

BYOBによるモデル検査入門*

神戸大学人間発達環境学研究科

高橋 真

この資料はビジュアルプログラミング環境の BYOB(Build Your Own Blocks)¹を利用してモデル検査の初歩を分かりやすく解説することを目的としています.

*この講義資料は JSPS 科研費 23650507 及び 26560089 の助成を受けた研究の過程で得られたものです.

¹Build Your Own Blocks <http://snap.berkeley.edu/>

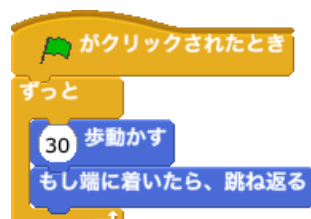
目次

1	並行性	1
1.1	並行に動作するプログラム	1
1.2	BYOB のスクリプトの実行順序について	2
1.3	意図しない結果になるプログラムの例	4
2	有限 1 次元セルオートマトン	6
3	かんたんなモデル検査	10
3.1	状態遷移図による検査	10
3.2	時相論理について	12
4	SPIN のかんたんな使い方	13
4.1	プロセスの記述	14
4.1.1	変数	14
4.1.2	プロセスの記述その 1	14
4.1.3	do 文	14
4.1.4	if 文	16
4.2	性質のチェック方法	16
4.2.1	assert 文	16
4.2.2	LTL 論理式	18
5	複数プロセス	22
5.1	複数プロセスの記述	22
5.2	排他制御	24
5.3	具体的なプログラムの検証	25
5.3.1	BYOB スクリプトの解説	25
5.3.2	Spin による解析	29

1 並行性

1.1 並行に動作するプログラム

BYOB では一つのスプライトに対して複数のスクリプトを作成することが可能であり、またスプライトも複数あるとさらに多くのスクリプトが作成される。これらは一度に動作することが可能であり、その場合複数のスクリプトが同時に動作していることになる。このようにシステムが複数のプロセス（スクリプト）から構成され、それらが同時に動作するようなシステムを並行システムという。たとえば、BYOB で次のスクリプトをもつ 2 つのスプライトを作成して緑旗をクリックすると二つのスプライトは同時に動き出す。



二つのスクリプトに共通の資源（変数など）がない場合は並行に動作しても問題はおきないが、共有資源があると単体で動作する時には発生しなかった問題がおきることがある。例えば、次のようなスクリプトを作成する。



変数 `var` の値を 0 にして、これを実行すると変数 `var` の値は 10 で停止する。さらに、このスクリプトを複製して二つ同時に動かすと変数 `var` の値は 20 になり停止する。

このスクリプトを繰り返しブロックを用いて次のように変更する。

追加したものは単にブロック内を 1 回実行するだけなので、もとのスクリプトと実質的な内容は同じで変数 `var` の値は 10 で停止する。従って、これをそのまま動かしても特に問題はおきない。しかし、このスクリプトを複製して同時に二つのスクリプトを動かすとどうなるか。それぞれのスクリプトは変数 `var` の値を 1 増やす操作をそれぞれ 10 回行うものであるから、最初のスクリプトと同じで変数 `var` の値が 20 になることが期待されるが、実際に動かしてみると変数 `var` の値は 10 で停止してしまう。単体では同じ結果を生むのに、複製して二つ同時に動かすと異なる結果を生む。これが並行に動作するシステムに潜む大きな問題を端的に表している。



なぜこのようなことが起きてしまうのか。これは後者の場合二つのスプライトの命令がインターリーブ (interleave) して交互に実行されるためである。つぎのように、いまどこを実行しているかわかるように各命令の間にリストへのテキスト出力を行うようにしてみると確かに交互にスクリプトが動作していることが確認できる。



図 1: インターリーブして交互に動作する

上で確認したようにプロセスが複数互いに変数を共有したりメッセージを交換して動作する場合は単体では起こりえなかった問題が出現する。最初にあげた例のような場合はプログラムを作成する際に問題を見つけることが可能であるが、もっと複雑なプログラムになると単に動作を追うだけではわからない問題が潜む可能性がある。これが個人的に使うプログラムであれば不具合があっても自分の問題で済むが、社会において重要な役割を担うシステムでは、常に高い信頼性が求められる。そのような重要なシステムでは、設計の段階からシステムの動作に問題が無いことを検証しておく必要がある。

この講義では、このような並行システムにおける問題をモデル検査ツール Spin を利用して検証する仕組みについて学ぶ。モデル検査とは「状態空間の網羅的な探索によって、当該システムが与えられた仕様を満たすか否かを判定する方法の総称」(中島 [5]) である。

1.2 BYOB のスクリプトの実行順序について

並行性に潜む問題を BYOB のスクリプトを例にあげながら学んでいくが、その前に BYOB のスクリプトの命令がどのような順序で実行されるか確認をしておく。BYOB で複数のスクリプトが同時に動作する場合、それぞれのスクリプトが順に実行されるが、スクリプト内に制御タイルで最後に→があるものは制御がこのタイルの最後まで来たときは、いったん次のスクリプトに実行が移る。例えば、上であげた図 1 の (a) のスクリプトを考える。



図 2: 実行順序

緑旗がクリックされると最初に右側のスクリプトが動き、「1 回繰り返す」を終えた時点で左のスクリプトが動作し「1 回繰り返す」を行う。再び右側のスクリプトが動き次の「1 回繰り返す」を実行して、左のスクリプトに実行が移り「1 回繰り返す」を行う。また、右のスクリプトに実行が移ると「10 回繰り返す」タイトルの最後に→があるので、左に実行が移る左も同様に「10 回繰り返す」タイトルの最後に→があるので、そのまま右へ移る。右のスクリプトは「10 回繰り返す」の先頭にもどり同じことを繰り返す。このことは最初に確認したように labels リストの内容をみることも確認できる。この実行順序はスクリプト内に自分で作成したブロックがある場合も、そのブロック内の定義中に最後が→であるタイトルを含む場合は同様にその時点で制御が他のスクリプトに移動する。例えば、上で作成したスクリプトを用いて、次のようにブロック left, right を作成してみる。ただし、上部のアトミックのチェックは入れないで置く。



これを用いて、次のようにスクリプトを作る。



このスクリプトを実行すると最初と全く同じ結果になる。

しかし、ブロックがアトミック指定されている場合はこのブロックの実行中に他のスクリプトが動くことはない。実際に上の left と right のブロック定義で上部のアトミック指定にチェックを入れてからスクリプトを実行すると、リスト labels は右図のようになり、interleave がおきていないことがわかる。また、「待つ」タイルである場合は、指定された時間が経過するまでは他のスクリプトが動作する。

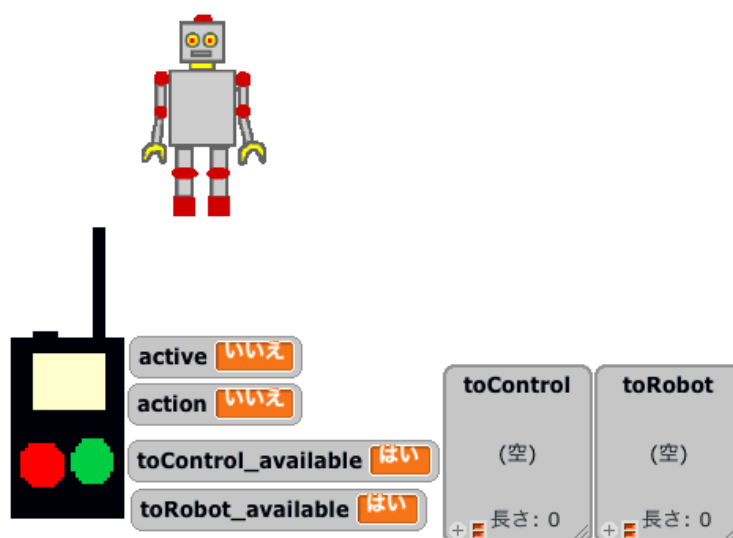


1.3 意図しない結果になるプログラムの例

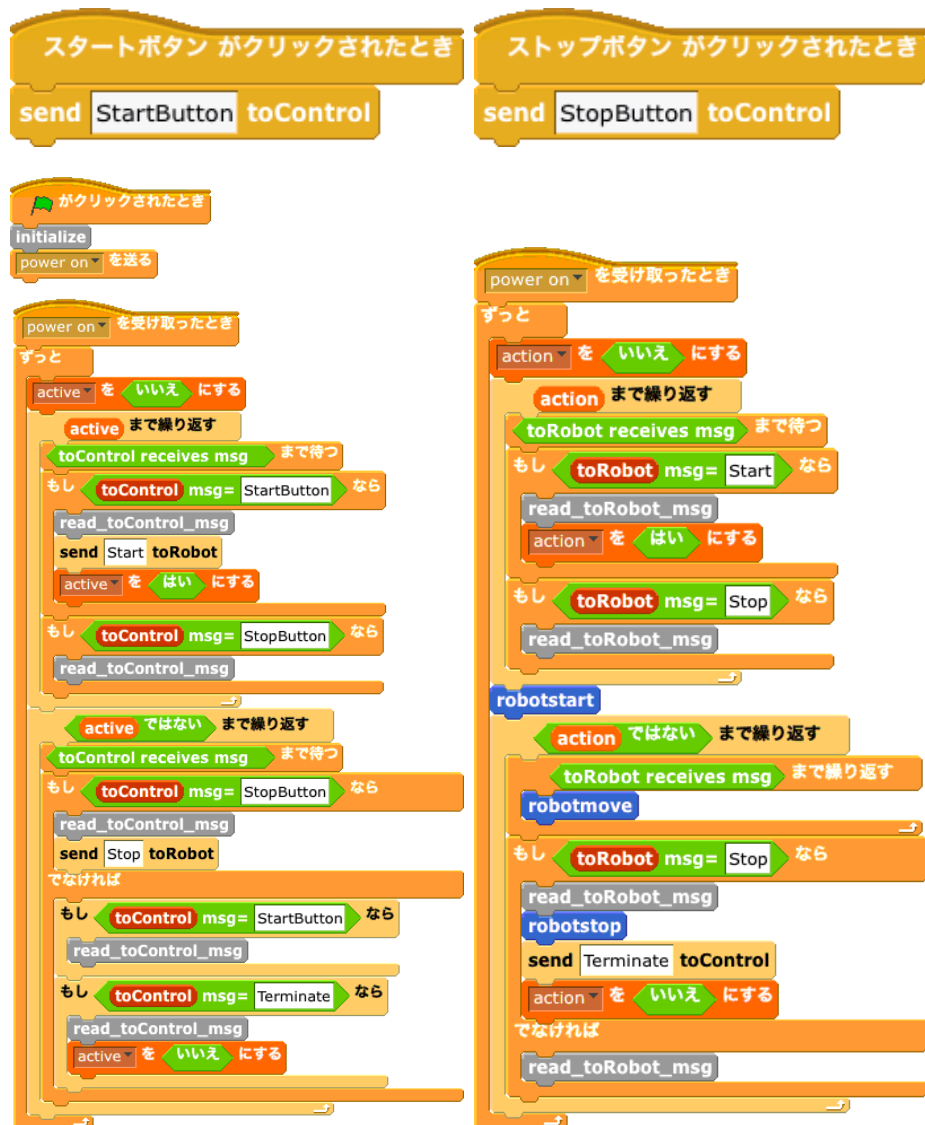
単純なスクリプトであれば並行に動作していても実行順序の予想はつくが、長いスクリプトがある場合やメッセージのやり取りをする場合などは実際にどのように実行が経過していくかは予想がつかず意図しない結果になる場合もある。

次のプログラムは、リモコンを用いてロボット操作を行うものである²。4つのスプライトと5つのスクリプトからできている。

プログラムの詳細は5.3で解説するが、ここではロボットとリモコンの間を高々長さ1のふたつのリスト toRobot と toControl を使用してデータの送受信を行う前提でプログラムが書かれている。



²この例は [10] で [5] の検証例を下に作成したものをこの講義資料用に改変したものである。



プログラムは次のように動くことを意図している。

- ユーザがスタートボタンを押すと、toControlに StartButton が送られ、リモコンは toControl から StartButton を読み込むと toRobot に Start を送る。
- ロボットは toRobot から Start を読み込むと動き始める。
- ユーザがストップボタンを押すと toControl に StopButton が送られ、リモコンはそれを受けて toRobot に Stop を送る。
- ロボットは Stop を受け取ると動作を停止し、toControl に Terminate を送る。
- リモコンは Terminate を受け取ると待機状態に戻り、ユーザがボタンを押すのを待つ。

スタートボタンとストップボタンを通常の使い方でクリックする限りでは正常に動作するため特に問題はないように思えるが、ロボットが動いているときに、ストップボタンを連打するとデッドロックをおこして動かなくなる。なぜ、このようなことになるかは5.3でモデル検査ソフトの Spin を用いて分析する。

2 有限1次元セルオートマトン

セルオートマトンは、チューリング機械と同様に両側に無限にのびたテープをもち、各セルは近傍（例えば両隣と自分）のセルの値から次の値を計算していく。例えば、テープ記号として0,1を使い、次の時刻のセルの値の計算を両隣のセルの値から次の規則

(左, 自分, 右)	000	001	010	011	100	101	110	111
次の値	0	0	1	0	1	0	1	0

により決めるとする。このとき、テープ上に1がひとつだけあるテープをこの規則を用いて計算すると、（両側に無限にある0は無視して記載すると） $010 \rightarrow 1, 100 \rightarrow 1, 000 \rightarrow 0$ より、100000は110000となる。これを繰り返すと

100000
110000
011000
001100
000110
...

と変化していく。

両側無限でなく有限の長さのテープにすると端点の規則（境界条件）をどうするかが問題になるが、端と端をつないでリング状に考えたり（周期境界条件）、端と同じ状態を仮想的に隣に考えたり（反射境界条件）、特定の値に固定して考えたり（固定境界条件）する。

ここでは、マス目が10個のテープで周期境界条件により規則

000	001	010	011	100	101	110	111
0	1	1	0	1	0	0	1

を適用した例をあげる。

1000000000
1100000001
1010000010
1011000110
1000101000
...

となる。

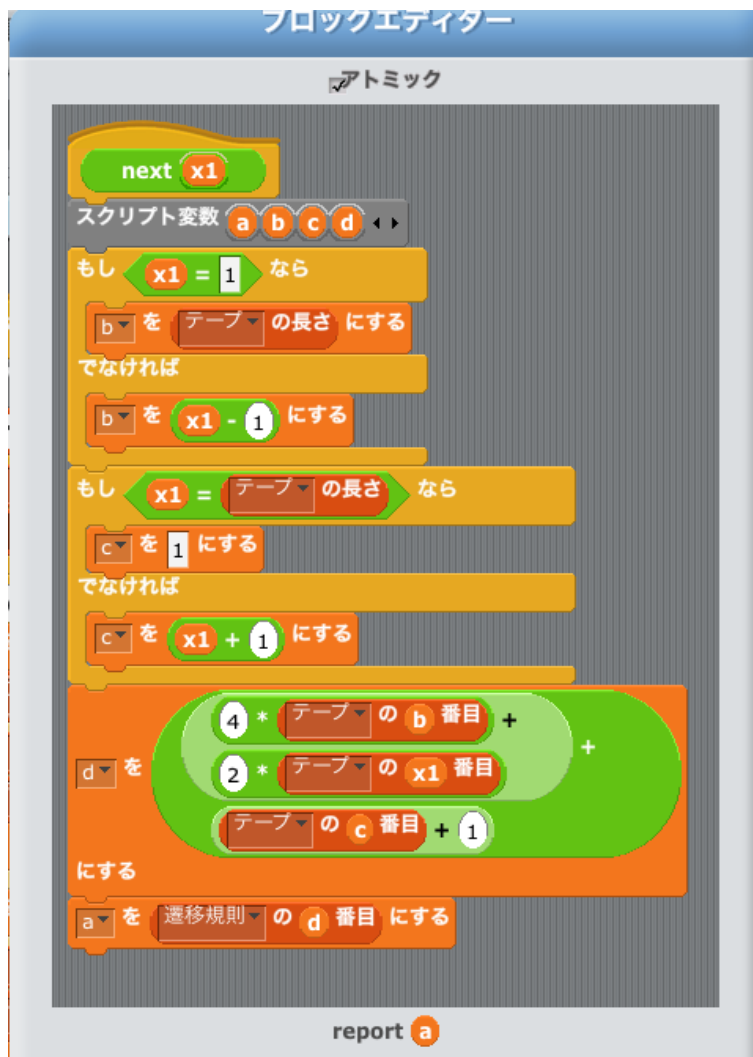
問題 2.1 上の遷移を実際に計算して確認せよ。

手計算では大変なので、上記の設定でセルオートマトンの時間発展を表示するプログラムをBYOBで作成してみよう。

まず、遷移規則、テープ、（次の時刻のテープ内容を記録する）仮テープ、時間発展のリストを作成する。遷移規則は000,001,010,011,100,101,110,111の順でリストに値を入れて定義することにする。それぞれを2進表示と考えて10進表示にすると0,1,2,3,4,5,6,7

となるので、それぞれに対応する値は2進表示から10進表示を求めて1足して遷移規則の対応する値を調べることで得られる。例えば $(100)_{10} = 4$ であるから、遷移規則の5番目に100の対応する値が書かれていることになる。

テープの i 番目のセルの次の時刻の値を計算する関数ブロック `next` は次のように定義できる。周期境界条件で考えるため最初に両端の処理を行っていることに注意する。



これを利用して次のようにスクリプトを作成する。



時間発展の回数を 20 回に固定しているが、自由に変更したい場合は、変数を定義してそれを利用するようにしたらい。

さきほどの例を実際に実行してみると次のように時間発展が得られる。

問題 2.2 上記の BYOB スクリプトを作成して、遷移規則や初期値（長さも含めて）をいろいろ変更してセルがどのように変化するか調べよ。

ここで次のような問題を考える。与えられた遷移規則とテープの初期値に対して、テープのセルがすべて 1 になることがあるかどうか調べる。例えばテープの長さが 10 のときはテープの内容は高々 $2^{10} = 1024$ しかないので、1024 回書き換えを実行すると途中の異なる時刻に同一のセル内容が必ず現れる。時刻 t_1 のテープの内容 $C(t_1)$ と時刻 t_2 のテープ内容 $C(t_2)$ が同じとすると、 $t_1 < t_2$ ならば、 t_2 以降は t_1 から t_2 までの変化を繰り返すことになる。従って、1024 回書き換えを実行して、テープのセルがすべて 1 になることがなければ、それ以降も決して

時間発展	
1	1000000000
2	1100000001
3	1010000010
4	1011000110
5	1000101000
6	1101101101
7	1000000000
8	1100000001
9	1010000010
10	1011000110

長さ: 20

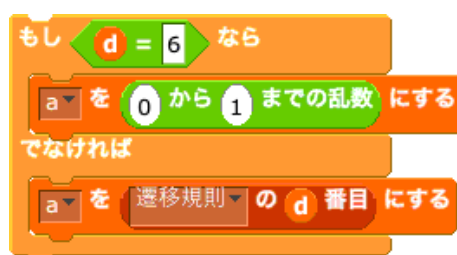
到達しないことがわかる。上の例では、6回書き換えで初期値の 1000000000 に戻っている。従って、それ以降同じことの繰り返しになるので 1000000000 から 1111111111 に遷移することがないことがわかる。

ここまでは決定的なセルオートマトンを考えてきたが、ここで非決定的なセルオートマトンを考えてみる。例えば、上の例では、 $101 \rightarrow 0$ であったが、101 は 0 でも 1 でもどちらに変えても良いことにする。このような非決定的な規則は次のように記述することにする。

000	001	010	011	100	101	110	111
0	1	1	0	1	0/1	0	1

こうすると、与えられた初期値に対して可能な遷移は 101 が出現する箇所でも 2通り考えられるため、20 回遷移を行うとその可能性は爆発的に増えることになる。さてこのように定義された非決定的セルオートマトンにおいて与えられた初期設定からテープのセルがすべて 1 になることがあるかどうか調べるにはどうしたらよいか。

非決定的なプログラムを BYOB で実行することはできないため、乱数を使ってシミュレートすることにする。遷移規則の決めるスクリプトの一部を次のように変更する。



これで時間発展の回数を 1000 回ほどにしてシミュレートしてみると 1111111111 になる場合があることがわかる。

問題 2.3 実際に BYOB のスクリプトを動かしてこのことを確かめよ。

では、 $110 \rightarrow 0$ を同じように変える場合はどうなるか。こちらも同じようにシミュレートしてもなかなか 1111111111 にはならない。

問題 2.4 実際に BYOB のスクリプトを動かしてこのことを確かめよ。

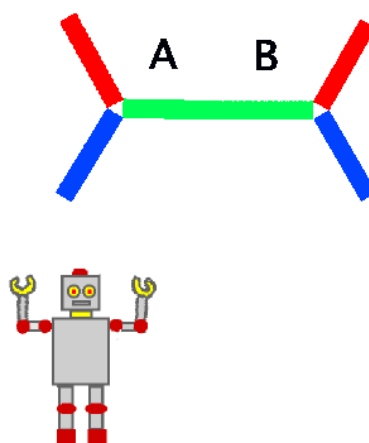
上の例のような場合テストだけでは、本当に 1111111111 に到達しないのかどうかはわからない。このようなときにモデル検査ツールを利用すると全数探索（もちろん資源に限りがあるのでいつでもできるわけではないが）を行って本当に 1111111111 に到達する可能性はないのかどうかを調べることができる。

3 かんたんなモデル検査

3.1 状態遷移図による検査

ここでは簡単なシステムに対しシステムが与えられた性質をみたすかどうか、状態遷移図を実際を書いてみてしらべることにする。

例 3.1 次の様なシステムを考える。A と B はそれぞれ緑，赤，青の羽根を持ち，互いに連携しながら回転し，A と B が接触する色によりロボットが動いたり停止する。



システムの動作条件

1. A は最初，B と緑色の羽根で接触する位置にいて，順に右回りに 120 度ずつ回転する。ただし，B の赤の羽根と接触する場合，次の瞬間に回転しないこともある。
2. B は最初，A と緑色の羽根で接触する位置にいて，順に左回りに 120 度ずつ回転する。ただし，A と B が同じ色の羽根で接触するときは，次の瞬間には回転しない。
3. ロボットは最初停止していて，A と B がともに青で接触すると動き始め，A と B がともに赤で接触する時に停止する。それ以外は動作は変化しない。

このシステムを BYOB で実際に作成すると，例えば次のようなスクリプトになる。変数 stop はブール変数³としている。



ロボットのスクリプト

³値として真偽値「はい」「いいえ」をもつ変数をブール変数とよぶ。BYOB では変数に区別はないので単に「はい」「いいえ」を代入するだけである。

A の動作条件の非決定的な動きは記述できないので、A のスクリプトでは乱数を用いている。



A のスクリプト



B のスクリプト

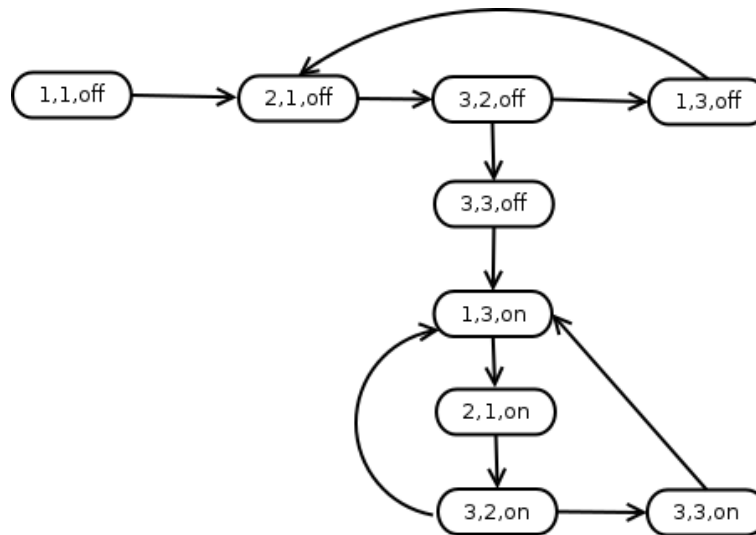
問題 3.2 上のスクリプトで「1 回繰り返す」のブロックを外すと、意図した動作をしない。理由を考えよ。

このシステムの状態遷移図を書き、次の性質が成り立つかどうか確認する。

検査項目

- ・ロボットが停止していても、いつか動き出す。
- ・一旦ロボットが動き出すと、ずっと動いたままである。

このシステムの状態を、(接触地点の A の羽根の色, 接触地点の B の羽根の色, ロボットの on/off) の 3 つ組で表すことにすると、状態遷移図は次のように表される。ただし、緑, 赤, 青をそれぞれ 1,2,3 で表している



従って、 $(1,1,\text{off}) \rightarrow (2,1,\text{off}) \rightarrow (3,2,\text{off}) \rightarrow (1,3,\text{off}) \rightarrow (2,1,\text{off}) \rightarrow \dots$ と繰り返すとロボットは停止したままであるので、最初の検査項目の「ロボットが停止していても、いつか動き出す」は間違いであることがわかる⁴。一方、一度ロボットが動き出すと、その後はロボットは on のままであるため、検査項目「一旦ロボットが動き出すと、ずっと動いたままである」は正しいことになる。

問題 3.3 例 3.1 で A が右回転ではなく左回転をするとき、検査項目の真偽はどうなるか答えよ。

問題 3.4 例 3.1 で、動作条件を次のように変更するとき検査項目の真偽はどうなるか答えよ。

1. A は最初、B と緑色の羽根で接触する位置にいて、順に右回りに 120 度ずつ回転する。ただし、B の赤の羽根と接触する場合、次の瞬間に回転しないこともある。
2. B は最初、A と緑色の羽根で接触する位置にいて、順に左回りに 120 度ずつ回転する。ただし、A と B が青色の羽根で接触するときは、次の瞬間に逆回転することもある。
3. ロボットは最初停止していて、A と B がともに青で接触すると動き始め、A と B がともに赤で接触する時に停止する。それ以外は動作は変化しない。

3.2 時相論理について

システムの状態は時間とともに変化するため、システムに関わる問題を考察する場合時間に関わる推論が必要となる。モデル検査では時間とともに変化するシステムが望ましい

⁴最初の検査項目の反例は確率的なプログラムで動かし続けると非常に小さな確率になるので、上であげた BYOB のスクリプトを実行するとロボットは常に動き出す。BYOB のスクリプトで乱数を取っている部分を外部のセンサー値、例えば光センサーの値でどちらかをとることにすると光センサーの置かれた場所によってはいつまでたってもロボットが動き出さない可能性がある。

性質をもつかどうか検査する。この時間とともに変化する性質を記述するために時相論理式を用いる。時相論理では、 $\wedge, \vee, \rightarrow, \neg$ などの論理結合子とともに、 $\square$ (いつでも)、 $\Diamond$ (いつか)、 $\circ$ (次に) などの様相結合子を用いて論理式⁵を記述する。

例えば、「ロボットが停止していても、いつか動き出す」は、

$$\square(\text{robot} == \text{off} \rightarrow \Diamond(\text{robot} == \text{on}))$$

と表すことができ、「一旦ロボットが動き出すと、ずっと動いたままである」は、

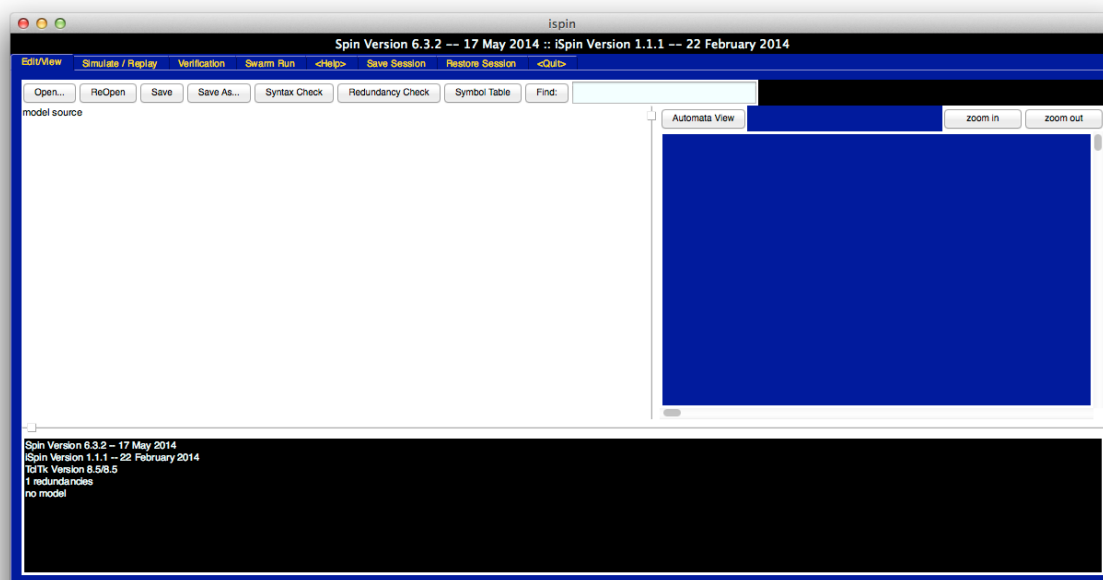
$$\square(\text{robot} == \text{on} \rightarrow \square(\text{robot} == \text{on}))$$

と表すことができる。

4 SPIN のかんたんな使い方

SPIN(Simple Promela INterpreter) のインストールについては、http://herb.h.kobe-u.ac.jp/spin_install_document.html を参考にせよ。ここでは ispin を用いて説明する。

ispin を実行すると次のウィンドウが出る。



左側の model source の部分に Promela(Process Meta Language) (C 言語に近い) でプロセスの動作を記述する。外部エディターでソースプログラムを書いて、ispin で読み込むこともできる。このテキストだけで Promela の説明をするのは困難であるので、さらに SPIN や Promela について知りたい人は [2], [8] などを参照して欲しい。

⁵ この授業で扱う時相論理は LTL とよばれる線形時相論理であるが、この授業の範囲では様相結合子の意味だけわかれば十分であるので詳しくは述べない。詳しくは参考文献を参照して欲しい。

4.1 プロセスの記述

4.1.1 変数

データ型は, `int`, `bool`, `bit`, `byte`, `short` が使える. 変数の宣言はCと似ている. 例えば, `int` 型の変数 `x,y` を宣言するときは

```
int x,y;
```

と宣言する. 変数はできるだけ宣言と同時に初期化する. これは記述したシステムの状態数をできるだけ減らすためである. 初期化する場合,

```
int x=1,y=0;
```

とする. (初期化しなければ, `spin` は自動的に初期値として0または`false`を入れるため不要な状態が増える.)

次にCにはないものであるが, 特定の文字列を値にもつ変数を宣言するために `mtype` がある. 例えば, `left` と `right` を値にとる変数 `move` を作りたい場合は,

```
mtype ={left,right};  
mtype move=left;
```

とする.

4.1.2 プロセスの記述その1

並行システムでは複数のプロセスが並行的に動作するが, それぞれのプロセスを表現するために Promela では次のように1つのプロセスを表現する. なお, `active` をつけることで, プログラムの実行と同時にプロセスが動き始める. (`active` を付けずに, 他のプロセスから明示的に動作を開始させる方法もある.)

```
active proctype P(){  
  
/* ここにプロセスを記述 */  
  
}
```

Cと同様に, `/*と*/`で囲まれた部分はコメントになる.

4.1.3 do文

並行システム等で停止せずに動き続けることが求められる場合は, プロセスの記述もそのように行う. 繰り返しを表現するのはおもに `do` 文である. `do` 文は, `do` と `od` にはさんで記述し, `break` 文で強制的に停止しない限り動き続ける. `do` 文は次のように記述する. `else` はあってもなくても構わない. 条件1から条件nのいずれも成り立たない場合にのみ実行される.

```

do
:: 条件 1->文; 文;...; 文
:: 条件 2->文; 文;...; 文
...
:: 条件 n->文; 文;...; 文
:: else ->文; 文;...; 文
od

```

条件は排他的である必要はなく、同時に複数の条件を満たす場合があってもよい。その場合は非決定的にどれかのプロセスが実行される。また、条件はなくても構わない。->は条件をわかりやすく記述するためのもので実際はセミコロン;と同じ働きである。;はPromelaでは区切りの文字なので、最後は書かなくて構わない。

例として次のプロセスを考える。

```

int x=0;

active proctype P(){
do
:: x=x+1
od
}

```

このプロセスは、変数 x の値を増加させるだけある。状態遷移図で表すと次の様なプロセスを表す。



もう少し複雑な例を作ると、

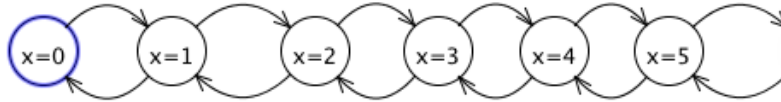
```

int x=0;

active proctype P(){
do
:: true->x=x+1
:: x>0->x=x-1
od
}

```

true 文は常に真である文である。これを状態遷移図で表すと次のようになる。



4.1.4 if文

if 文も do 文に似た構文である.

```

if
:: 条件 1->文; 文;...; 文
:: 条件 2->文; 文;...; 文
...
:: 条件 n->文; 文;...; 文
:: else ->文; 文;...; 文
fi
  
```

条件が成り立つときに, その後の文が実行される.

4.2 性質のチェック方法

Promela で記述したシステムに対して, 与えられた性質をもつかどうかチェックをすることができる. 最初は一番かんたんな assert 文を利用したチェック方法について説明し, その後 LTL 論理式で表現された性質のチェック方法について説明する.

4.2.1 assert 文

assertion 文は, 次のように書く.

```
assert{ブール式}
```

ブール式を評価して false になると assertion error を報告する. その時点で成り立って欲しい性質を記述することで, 確かに問題が起きないことの検査ができる.

例えば, 次のような Promela プログラムを作成して ispin で読み込む.

```

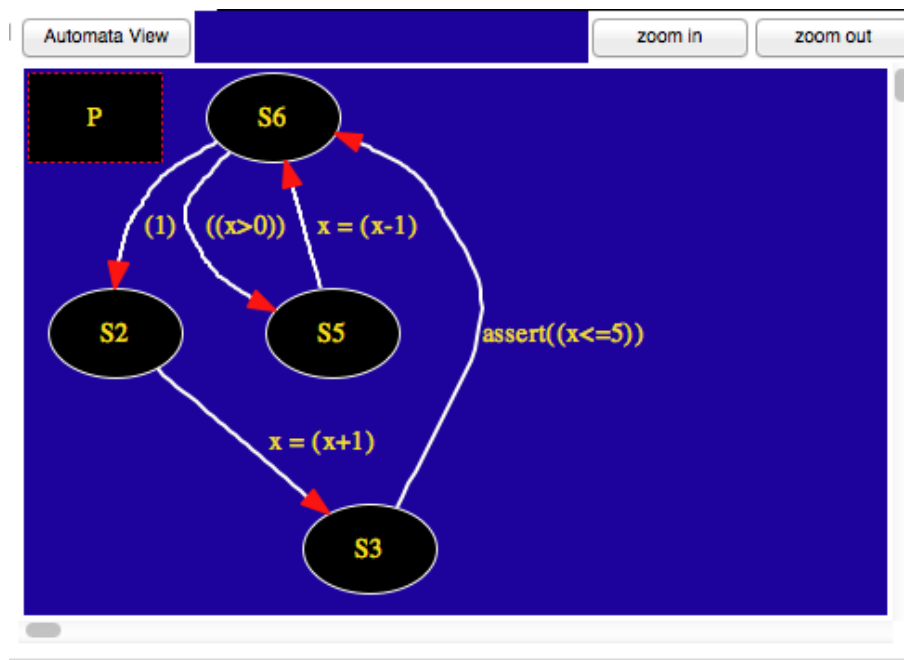
int x=0;

active proctype P(){
do
:: true->x=x+1;assert{x <=5}
:: x>0->x=x-1
od;
}
  
```

ispin の Edit/View タブの Open をクリックして、保存したファイルを開く。最初に、Syntax Check を実行する。ウインドウ下部の黒い部分に、spin: nothing to report と出れば文法エラーはない。(ファイルを編集したときは必ずこのチェックを忘れないようにする。)

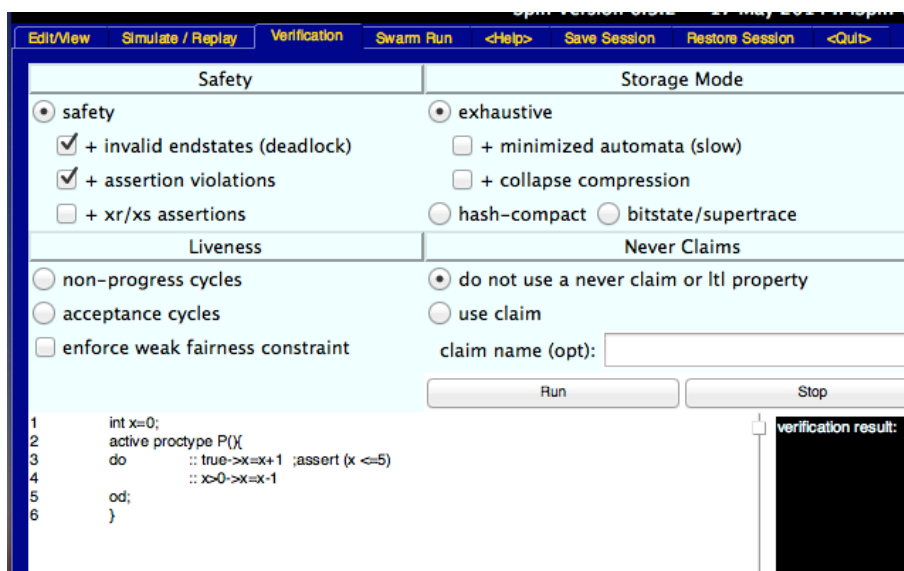
```
Spin Version 6.3.2 – 17 May 2014
iSpin Version 1.1.1 – 22 February 2014
Tcl/Tk Version 8.6/8.6
1 /Users/makoto/Desktop/ispin/test.pml:1
2 syntax check
spin: nothing to report
```

Dot⁶がインストールされている場合は、同じウインドウで Automata view をクリックすると状態遷移図が作成され確認できる。



次に、Verification タブをクリックする。デフォルトでは、Safety にチェックがあり、invalid endstates(dead lock) と assertion violation にチェックがあるが、もし、そうになっていなければ Safety にチェックを入れて、invalid endstates(dead lock) と assertion violation にチェックを入れる。

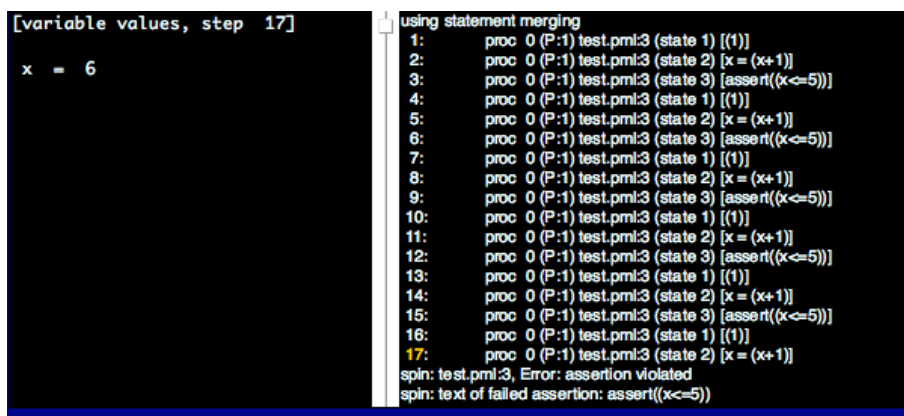
⁶<http://www.graphviz.org> より Graphviz をインストールすると Dot が自動的にインストールされる。



確認できたら下の Run ボタンをクリックすると、検証結果がでる。今回は、

pan:1: assertion violated (x<=5) (at depth 17)

となり、assertion が成り立たない場合があることがわかる。どのような場合に assertion が成り立たないか調べるには、Simulate/Replay タブ内の Run を、Guided にチェックが付いているのを確認してクリックする。



Rewind をクリックすると、実行ステップが先頭にもどる。Step Forward をクリックしていくとエラーが出てくるまでのステップを表示する。spin は判断した条件なども 1 ステップで表示するので、プロセスの経過が読みにくいですが、今回は variable values ウィンドウを見ると、 $x=x+1$ を実行し続けることで x の値が 6 になり、assertion が成り立たなくなったことがわかる。

4.2.2 LTL 論理式

上の assert 文と同じプログラム (assert 文は除く) に対して、LTL 論理式で検査を行うことにする。検査論理式は $\Box(x \leq 5)$ 、すなわち、いつでも $x \leq 5$ がなりたつこととする。これを検査するには、プログラムに以下の文を追記する。

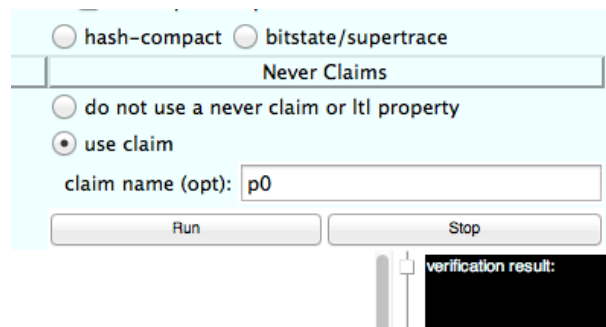
```
ltl p0 {[] (x<=5)}
```

LTL 論理式を記述する文法は

```
ltl [名前] {LTL 論理式}
```

である。名前は LTL 論理式を区別するためにつけるがなくても良い。付けない場合は自動的に `ltl_0` のように spin が自動的に付ける。なお、`ltl` 文はプロセス記述の外側に書かなくてはならない。Promela では、様相結合子として、`[]`, `◇` の他に `U`(Until), `W`(weak Until), `V`(dual of `U`) を使うことができる。詳しくは Promela の Man-Pages and Semantics Definition[9] を参照してほしい。

LTL 論理式を検査するには、Verification タブをクリックして、Never claims のところの `use claim` にチェックを入れて、検証する式の名前、例えば `p0` を入力する。検証式が一つしか書いていない場合や先頭の検証式を検証する場合は空欄でも構わない。



Run をクリックすると検証が始まる。検証式に反する実行経路がある場合、`errors:1` のようにエラーの存在を表示する。今の場合は `assertion` 文でチェックしたように、反例があるのでエラーがあることを表示して停止する。

```
spin -a test.pml
ltl p0: [] ((x<=5))
gcc -DMEMLIM=1024 -O2 -DXSAFE -w -o pan pan.o
./pan -m10000 -a -N p0
Pid: 21349
warning: only one claim defined, -N ignored
pan:1: assertion violated !((x<=5)) (at depth 24)
pan: wrote test.pml.trail

(Spin Version 6.3.2 -- 17 May 2014)
Warning: Search not completed
+ Partial Order Reduction

Full statespace search for:
  never claim      + (p0)
  assertion violations + (if within scope of claim)
  acceptance cycles + (fairness disabled)
  invalid end states - (disabled by never claim)

State-vector 28 byte, depth reached 24, errors: 1
  13 states, stored
  0 states, matched
  13 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
  0.001 equivalent memory usage for states (stored*(State-vector + overhead))
  0.291 actual memory usage for states
 128.000 memory used for hash table (-w24)
  0.534 memory used for DFS stack (-m10000)
 128.730 total actual memory usage

pan: elapsed time 0.01 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

反例に至る実行経路は `assert` 文のときと同様に確認できる。

例 4.1 例 3.1 では、システムが同期して変化するがこのようなシステムの場合は、変数の更新を同時にする必要がある。例えば、例 3.1 は次のように Promela で記述できる。

```

mtype = {green, red, blue, on, off};
mtype a=green, b=green;
mtype robot=off;
mtype Na, Nb, Nrobot;

active proctype robo(){
    do
        ::true ->
            if          /* 次の瞬間の A の状態の決定 */
                :: (b==red) -> Na = a
                :: (a==green) -> Na = red
                :: (a==red) -> Na = blue
                :: (a==blue) -> Na = green
            fi;

            if          /* 次の瞬間の B の状態の決定 */
                :: (a==b) -> Nb = b
                :: else ->
                    if
                        :: (b==green) -> Nb = red
                        :: (b==red) -> Nb = blue
                        :: (b==blue) -> Nb = green
                    fi
                fi;

            if          /* 次の瞬間の robot の動作決定 */
                :: (a==blue && b==blue) -> Nrobot = on
                :: (a==red && b==red) -> Nrobot = off
                :: else -> Nrobot = robot
            fi;
        /* 値の更新*/
        a = Na;
        b = Nb;
        robot = Nrobot
    od
}

```

問題 4.2 上の Promela プログラムに対し、

$$\Box(\text{robot}=\text{off} \rightarrow \Diamond(\text{robot}=\text{on}))$$

及び

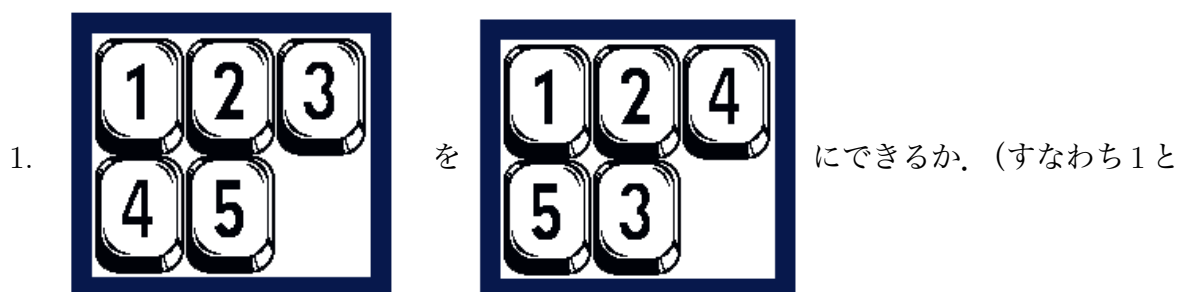
$$\Box(\text{robot}==\text{on} \rightarrow \Box(\text{robot}==\text{on}))$$

を Spin で検査せよ.

問題 4.3 6つのマス目のなかに5枚の数字が入ったコマがあり、一つの空きスペースを利用して、コマを動かすことができる.

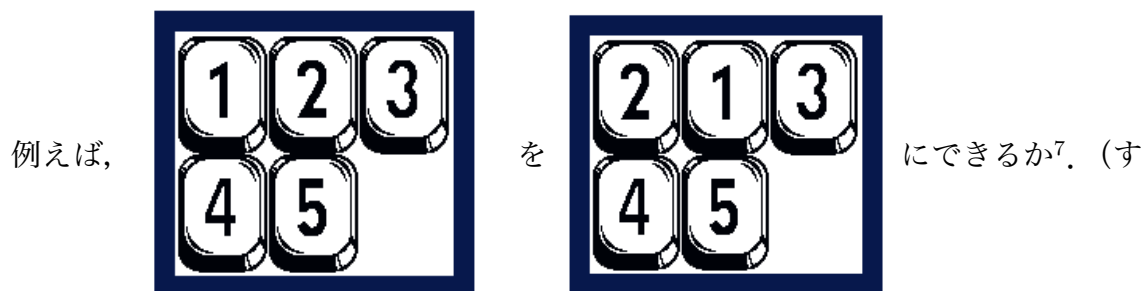


このとき、何回かコマを動かして指定された形にできるか Spin を利用して調べよ.



2 は固定して, 3,4,5 を右回りに回せるか?) できる場合はその手順を示せ.

2. 任意の2つのコマを入れ替えることはできるか.



なわち 3 と 4 と 5 は固定して, 1 と 2 を交換できるか?)

問題 4.4 次の非決定的な有限セルオートマトンに対し、テープ 10000000000 をかけて動かす. ただし、境界条件は周期境界条件とする. このとき、1111111111 となる可能性があるかどうか Spin を用いて調べよ.

⁷これができると、他の場合もできることが少し考えるとわかる.

000	001	010	011	100	101	110	111
0	1	1	0	1	0	0/1	1

ヒント：同じ様な記述が続くので inline 記法を用いて、次のようにマクロ定義を利用するとよい。

```
inline trans(x,y,z,u){
if
:: (x==0&&y==0&&z==0) || (x==0&&y==1&&z==1) ||
    (x==1&&y==0&&z==1) || (x==1&&y==1&&z==0) -> u=0
:: (x==0&&y==0&&z==1) || (x==0&&y==1&&z==0) ||
    (x==1&&y==0&&z==0) || (x==1&&y==1&&z==0) || (x==1&&y==1&&z==1) -> u=1
fi;
}
```

5 複数プロセス

5.1 複数プロセスの記述

ここまでの Promela プログラムの例は、例 3.1 の Promela 記述も含め、すべて単一のプロセス内で状態の分岐がおきるものであったが、一般的なりアクティブシステムでは複数のプロセスが相互作用しながら動作を続けて行く。従って、今後は複数のプロセスが同時に動作しているシステムの記述とそこでおきる問題について考察していく。

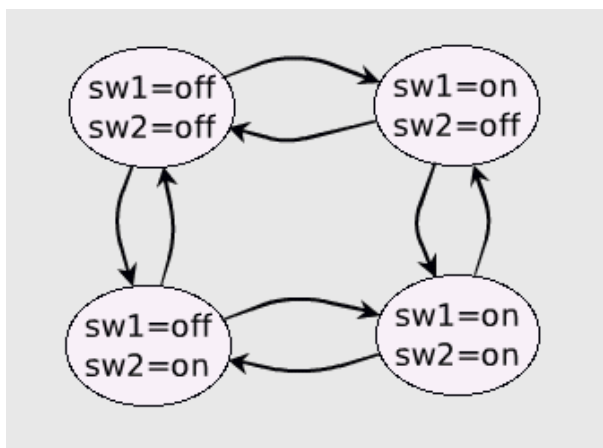
例 5.1 ふたつのスイッチ sw1 と sw2 があり、単にそれらが独立に on と off を繰り返すシステムは次のように記述できる。

```
mtype = {on, off};
mtype sw1=off, sw2=off;

active proctype P1(){
do
:: sw1==off-> sw1=on
:: sw1==on -> sw1=off
od
}

active proctype P2(){
do
:: sw2==off-> sw2=on
:: sw2==on -> sw2=off
od
}
```

Spin は複数のプロセスに対して、合成したプロセスを作成して検証する。例えば、この例では、spin 内部では次のような状態遷移図をもつプロセスが構成される。



この例は二つのプロセスが独立に動いているので、互いに相手に影響を及ぼすことはないが、次は同時に動作することで影響するプロセスの例である。

例 5.2 int x;

```

proctype Proc1() {
    x = 1;
    x = x + x;
    assert(x == 2);
}

```

```

proctype Proc2() {
    x = 0;
}

```

```

init {
    run Proc1();
    run Proc2();
}

```

ここでは、プロセス定義には active を付けない代わりに、init プロセスを定義して、Proc1 と Proc2 を順に走らせている。一見すると、Proc1 が動作を終えてから、Proc2 を動作させているように思えるが、プロセスは独自の実行速度をもち、また文の実行も Proc1 と Proc2 がインターリーブ（間に互いに挟まって実行が進む）するため、assert 文がなりたたない場合があることを spin は報告する。

同時に動くことを想定している場合は、このように意図せぬ動作がおきることには注意をする必要がある。上の例では、 $x=1; x=x+x$ の間に他のプロセスにより、 x の値が書き変わることを防ぐ必要がある。Promela ではそのような対処をするための仕組みとして、atomic がある。一連のプログラムを atomic でくくると、その内部のプログラムを実行中は他の

プロセスによるインターリーブを許さないことができる。これは BYOB で関数ブロックを定義してアトミックの指定をするのと同じである。

問題 5.3 次のシステムを検証せよ。

```
int x;

proctype Proc1() {
  atomic{x = 1;
  x = x + x;
  assert(x == 2)
}
}

proctype Proc2() {
  x = 0;
}

init {
  run Proc1();
  run Proc2();
}
```

5.2 排他制御

上の例のように、何かの変数を変えたりするときに他のプロセスに邪魔をされたくない場所のことを危険部分 (Critical Section) とよび、その危険部分に入るプロセスを制御する仕組みを相互排除とよぶ。相互排除を実現するための仕組みはいろいろあるが、デッドロック (互いに動作を牽制し合い誰も動けなくなる) がおきないこと、応答性 (入ろうとしたら必ずいつか入れる)などを保障することが重要である。次にあげるのは相互排除のためのアルゴリズムの一つであるピーターソンのアルゴリズムである。なお、この Promela 記述で、`try` とか `cs` とあるのはラベルで、プログラムそのものには影響はない。検証する時にプログラムの位置を示すのに利用する。

```
bool wantP=false,wantQ=false;
byte last=1;

active proctype P(){

do
:: wantP=true;
  last=1;
```

```

    try: (wantQ==false)|| (last==2);
    cs:   wantP=false
  od
}

active proctype Q(){

do
:: wantQ=true;
   last=2;
  try: (wantP==false)|| (last==1);
  cs:   wantQ=false
  od
}

```

二つのプロセス P, Q はそれぞれ危険部分 (cs のラベルがある部分) に入りたい場合は、それぞれ wantP と wantQ を true にする。次に last 変数を P は 1 に Q は 2 にする。これは危険部分を直前に自分が使っていて相手が待っている可能性があるので、その場合に相手に先に入らせるための仕組みである。ラベル try の場所では、相手が危険部分に入ること表明していないか、相手が自分に先に入れるよう last 変数が自分用に代わっているかを判断して、どちらかが真のときは危険部分に入り、自分の要求を偽にして、もとにもどる。(本来であれば、危険分で何らかの仕事をしたり、危険部分に入ろうとする以前に何らかの仕事をしているはずであるが、その部分はすべて隠して本質的な部分のみをこのアルゴリズムは表現している。上で、ラベルは検証する時にプログラムの位置を示すのに利用すると書いたが、例えば、P@cs という述語は、プロセス P がラベル cs の位置にいれば真で、他の位置にいれば偽になる。従って、 $\Diamond(P@cs \ \&\& \ Q@cs)$ は、実行途中で、P も Q も cs の位置にいたことがあれば真になるので、この検証式の否定 $\neg \Diamond(P@cs \ \&\& \ Q@cs)$ が正しいかどうかを検証して、もし反例があがればピーターソンのアルゴリズムは正しくないことになる。

問題 5.4 ピーターソンのアルゴリズムが正しいことを検証せよ。

問題 5.5 BYOB でピーターソンのアルゴリズムを実装してその動きを確認せよ。

5.3 具体的なプログラムの検証

最後に 1.3 であげたプログラムに対して、Spin を用いて検証を行う。

5.3.1 BYOB スクリプトの解説

検証を行う前にプログラムの詳細を解説する。プログラムで使用するスプライトは、リモコン、ロボット、スタートボタン、ストップボタンでそれぞれに対して次のようにスクリプトを作成する。

まず最初に（デッドロックに陥った時に初期設定を容易にするために）緑旗をクリックしたら変数やリストを初期化してリモコンとロボットの動作体制をとるため次のようにスクリプトを作成する。



ここで initialize ブロックはつぎのように作成する。



変数の toControl_available と toRobot_available はそれぞれリスト toControl, toRobot が使用できるかどうかを表すためのブール変数である。リストには高々1つのデータしか登録できないという仕様のために、ここが危険領域になる。そこでピーターソンのアルゴリズムと同様にこのような変数を用意して管理する。また、変数の active はリモコンからロボットに Start を送信したことを表すためのブール変数で、変数 action はロボットが動いていることを表すためのブール変数である。

スタートボタンとストップボタンのスクリプトはつぎのようにする。



ここで send-message タイルは次のように作成する。



toControl リストが使用可能になるまで待つ、メッセージ (StartButton/StopButton) を送る。メッセージが相手に読み込まれるまでの間、toControl に他のメッセージが送られてくることを阻止するために available 変数を「いいえ」にしている。

リモコンのスク립トは次のようにする。



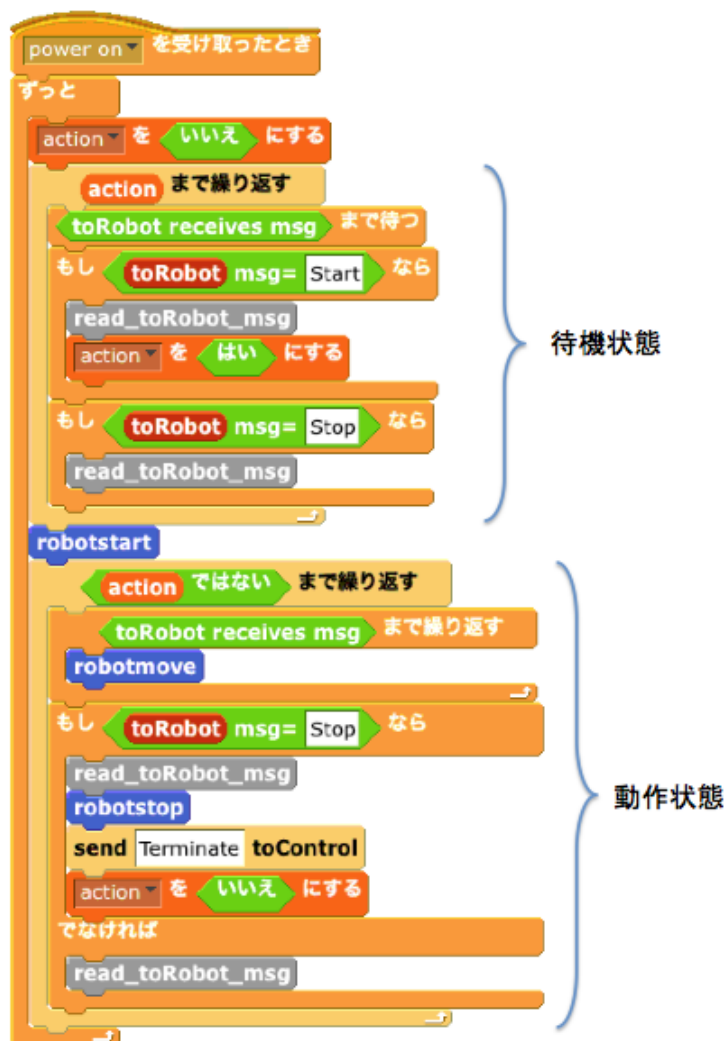
リモコンはメッセージを受信すると何らかの動作を行うようにしている。toControl receive msg ブロックはつぎのような述語型のブロックとして作成しているため、「toControl receive msg を待つ」は「toControl_available を待つ」と同じことである。メッセージのやり取りを意識してこのようにしている。

待機状態（ロボットに動作指令を出していない状態）のときは、toControl にメッセージが送られてくるのを待ち、メッセージが StartButton ならば、ロボット（正確には toRobot リスト）に Start を送る。待機状態で Stopbutton を読み込むとき無視することになっている。Start を送信後は待



機状態から動作状態に移行し、次のメッセージの受信を待つ。動作状態で toControl から StopButton を読み込むと、toRobot に Stop を送る。動作状態で StartButton を読み込むときは無視をする。toControl から Terminate（これはロボットは動作を停止したことをリモコンに伝えるために送るメッセージ）を読み込んだら待機状態に戻る。

ロボットのスクリプトは次のようにする。



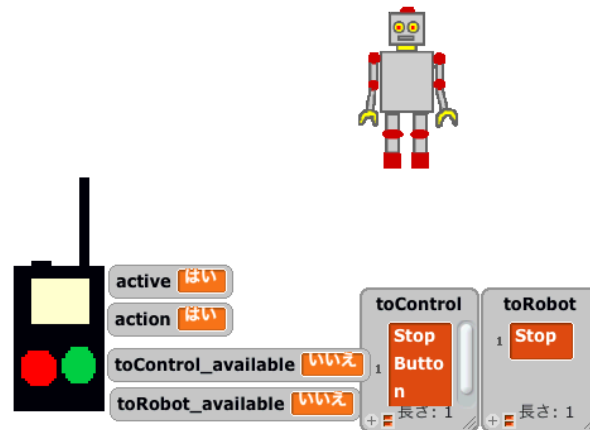
ロボットは、待機状態ではリモコンから toRobot へのメッセージ受信を待ち、toRobot に Start が送られて来たら動作状態に移行する。待機状態で Stop が送られて来た場合は無視する。動作状態に移行後は、リモコンから toRobot に Stop が送られてくるまで動き続ける。動作状態で Start が送られて来た場合は無視する。Stop が送られて来たら動作を停止し、リモコン（正確には toControl に）Terminate を送信して待機状態に戻る。

以上がプログラムの全体である。

このプログラムを緑旗のクリックから開始し、スタートボタン／ストップボタンをクリックしてみても特に問題はないように思えるが、次のようなスクリプトを作成して、ロボットの動作状態で、左矢印キーを押し続けてみる。



すると次のように toControl と toRobot にメッセージが残ったままデッドロックに落ちている。



5.3.2 Spin による解析

上で作成したBYOBプログラムをSpinを用いて解析してみる。このプログラムをPromelaで表現すると例えば次のようにできる。

```
mttype = {StopButton, Terminate, StartButton, Start, Stop};
mttype msg;
chan toControl =[1] of {mttype};
chan toRobot = [1] of {mttype};
```

```
active proctype PushStart ()
{
end:
do
:: toControl!StartButton
od
}
```

```
active proctype PushStop ()
{
end:
do
:: toControl!StopButton
```

```

od
}

active proctype Controller()
{
Inactive: end:
do
::atomic{ toControl?msg ->
if
:: (msg==StartButton) -> toRobot!Start ; goto Active
:: (msg==StopButton)
fi
}
od;

Active:
do
::atomic{ toControl?msg ->
if
:: (msg==StopButton) -> toRobot!Stop
:: (msg==Terminate) -> goto Inactive
:: (msg==StartButton)
fi
}
od
}

active proctype Robot()
{
Inactive: end:
do
:: atomic{toRobot?Start -> goto Action}
:: toRobot?Stop
od;

Action:
do
:: atomic {toRobot?Stop -> toControl!Terminate; goto Inactive}
od
}

```

これを Spin にかけて検証を行うと,

```
spin -a robot.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 68772
pan:1: invalid end state (at depth 44)
pan: wrote robot.pml.trail
```

(Spin Version 6.3.2 -- 17 May 2014)

Warning: Search not completed

+ Partial Order Reduction

Full statespace search for:

never claim - (not selected)

assertion violations +

cycle checks - (disabled by -DSAFETY)

invalid end states +

State-vector 60 byte, depth reached 45, errors: 1

46 states, stored

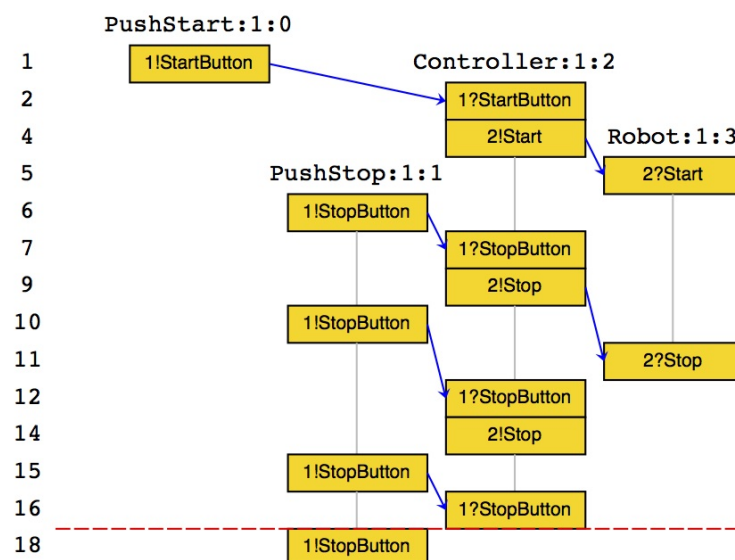
21 states, matched

67 transitions (= stored+matched)

36 atomic steps

hash conflicts: 0 (resolved)

となりエラーを報告する。エラーがおきるまでの過程を (shorttrail のオプションにチェックを入れて) Simulate で確認すると,



よって、このデッドロックを解消するにはリモコンが動作状態のときに StopButton を受け取ったら、ロボットから Terminate が送られてくるまでストップボタンが押されても無視する必要があることがわかる。例えば、次のようにリモコンのスクリプトを書き換えるとこのデッドロックは解消される。



32

参考文献

- [1] Principles of Model Checking, C. Baier and J.P. Katonen, The MIT Press
- [2] SPIN モデル検査入門, Mordechai Ben-Ari 著 中島 震 監訳, オーム社
- [3] Logic in Computer Science-Modelling and Reasoning about Systems, M. Huth and M. Ryan, 2nd edition, Cambridge University Press
- [4] コンピュータサイエンス入門 論理とプログラム意味論, 田辺誠・中島玲二・長谷川真人著, 岩波書店
- [5] SPIN モデル検査 検証モデリング技法, 中島 震著, 近代科学社
- [6] SPIN モデル検査入門, 中島震・谷津弘一・野中哲・足立太郎著, オーム社
- [7] 4日で学ぶモデル検査, 産業技術総合研究所システム検証研究センター, NTS
- [8] Spin Online References, <http://spinroot.com/spin/Man/index.html>
- [9] Promela:Man-Pages and Semantics Definition, <http://spinroot.com/spin/Man/promela.html>
- [10] ビジュアルプログラミング環境を用いたシステム検証の教育について, 山上和哉, 修士論文, 神戸大学大学院人間発達環境学研究科, 2013

平成 26 年 12 月 2 日作成

平成 27 年 6 月 12 日修正

平成 27 年 7 月 15 日修正

平成 27 年 10 月 14 日修正