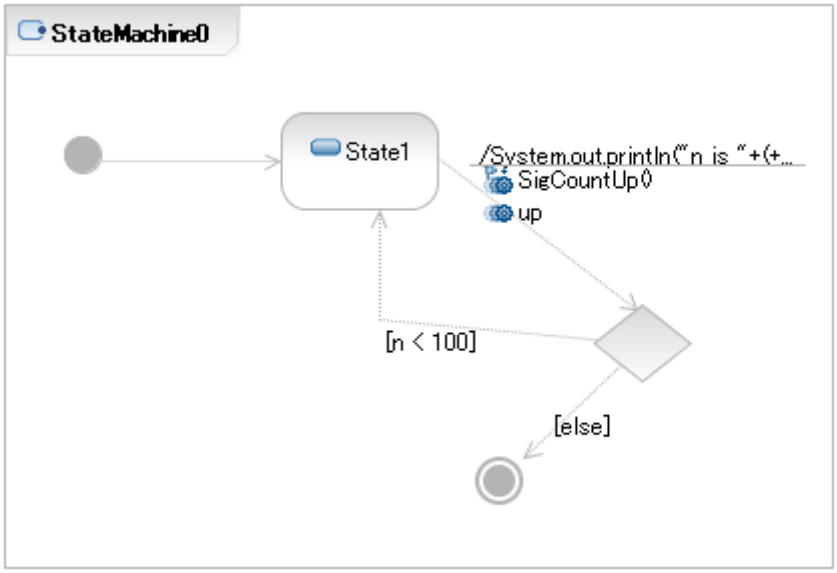

図5.

図にはあらわれていないが、State1 からの SigCountUp シグナルによる遷移の効果 up には、プログラム・フラグメント

```
System.out.println("n is "+(++n));
```

が定義されており、遷移が起こるたびに、

```
n is 1
n is 2
…
```

というログが実行トレースに出力されることになる。

検証の最初の段階である、参照出力を得るための実行では、ブロック Counter のインスタンスが、StateMachine0 を持った状態で生成され、実行される。このためのテスト・ドライバーは、以下のようなものとなる。

```
Counter obj = new Counter(true);// true: ブロックの振る舞いを含める
sendEvent(obj.port, new EvCountUp());   // EvCountUp はシグナル SigCountUp を持ったシグナル・イベント
sendEvent(obj.port, new EvCountUp());
sendEvent(obj.port, new EvCountUp());
…       // テストケースに定められた回数だけイベントを送付する
sendEvent(obj.port, new EvCountUp());
```

この時のログは、上述のように、

```
n is 1
n is 2
…
```

となり、これが参照出力となる。

続いて、ブロック Counter を展開した場合の検証について説明する。図 4 にあるように、Counter は、Digit1 と Digit10 に展開される。これらのブロックも振る舞いが定義されており、展開時には、Counter に加えて、Digit1, StateMachine1, Digit10, StateMachine10 のインスタンスを用いることになる。ここで、ブロック Counter の内部ブロック図は図 6 のように定義されているとする。

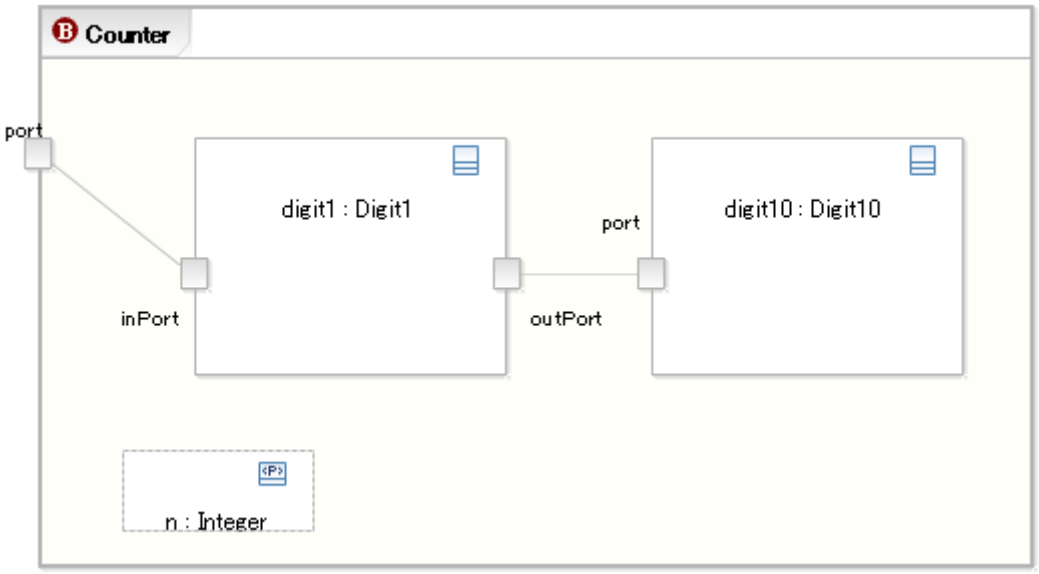

図6.

また、StateMachine1 と StateMachine10 の定義は、それぞれ図 7、図 8 であるとする。

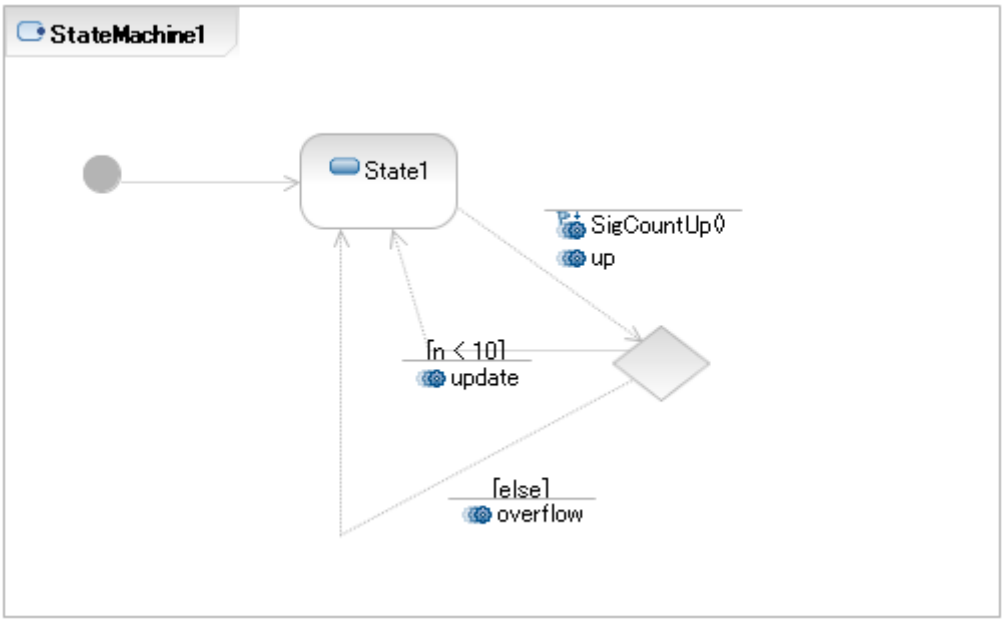

図7.

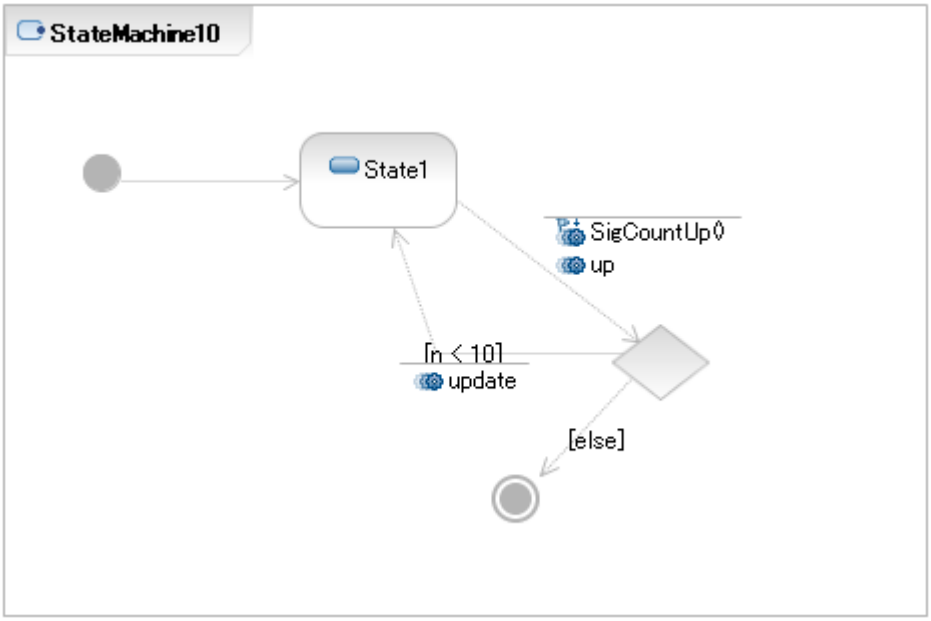

図8.

図 7 の StateMachine1 において、State1 でのシグナル SigCountUp に対する遷移は、効果 up（StateMachine0 の効果と同じ名前であるが、効果は状態機械ごとの定義であることに注意）を持ち、その定義は、プログラム・フラグメント

```
++n;
```

である。また、それに続く条件分岐で、n<10 の場合には、効果 update

```
parent.update();
```

が実行され、それ以外の場合には、効果 overflow

```
n = 0;
send(outPort, new EvCountUp());
```

が実行されるように定義されている。ここに、parent は、親である Counter ブロックを得るためのプロパティー（図 4 のコンポジション関連によって定義されている）、outPort は、（図 6 でも図示されている）Digit10 にシグナルを送るのに用いるポートである。したがって、前者では、親ブロックの update オペレーションを呼び出して、親ブロックの状態を更新する

のであるが、このオペレーションは、

```
n = digit1.n + digit10.n * 10;
System.out.println(“n is “+n);
```

と定義しておくことにより、展開された場合の振る舞いを展開前と同等のものとすることができる。

同様に、図 8 の StateMachine10 では、State1 でのシグナル SigCountUp に対する遷移の効果 up と、それに続く、n<10 の場合の効果 update は、StateMachine1 と全く同じものが定義されているとする。

ここで、内部ブロック図を考慮して生成されるテスト・ドライバーは、

```
Counter obj = new Counter(false);          // false: 振る舞いは含めない
obj.digit1 = new Digit1(true);
obj.digit10 = new Digit10(true);
obj.port.connectInternal(digit1.inPort);   // Counter の port に来たイベントは Digit1 の inPort へ転送する
digit1.outPort.connect(digit10.port);      // Digit1 の outPort からのイベントは Digit10 の port に与えられる
// 以下の、テスト用のイベント列は全く同じものを用いる
sendEvent(obj.port, new EvCountUp());
sendEvent(obj.port, new EvCountUp());
sendEvent(obj.port, new EvCountUp());
…
sendEvent(obj.port, new EvCountUp());
```

となる。これまでに述べたように、振る舞いが定義されている場合には、実行トレースに含まれるログは、やはり、

```
n is 1
n is 2
…
```

となり、ブロック展開が正しくおこなわれていたことが検証できた。

おわりに

SysML を用いての段階的詳細化による設計の検証手法について述べた。例では、離散イベントでのテスト実行を用いたが、連続系のプラントモデルに対しても、実行結果の一致の判定で適切に設定された範囲内の誤差を許容することで同様の手法が適用可能である。

参考文献

[1] SysML: http://www.omgsysml.org/

[2] MEX: http://www.haifa.ibm.com/projects/software/ple/mex/
