本文書は、Spin Model Checker の公式サイト http://spinroot.com にある Basic Spin Manual を翻訳したものです。この文書は、Gerard J. Holzmann に許可を得て公開しています。誤訳、誤記などありましたら、https://github.com/masateruk/BasicSpinManualにてIssueまたはPRにてご報告いただけると幸いです。

訳者 Masateru Kawaguchi

Basic Spin Manual

Spin は並行システム（中でもデータ通信プロトコル）の論理的一貫性を検証するツールです。システムの振る舞いは、Promela（Process Meta Language）と呼ばれるモデリング言語によって記述します。この言語では、並行プロセスの動的生成を記述する事ができます。チャネルを介したメッセージ通信は同期通信（ランデブ）、非同期通信（キューによるバッファリング）の両方を記述する事ができます。

われわれは、*Spin* を使い Promela で記述したシステムをランダムシミュレーションしたり、また、Promela で記述したモデルから C 言語の検証プログラムを生成し、システムの正当性の検証を行う事ができます。シミューレーションや検証の間、*Spin* はデッドロックがない事や意図しないメッセージを受け取らない事、また実行していないコードがないか、などチェックします。検証プログラムはまたシステムの不変条件が保持されているかどうかを検証したり、ノンプログレスサイクル（non-progress execution cycles）がないか検索したり、線形時相論理（Linear Temporal Logic）で記述した制約に違反していないかを検証します。

検証プログラムは効率性にすぐれ、メモリ使用量も多くありません。*Spin* による全探索検証の結果、われわれは、システムの振る舞いにエラーがないかどうかについて、数学的に裏付けされた確証を得ることができるのです。コンピュータのリソースの制限から通常は不可能とされる大きなシステムの検証も、"ビット状態空間手法" という倹約技術がその検証を可能としています。この手法では、副作用を最小限に抑えながら、到達可能な各状態を数ビットに圧縮することができます。

このメモの第 1 部では Promela の紹介を、第 2 部では *Spin* の使用方法についてとりあげます。Promela の簡潔なリファレンスはこちらにあります。付録では、いくつかの例を示し、*Spin* で検証するために Promela を使った基礎的なモデルの作成方法について説明します。

このマニュアルでは、*Spin* の主要部分のみについて議論します。Promela 言語の拡張やツールの最近のバージョンに関する話題については触れていません（詳しくは、このマニュアルの最後を参照してください）。また、LTL で記述した制約を使用した検証についても議論しません。

モデリング言語（Modeling Language）

Promela は検証用のモデリング言語です。この言語では、検証する上で関心のないプロセス間相互作用を捨象し抽象化したプロトコル（または、一般的な分散システム）を記述することができます。*Spin* の使用目的は、いくつかの理由により疑わしいと思われる少数のプロセスの振る舞いを検証することです。 Promela で関心のある振る舞い部分について記述し、検証します。Promela モデルを段階的に詳細化していき、一連のステップを経て検証を完結させます。各々のモデルに対し、環境によって異なる様々な仮定（たとえば、メッセージをロスするとか、メッセージが重複するとか）を設定して、検証することができます。一度、*Spin* により正しいモデルを構築することができれば、その後のモデルの作成と検証は、その事実をもとに進めることができます。

Promela モデルは *プロセス*、*メッセージチャネル*、*変数*からなります。プロセスはグローバルオブジェクトです。メッセージチャネルと変数はグローバル、もしくは、プロセス内ローカルに宣言して利用することができます。プロセスは振る舞いを記述し、チャネルとグローバル変数はプロセスが走る環境を定義します。

実行可能性（Executability）

Promela には、条件と文の間になにも違いはありません。単独のブール条件は単に文として扱われます。各文の実行は、その*実行可能性*によります。文は実行可能であるか、実行不可能でブロックされるかのどちらになります。実行可能性は、同期をとる基本的な方法です。ブロックされたプロセスは、実行可能になるイベントが発生するのを待ちます。たとえば、以下のようにビジーループで書く代わりに

```
while (a != b)
        skip    /* wait for a==b */
```

Promela では下のように書いて、同じ効果を得ることができます。

```
(a == b)
```

条件は、その条件が満たされるときのみ実行可能になります。もし、条件が満たされないときは、その条件が満たされるまでブロックされます。

グローバル変数はシステムに関するグローバルな情報を保持し、プロセス内のローカル変数は各プロセスの状態を保持します。変数の宣言は、

```
bool flag;
int state;
byte msg;
```

のように記述します。この例のように Promela は値域の異なる3つの整数型をサポートしています。変数のスコープは、プロセスの宣言の外側で宣言されたものはすべてグローバルになり、プロセス内で宣言したものはプロセス内ローカルになります。

データ型（Data Types）

下記の表は、32-bit ワードサイズのコンピュータにおける基本データ型、そのサイズ、範囲をまとめたものです（データ型参照）。

Table 1 - Data Types

Typename	C-equivalent	Macro in limits.h	Typical Range
bit or bool	-		0..1
byte	bit-field	CHAR_BIT (width in bits)	0..255
short	uchar	SHRT_MIN..SHRT_MAX	-2^15 - 1 .. 2^15 - 1
int	short int	INT_MIN..INT_MAX	-2^31 - 1 .. 2^31 - 1

bit と bool は 1 ビットの情報を表す同義語です。byte は 0 から 255 までの値をとる符号なしの整数です。shot と int はそれらがとりうる値の範囲が違うだけで、ともに符号付き整数です。

mtype は、mtype = { ... } という形で宣言される列挙型です。これについては、後述します。

配列（Array Variables）

変数は、配列としても宣言できます。たとえば、

```
byte state[N]
```

と宣言した場合、state は N 個の byte 型の変数をもつことができる配列です。配列の各要素への代入および値の参照は、

```
state[0] = state[3] + 5 * state[3*2/n]
```

のように記述し、インデックスには変数や定数を使用する事ができます。配列のインデックスはユニークな整数値を表す式を使用することができます。配列の範囲（0 .. N-1）外のインデッ

クスを指定した場合、その値は未定義です。たいていはランタイムエラーとなります（多次元配列も typedef を使用する事により間接的に宣言できます。WhatsNew.html 2.1.7 節参照）。

変数宣言と 2 つのタイプの文、ブール条件文と代入文の例を見てきました。宣言と代入は常に*実行可能*です。また、条件文はそれが満たされているときのみ実行可能です。

プロセス型（Process Types）

変数やチャネルの状態の参照や変更ができるのは、プロセスのみです。プロセスの振る舞いは proctype 宣言により定義します。たとえば、1 つのローカル変数 state をもつプロセスの宣言の例を下記に示します。

```
proctype A()
{       byte state;

        state = 3
}
```

この場合、プロセス型の名前は A となります。宣言の本体は中括弧 {} で囲みます。宣言の本体は、0個以上のローカル変数の宣言、および、文のリストからなります。上記の宣言は、1 つのローカル変数宣言と変数 state に値 3 を代入するひとつの文からなります。

文は、セミコロンにより分けられます（セミコロンは文の終了記号ではありません。そのため、最後の文のうしろのセミコロンは意味を持ちません）。Promela は 2 つの異なる文の*分離記号*（矢印記号 '->' とセミコロン ';'）を持ちます。2 つの違いはありません。矢印記号はときどき 2 つの文の因果関係を表すのに使われます。たとえば、以下の例のように使われます。

```
byte state = 2;

proctype A()
{       (state == 1) -> state = 3
}

proctype B()
{       state = state - 1
}
```

この例では、2 つのプロセス型 A と B を宣言しています。変数 state はグローバル変数として宣言され、値 2 により初期化されます。プロセス型 A は矢印記号により分けられた 2 つの文からなります。この例では、プロセス型 B は変数 state の値を 1 減らすひとつの文からなります。代入文は常に実行可能なので、B 型のプロセスはブロックされる事なく常にその動作を完了させる事ができます。一方 A 型のプロセスは変数 state の値が適切な値をとるまで条件文のところでブロックされます。

プロセスのインスタンス生成（Process Instantiation）

proctype による定義はプロセスの振る舞いを宣言するのみで、実行はしません。Promela モデルでははじめ、タイプ init のひとつのプロセスのみが実行されます。このプロセス型はすべての Promela モデルの中で明示的に宣言する必要があります*1。Promela で定義できる最も小さいモデルは、以下のものです。

```
init { skip }
```

skip は何もしないダミーの文です。しかし興味深い事に、この初期プロセスはグローバル変数を初期化し、プロセスのインスタンスを生成することができます。上記のシステム用の init プロセスの宣言は以下のようになります。

```
init
{       run A(); run B()
}
```

run はプロセスの名前（例えば、A）を引数にとる単項演算子です。run は、そのプロセス型が定義されていてインスタンスが生成できるときのみ実行可能です。たとえばすでにたくさんのプロセスが走っていて新たなインスタンスを生成できないときは、実行不可能となります。

run は、生成するプロセスに対し基本データ型のパラメータを渡す事ができます。たとえば、以下のように宣言を記述する事ができます。

```
proctype A(byte state; short foo)
{
        (state == 1) -> state = foo
}

init
{
        run A(1, 3)
}
```

データの配列やプロセス型をパラメータとして渡す事はできません。後で見るように、基本データ型以外にパラメータとして渡せるのは、唯一メッセージチャネルのみです。

新たなプロセスを生成する run は init プロセスだけでなくすべてのプロセスで使用可能です。プロセスは、run により生成されます。プロセスは、終了したときに（すなわち、プロセス型の宣言の本体部分の最後まで到達すると）姿を消します。プロセスは自身が生成したプロセスが終了してから終了します。

run によりプロセス型 A と B のコピーをいくつでも作る事ができます。しかしひとつ以上の並行プロセスがひとつのグローバル変数を読み書きする場合、よく知られる問題を引き起こしかねません（[2] に例があります）。たとえば、以下のような 2 つのプロセスがグローバル変数 state を共有するシステムを考えましょう。

```
byte state = 1;
```

```
proctype A()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp+1; state = tmp
}

proctype B()
{       byte tmp;
        (state==1) -> tmp = state; tmp = tmp-1; state = tmp
}

init
{       run A(); run B()
}
```

もし 2 つのうちひとつのプロセスが相手のプロセスがスタートする前に完了してしまうと、もう一方のプロセスははじめの条件文のところで永遠にブロックされてしまいます。もし 2 つのプロセスが同時に条件文を通り抜けたら、2 つとも完了する事ができるでしょう。しかしそのときの `state` の値は予測不可能です。`state` の値は 0, 1, 2 のうちのどれかになるでしょう。

この問題にはたくさんの解が考えられます。グローバル変数を使わないというような方法から、共有変数のテストとセットを不可分で行う事を保証する特別なマシン語を利用する方法まで。下記の例は最初に公開された解のひとつです。これはドイツの数学者 Dekker によるものです。このアルゴリズムにより、3 つのグローバル変数を使い、コード中の任意の *クリティカルセクション*への進入を 2 つのプロセスが排他的に行うことができます。`Promela` モデルの最初の 4 行は、C 言語スタイルのマクロ定義です。最初の 2 つのマクロは `true` を定数 1、`false` を定数 0 に定義しています。同様に、*Aturn*, *Bturn* も定数として定義しています。

```
#define true    1
#define false   0
#define Aturn   false
#define Bturn   true

bool x, y, t;

proctype A()
{       x = true;
        t = Bturn;
        (y == false || t == Aturn);
        /* critical section */
        x = false
}

proctype B()
{       y = true;
        t = Aturn;
        (x == false || t == Bturn);
        /* critical section */
        y = false
}

init
{       run A(); run B()
}
```

このアルゴリズムは、繰り返し実行する事ができ、2つのプロセスの実行スピードにはまったく依存していません。

atomic シーケンス（Atomic Sequences）

Promela にはこのテスト & セットの問題を回避する別の手段があります。それは、atomic です。キーワード atomic で囲まれる文のシーケンスは、他のプロセスによってインターリーブされず、不可分に実行されます。atomic シーケンスの中の最初の文以外のところでブロックされた場合、ランタイムエラーとなります*2。このように atomic シーケンスを使う事により、先の例に示したグローバル変数 state に対する並行アクセスに関する問題を防ぐことができます。

```
byte state = 1;

proctype A()
{       atomic {
          (state==1) -> state = state+1
        }
}

proctype B()
{       atomic {
          (state==1) -> state = state-1
        }
}

init
{       run A(); run B()
}
```

このケースでは state の最後の値は、どちらのプロセスが実行したかにより 0 または 2 になります。もう一方のプロセスは永遠にブロックされます。

atomic シーケンスは検証するモデルの複雑さを軽減する重要な道具です。atomic シーケンスは分散システムがとりうるインターリービングを制限することに注意してください。例えば、ローカル変数に対する操作を atomic シーケンスとすることによって、複雑なモデルを扱う事ができるようになります。これにより複雑さを劇的に減少させる事ができます。

メッセージ通信（Message Passing）

メッセージチャネルはひとつのプロセスから他のプロセスへのデータ転送をモデル化するのに使用されます。以下で示すように、チャネルはローカル変数でもグローバル変数でも宣言できます。

```
chan qname = [16] of { short }
```

上記は short 型のメッセージを 16 個保持する事ができるチャネルを宣言しています。チャネルの名前はチャネルを通して他のプロセスへ送ったり、プロセスのインスタンス生成時のパラメ

ータとして渡したりすることができます。もしそのチャネルでやり取りされるメッセージがひとつ以上のフィールドを持つ場合の宣言は以下のようになります。

```
chan qname = [16] of { byte, int, chan, byte }
```

このときこのチャネルは、それぞれ 2 つの 8 bitの値、1 つの 32 bitの値、チャネルの名前をもつメッセージを 16 個保持することができます。

次の文、

```
qname!expr
```

は、式 expr の値を上記の例で作ったチャネル qname へ送信します。チャネルの（キューの）末尾にその値を追加します。

```
qname?msg
```

上の文は、チャネルの先頭からメッセージを受信し、変数 msg に代入します。チャネルは first-in-first-out 順でメッセージをやりとりします。上記のケースではひとつの値がチャネルを通してやりとりされてます。もしメッセージごとに 2 つ以上の値をやりとりしたい場合は、下記のようにコンマで区切ったリストをチャネルに続き記述します。

```
qname!expr1,expr2,expr3
qname?var1,var2,var3
```

チャネルの宣言よりもメッセージの送受信のパラメータが多かったり、少なかったりした場合はエラーとなります。
便宜のため、メッセージの最初のフィールドはよくメッセージタイプを表すために使われます（つまり、定数が使われる事が多いのです）。このような場合の別の書き方として、送受信操作のパラメータを、メッセージタイプに続き括弧でくくってメッセージフィールドのリストを続けることがあります。

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

メッセージの送信操作は、チャネルがいっぱいでないとき実行可能となります。また受信操作は受信しようとしているチャネルが空でないとき実行可能となります。また、受信時のパラメータのいくつかを定数とする事もできます。

```
qname?cons1,var2,cons2
```

この場合、定数で指定したすべてのメッセージフィールドの値がチャネルの先頭のメッセージと一致する場合、受信操作は実行可能となります。繰り返しになりますが、実行不可能であったとしてもなにも悪い事はおきません。プロセスはその文が実行可能になるまでブロックされるか、他の実行可能な文を実行します。

ここにいままで紹介したいくつかの機構を使った例を示しましょう。

```
proctype A(chan q1)
{       chan q2;
        q1?q2;
        q2!123
}

proctype B(chan qforb)
{       int x;
        qforb?x;
        printf("x = %d¥n", x)
}

init {
        chan qname = [1] of { chan };
        chan qforb = [1] of { int };
        run A(qname);
        run B(qforb);
        qname!qforb
}
```

プリントされる値は、123 となります。

定義済みの関数 len(qname) はパラメータで渡した名前をもつチャネルに現在保存されているメッセージの数を返します。もし代入文の右辺ではなく単にひとつの文として使われた場合、チャネルが空の場合は実行不可能になる事に注意してください。なぜなら、空の場合は 0 を返し、定義によりそれは一時的に実行不可能となるからです。また、

```
(qname?var == 0)        /* syntax error */
```

または

```
(a > b && qname!123)    /* syntax error */
```

のように、Promela では条件文と送受信操作を組み合わせることはできません（このような条件文は副作用なしには評価できない事に注意してください）。上記のような受信文の代わりに、? のうしろに [] を使うことができます。

```
qname?[ack,var]
```

この文は、条件文として評価されます。もし以下の文が実行可能だったら（qname チャネルの先頭に ack メッセージがあったら）、1 を返します。そうでなければ、0 を返します。

```
qname?ack,var
```

以下の文は、受信できるかどうか評価されますが、決して受信しない、副作用のない文です。

```
qname?[ack,var]
```

以下のような non-atomic なシーケンスに注意してください。

```
(len(qname) < MAX) -> qname!msgtype
```

または

```
qname?[msgtype] -> qname?msgtype
```

これらの文は、2 つ目の文がひとつ目の文のあとに必ず実行されるとは限りません。複数のプロセス間で共有するチャネルにアクセスする場合はレースコンディションが存在するかもしれません。最初のケースでは、このプロセスがチャネルがいっぱいではないと決定した直後に他のプロセスがこのチャネルへメッセージを送信する事ができます。また、2 番目のケースは、われわれのプロセスがメッセージの存在を確認した直後に、他のプロセスがそのメッセージを奪い取ることができます。

ランデブ通信（Rendez-Vous Communication）

ここまではメッセージチャネルを介した非同期通信について見てきました。ここで N を正の整数とすると、バッファサイズ N のチャネルを以下のように宣言してきました。

```
chan qname = [N] of { byte }
```

このとき、論理的には以下のような宣言も認められます。

```
chan port = [0] of { byte }
```

これは、byte メッセージひとつをやりとりするランデブポートを定義しています。チャネルのサイズが 0 であるということは、このポートはメッセージを保持しておく事はできません。このようなランデブポートを介してのメッセージのやり取りは同期になります。つぎのような例を考えてみましょう。

```
#define msgtype 33

chan name = [0] of { byte, byte };

proctype A()
{       name!msgtype(124);
        name!msgtype(121)
}

proctype B()
{       byte state;
        name?msgtype(state)
}

init
{       atomic { run A(); run B() }
}
```

チャネル name はグローバルなランデブポートとして宣言されています。2 つのプロセスは最初の文を同期して実行します。msgtype によるハンドシェイクにより、124 が B のローカル変数

state へ転送されます。A の 2 番目の文は実行できません、なぜならプロセス B にそれに対応する受信操作がないからです。

もしチャネル name が 0 ではないバッファサイズで宣言されていると、振る舞いは異なります。バッファサイズが少なくとも 2 あれば、プロセス B がスタートする前であったとしてもプロセス A は実行を完了する事ができたでしょう。バッファサイズが 1 の場合は、次のようなシーケンスとなるでしょう。プロセス A が最初の送信操作を行った後、2 番目の送信操作でチャネルがフルのためブロックされます。続いて、プロセス B が最初のメッセージを受信します。この時点でプロセス A は再び実行可能となり、どのプロセスも受信しない余分なメッセージを送信します。

ランデブ通信は、2 者間で行われます。ランデブ通信により同期がとれるのは、送信プロセスと受信プロセスの 2 つのプロセスだけです。後半ではセマフォの構築するためにこのランデブ通信を利用する例を見ます。しかしその前に、役に立ちそうないくつかの制御構造について紹介したいと思います。

制御構造 (Control Flow)

前節までに、文の連結、プロセスの並行実行、アトミックシーケンスの 3 つの制御構造を見てきました。ここでは、さらに Promela でとりあげるべき 3 つの制御構造について紹介します。選択、反復、無条件ジャンプの 3 つです。

選択 (Case Selection)

最も単純な構造は選択構造です。たとえば、2 つの変数 a と b の関係を使って 2 つのオプションを書く事ができます。

```
if
:: (a != b) -> option1
:: (a == b) -> option2
fi
```

選択構造は 2 つの実行シーケンスからなります。それぞれの頭にはコロン 2 つをおきます。リストの中から実行されるシーケンスはひとつだけです。最初の文が実行可能なもののうちひとつが実行されます。最初の文は*ガード*と呼ばれます。

上記の例では、ガードは排他的になっていますが、必ずしもそうでなければならいというわけではありません。もしひとつ以上のガードが実行可能な場合、実行可能なもののうちどのシーケンスが実行されるかは非決定的です。もしすべてのガードが実行不可能だった場合、プロセスはすくなくともひとつのガードが選択できるようになるまでブロックされます。ガードとして使用できる文の種類に制限はありません。以下に示すように、メッセージの受信操作も書く

事ができます。

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A()
{       ch!a
}

proctype B()
{       ch!b
}

proctype C()
{       if
        :: ch?a
        :: ch?b
        fi
}

init
{       atomic { run A(); run B(); run C() }
}
```

この例では 3 つのプロセスとひとつのチャネルを定義しています。プロセス c の選択構造の最初のオプションはチャネルにメッセージ a（ここでは a はマクロで定義されているように定数 1）が含まれるとき実行可能になります。2 番目のオプションはメッセージ b（a と同様にこれも定数）が含まれるとき実行可能になります。プロセスの実行スピードに関して特に何も規定していないため、どちらのメッセージが最初に届くかは分かりません。

以下のプロセスは変数 count を一度だけ +1 するか、-1 します。

```
byte count;

proctype counter()
{
        if
        :: count = count + 1
        :: count = count - 1
        fi
}
```

反復（Repetition）

選択構造の論理的延長線上に反復構造があります。上記の例で示したプログラムに手を加えて、変数の値を繰り返しランダムにアップダウンさせるプログラムを得ることができます。

```
byte count;

proctype counter()
{
        do
```

```
        :: count = count + 1
        :: count = count - 1
        :: (count == 0) -> break
        od
}
```

実行のたびにひとつのオプションが選ばれます。そのオプションを実行し終わると、その構造を繰り返し実行します。反復構造を終了するには通常 break 文を使用します。この例では、その変数 count の値が 0 のときにループから抜ける事ができます。count が 0 のときに必ずしも break を実行するわけではないということに注意してください。他の 2 つのオプションは常に実行可能なので、count が 0 になったからといって必ずしも break されるわけではないのです。count が 0 になったときに必ず break するようにするには、以下のようにプログラムを変更する必要があります。

```
proctype counter()
{
        do
        :: (count != 0) ->
                if
                :: count = count + 1
                :: count = count - 1
                fi
        :: (count == 0) -> break
        od
}
```

無条件ジャンプ（Unconditional Jumps）

ループから抜けるもう一つの方法は、無条件ジャンプ、悪名高い goto 文、を使用する方法です。以下の例は、0 ではない 2 つの正の整数の最大公約数を見つけるユークリッドのアルゴリズムを実装したものです。

```
proctype Euclid(int x, y)
{
        do
        :: (x >  y) -> x = x - y
        :: (x <  y) -> y = y - x
        :: (x == y) -> goto done
        od;
done:
        skip
}
```

この例の goto ではラベル done へジャンプします。ラベルは文の前におく必要があります。上記ではプログラムの最後にジャンプさせたいのですが、こういったケースではダミー文の skip が役に立ちます。skip はいつでも実行可能な何も作用を及ぼさない文です。また goto は常に実行可能です。

以下の例ではチャネル in に入ってくるメッセージをその大きさによりそれぞれ large チャネルと small チャネルに分配するフィルタを定義しています。マクロにより定数 N は 128 に、size

は 16 に定義されています。

```
#define N    128
#define size  16

chan in    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };

proctype split()
{       short cargo;

        do
        :: in?cargo ->
                if
                :: (cargo >= N) ->
                        large!cargo
                :: (cargo <  N) ->
                        small!cargo
                fi
        od
}

init
{       run split()
}
```

以下に示すプロセス型 merge は、2つのストリームを再びひとつにまとめます。しかし、順番はもとのものは異なることになるでしょう。

```
proctype merge()
{       short cargo;

        do
        ::      if
                :: large?cargo
                :: small?cargo
                fi;
                in!cargo
        od
}
```

もし init プロセスを以下のようにしたら、split と merge プロセスは永遠に走り続けるでしょう。

```
init
{       in!345; in!12; in!6777;
        in!32;  in!0;
        run split();
        run merge()
}
```

最後の例は、ランデブ通信を使ってダイクストラのセマフォを実装したものです。

```
#define p       0
#define v       1
```

```
chan sema = [0] of { bit };

proctype dijkstra()
{       byte count = 1;

        do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}

proctype user()
{       do
        :: sema?p;
           /*     critical section */
           sema!v;
           /* non-critical section */
        od
}

init
{       run dijkstra();
        run user();
        run user();
        run user()
}
```

セマフォはクリティカルセクションに 2 つ以上のプロセスが同時にいないことを保障してくれます。しかし、ひとつのプロセスがクリティカルセクションを独占的にアクセスすることを回避してくれるわけではありません。

手続きのモデルリングと再帰（Modeling Procedures and Recursion）

手続きは、たとえ再帰的でも、プロセスによってモデル化可能です。返り値はグローバル変数か、メッセージによって呼び出しプロセスへ返されます。以下のプログラムはそれを示したものです。

```
proctype fact(int n; chan p)
{       chan child = [1] of { int };
        int result;

        if
        :: (n <= 1) -> p!1
        :: (n >= 2) ->
                run fact(n-1, child);
                child?result;
                p!n*result
        fi
}
init
{       chan child = [1] of { int };
        int result;

        run fact(7, child);
```

```
        child?result;
        printf("result: %d¥n", result)
}
```

プロセス *fact(n, p)* は、*n* の階乗を再帰的に計算し、親プロセス *p* へメッセージにより結果を返します。

タイムアウト（Timeouts）

前節までに Promela での定義済みの文を 2 つ、skip と break について紹介しました。もうひとつの定義済みの文は timeout です。timeout は、空のチャネルからメッセージを待つような、決して真にはならない条件を待つのを中断するためにつくられた、特別な条件をモデル化したものです。timeout キーワードは、システムがハングした状態からの脱出を Promela でモデル化するためのものです。timeout 条件は、分散システム内の他のすべての文が実行不可能になったときに真になります。絶対的なタイミングを考慮にいれていないことに注意してください。これは検証作業において極めて重要なことです。また、われわれは timeout がどのように実装されるべきかについては明示していません。以下に、システムが行き詰まったときにはいつでも *guard* チャネルに reset メッセージを送信するプロセスの例を示します。

```
proctype watchdog()
{
        do
        :: timeout -> guard!reset
        od
}
```

表明（Assertions）

Promela で多少説明が必要な重要なもうひとつの言語構成要素は assert 文です。この文は以下の形式をとります。

```
assert(any_boolean_condition)
```

assert はいつでも実行可能です。もし条件が満たされたならば、この文はなにも作用を及ぼしません。しかし、もし条件が満たされなかったら、この文はエラーを報告するでしょう。

高度な使い方（Advanced Usage）

このモデリング言語は検証面に特化したいつくかの特徴を持ちます。そのような特徴は、これから話題にあげるラベルの使い方や、Promela における timeout 文の意味や、assert のような文の使い方に見られます。

終了状態ラベル（End-State Labels）

検証言語として Promela を使用する場合は、モデル化した振る舞いについて非常に特殊な表明ができるようにしなければなりません。特に Promela によってデッドロックの存在をチェックする場合は、検証プログラムは、ユーザが望んでいないデッドロックと通常の*終了状態*を見分けられるようにしなければなりません。

Promela では、インスタンス化されたすべてのプロセスが定義されているプログラム本体の最後まで到達し、かつ、すべてのメッセージチャネルが空である状態を、通常の終了状態と見なします。しかし、すべてのプロセスがプログラム本体の最後まで到達するようにモデル化されるとは限りません。例えば、いくつかのプロセスは IDLE 状態に居続けるかもしれませんし、また、あるプロセスは新しいメッセージの受信をループの先頭で待ち続けるかもしれません。

そこで、検証プログラムがこのような終了状態とデッドロックを明確に見分けるために、Promela では終了状態ラベルを使う事ができます。たとえば、前述のプロセス型 dijkstra() にラベルをひとつ追加します*3。

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
                sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

end ラベルを追加することにより、プロセス型 dijkstra() がプロセス定義の最後の括弧まで到達しなくても、ループの先頭にいればエラーとはなりません。もちろん、このような状態はデッドロック状態の一部かもしれませんが、もしデッドロックがまだあるとしたら、それはこのプロセスが原因ではないでしょう（もし他のプロセスが正当な終了状態にいない場合は検証プログラムが報告してくれるでしょう）。

ひとつの検証モデルにつきひとつ以上の終了状態があるかもしれません。もしそのようなときは、プロセス内のラベルの名前はすべてユニークにしなければなりません。また、終了状態ラベルの名前は、かならず end の *3 文字から始める*必要があります。endde, end0, end_appel のようにラベル付けされた状態は検証プログラムにすべて正当な終了状態として扱われます。

プログレス状態ラベル（Progress-State Labels）

終了状態ラベルと同じように、ユーザは*プログレス状態*ラベルを定義する事ができます。この場合、プログレス状態ラベルはプロトコルが進展*しなければならない*状態にマークすることができます。プロトコルの実行中にこれらプログレス状態ラベルをひとつも通過しないすべての無限サイクルは、潜在的に餓死ループ（starvation loop）になり得ます。たとえば dijkstra の例では、セマフォテストの成功に "progress" というラベルをつけ、検証プログラムは、プロトコ

ルの実行中は必ずセマフォテストを成功するプロセスがあることを確認してくれます。

```
proctype dijkstra()
{       byte count = 1;

end:    do
        :: (count == 1) ->
progress:       sema!p; count = 0
        :: (count == 0) ->
                sema?v; count = 1
        od
}
```

もしひとつ以上の状態にプログレスラベルをつける場合は、progress0, progress_foo のような progress から始まるユニークな名前のラベルをつける必要があります。

-a フラグを指定して *Spin* に生成させた検証プログラムはランタイムオプション -l をサポートします。検証プログラムは、デフォルトではまずデッドロックの探索を行いますが、このフラグを指定した検証プログラムは non-progress ループの探索を行います。この探索はデッドロックの探索に比べ約 2 倍の時間とメモリ使用量を要します（標準の手法に対する大幅な改良は、強連結成分解析を基盤としています）。

メッセージタイプ定義（Message Type Definitions）

いままでは、どのように変数を宣言するのか、また、C スタイルのマクロを使ってどのように定数を定義するのか、を見てきました。Promela はメッセージタイプの定義のために以下のような定義も認めています。

```
mtype = { ack, nak, err, next, accept }
```

この方法は、それぞれのメッセージタイプに実際にどういった値を使用するかということを抽象している点でより好ましいやり方です。定数の名前はそのまま検証プログラムの中で使用する事ができるため、エラーの報告を改善する事ができます。

チャネルの宣言において mtype キーワードを使用する事により、対応するメッセージフィールドを数値としてではなく、記号として扱う事ができます。例えば、以下のように宣言できます。

```
chan q = [4] of { mtype, mtype, bit, short };
```

偽文（Pseudo Statements）

ここまでは、Promela で定義されるすべての基本文（代入、条件、送信／受信、assert、timeout、goto、break、skip）を見てきました。そして、文でありませんが chan, len, run といった条件文や代入文で使用する事ができる単項演算子についても見てきました。

skip 文は、実際にはまったく作用を及ぼさない、構文の要求を満たすのにフィルタの役割を果たす文として紹介しました。skip は形式的には言語の一部ではないのですが、単に 定数 (1) と書いた単純な条件文と同じ効果をもつ同義語であり、*偽文*です。同じような偽文に、(0) と等価な block や hang、assert(0) と等価な halt が考えられます（実際にはこれらは定義されていません）。else はもうひとつの偽文です。これは選択や反復の中で最後のオプションシーケンスの文として使用する事ができます。

```
if
:: a > b -> ...
:: else -> ...
fi
```

else は同じ選択の中の他のすべてのオプションが実行不可能なときにだけ実行可能になります。

例（Example）

Promela でモデル化した単純な（欠陥のある）プロトコルの例を示します。

```
mtype = { ack, nak, err, next, accept };

proctype transfer(chan in,out,chin,chout)
{       byte o, i;

        in?next(o);

        do
        :: chin?nak(i) ->
                        out!accept(i);
                        chout!ack(o)

        :: chin?ack(i) ->
                        out!accept(i);
                        in?next(o);
                        chout!ack(o)

        :: chin?err(i) ->
                        chout!nak(o)
        od
}

init
{       chan AtoB = [1] of { mtype, byte };
        chan BtoA = [1] of { mtype, byte };

        chan Ain  = [2] of { mtype, byte };
        chan Bin  = [2] of { mtype, byte };

        chan Aout = [2] of { mtype, byte };
        chan Bout = [2] of { mtype, byte };

        atomic {
          run transfer(Ain,Aout, AtoB,BtoA);
          run transfer(Bin,Bout, BtoA,AtoB)
```

```
                };

                AtoB!err(0)
        }
```

チャネル Ain と Bin は、ここでは定義していないバックグランドプロセス（この転送システムのユーザプロセス）によって、メッセージタイプ next と任意の値（ASCIIの文字コード）のトークンメッセージで埋められます。同様に、ユーザプロセスは、チャネル Aout, Bout からデータを受信する事ができます。チャネルとプロセスは、アトミックに初期化され、ダミーの err メッセージによりスタートします。

Spin（Spin）（Spin.htmlも見てください）

Promela で記述したシステムのモデルが与えられたとき、 *Spin* はシステムのランダムシミュレーションをしたり、システムの状態空間を素早く探索する C 言語の検証プログラムを生成したりすることができます。この検証プログラムは、たとえば、プロトコルを実行中、ユーザが記述したシステムの不変条件を違反していないか、チェックします。

何もオプションを指定しないで *Spin* を起動した場合は、ランダムシミュレーションを行います。オプション -n N を指定した場合は、ランダムシミュレーションのための種を整数 *N* に設定します。

オプション pglrs は、ユーザがシミュレーションについて出力したい情報のレベルを設定するのに使用します。出力のすべての行は、通常モデルのソースの行番号への参照を含みます。

-p は、すべてのステップに対して Promela プロセスの状態変化を表示します。

-g は、すべてのステップに対してそのときのグローバル変数の値を表示します。

-l は、ローカル変数をもつプロセスが状態変化したあとのローカル変数の値を表示します。 -p オプションを組み合わせて使うといいでしょう。

-r はすべてのメッセージ受信イベントを表示します。これは受信したプロセスの名前と番号、ソースの行番号、メッセージのパラメータ番号（パラメータごとに 1 行になります）、メッセージタイプとメッセージチャネルの名前と番号を表示します。

-s はすべてのメッセージ送信イベントを表示します。

Spin ではさらに 3 つのオプション（-a、-m、-t）を使用する事ができます。

-a はプロトコルに特化した検証プログラムを生成します。この出力は pan.[cbhmt] と名付けられるいくつかの C 言語のファイルに書き出されます。生成されたファイルをコンパイルする事により実行可能な検証器が生成されます。状態空間をくまなく探索する場合は、単に次のよう

にコンパイルするだけでいいでしょう。

```
$ gcc -o pan pan.c
```

大きなシステムの場合は、コンピュータのメモリを使い尽くすかもしれません。そのような大きなシステムの検証をする場合は、メモリを効率的に使うビット状態空間法を使う事により検証が可能になるかもしれません。ビット状態空間法を使用するときは以下のようにコンパイルします。

```
$ gcc -DBITSTATE -o pan pan.c
```

このような探索のカバレージは、*ハッシュ要素*（詳細は後述）によります。生成された実行可能な検証プログラム自身にもオプションを指定する事ができます。検証プログラムに指定できるオプションは、上記のようにコンパイルした場合は "`./pan -?`" により確認できます（検証プログラムに指定できるオプションについては、後述の *「検証プログラムを使用する」*で説明します）。

`-m` は、メッセージ送信のデフォルトの振る舞いを変更するのに使用します。通常は、送信操作はターゲットとなるチャネルがいっぱいでないときだけ実行可能になります。これは暗黙のうちに同期を引き起こす事がありますが、それはいつでも望まれるものではありません。`-m` オプションを指定する事により送信操作は常に実行可能になります。いっぱいになっているチャネルへ送信したメッセージは失われます。このオプションを `-a` オプションと組み合わせて検証プログラムを生成した場合は、生成される検証プログラムの振る舞いも同様に異なります。検証する際には、メッセージが失われる事も考慮に入れる必要があるでしょう。

`-t` は、軌跡表示用のオプションです。検証プログラムが `assert` に対する違反やデッドロックや望まないメッセージの受信を見つけた場合、検証プログラムは `pan.trail` という名のファイルにエラーの軌跡を書き出します。*Spin* に `-t` オプションを指定する事によりその詳細を見る事ができます。他のオプション（`pglrs`）と組み合わせる事によりエラーシーケンスに対するいろいろな視点を簡単に手に入れる事ができるでしょう。

ここで紹介しなかった *Spin* の他のオプションについては省略します。詳しくは、[5] を見てください。またヒントとして、このマニュアルの最後にある "さらに深く掘り下げる" も見てください。

シミュレータ（The Simulator）

以下に示すプロトコルの例を考えてみましょう。ここでは、以下のモデルを `lynch` というファイルに保存したとします。

```
1  #define MIN       9
2  #define MAX       12
3  #define FILL      99
```

```
 4
 5  mtype = { ack, nak, err }
 6
 7  proctype transfer(chan chin, chout)
 8  { byte o, i, last_i=MIN;
 9
10    o = MIN+1;
11    do
12    :: chin?nak(i) ->
13            assert(i == last_i+1);
14            chout!ack(o)
15    :: chin?ack(i) ->
16            if
17            :: (o <  MAX) -> o = o+1
18            :: (o >= MAX) -> o = FILL
19            fi;
20            chout!ack(o)
21    :: chin?err(i) ->
22            chout!nak(o)
23    od
24  }
25
26  proctype channel(chan in, out)
27  { byte md, mt;
28    do
29    :: in?mt,md ->
30            if
31            :: out!mt,md
32            :: out!err,0
33            fi
34    od
35  }
36
37  init
38  { chan AtoB = [1] of { mtype, byte };
39    chan BtoC = [1] of { mtype, byte };
40    chan CtoA = [1] of { mtype, byte };
41    atomic {
42            run transfer(AtoB, BtoC);
43            run channel(BtoC, CtoA);
44            run transfer(CtoA, AtoB)
45    };
46    AtoB!err,0;     /* start */
47    0               /* hang */
48  }
```

このプロトコルはメッセージタイプ ack、nak、err を使用します。err は、2 つの転送プロセス間のチャネル通信におけるメッセージのゆがみをモデル化するのに使用しています。チャネルの振る舞いはチャネルプロセスによりモデル化しています。また転送プロセスの 2 つのローカル変数間の不変条件（上記のものはわざと間違っています）を assert を使って表明しています。

何もオプションを指定しないで *Spin* を走らせると、ランダムシミュレーションを行い、実行が終わると単に出力するか、実行中に通った printf 文を出力します。この場合は、以下のような出力になります。

```
$ spin lynch    # no options, not recommended
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer)      line 11 (state 15)
proc  2 (channel)       line 28 (state 6)
proc  1 (transfer)      line 13 (state 3)
proc  0 (:init:)        line 48 (state 6)
4 processes created
$
```

上記のモデルには printf はないので printf に対する出力はありませんが、プロセスが assert に違反したため、そのことが出力されています。さらに知りたい場合は、よりたくさん（たとえば、すべての受信イベント）出力するように走らせる事もできます。その結果を下の図 1 に示します。出力についてはほとんど説明するまでもないでしょう。

上記のシミュレーションでは何度行っても同じ assert 違反を検出します。シミュレーションはランダムに非決定的選択をしているので、つねにこのケースの違反を見つける必要はありません。再現可能な実行シーケンスを強制する場合は、-n N オプションを使います。例えば、

```
$ spin -r -n100 lynch
```

とすると、ランダムジェネレータの種に整数 100 を使用し、毎回同じ出力が得られる事を保証します。

別のオプションを指定する事により、シミュレーションの実行による出力を多く出す事ができますが、出力されるテキストはすぐに膨大な量になります。それに対する簡単な解は *grep* を使って出力をフィルタリングする事です。たとえば、上記の例においてチャネルプロセスの振る舞いしか興味なかったとしてら、以下のようにします。

```
$ spin -n100 -r lynch | grep "proc  2"
```

そうすると、図 1 のような結果になります。

```
$ spin -r lynch
proc  1 (transfer) line  21, Recv err,0  <- queue 1 (chin)
proc  2 (channel)  line  29, Recv nak,10 <- queue 2 (in)
proc  3 (transfer) line  12, Recv nak,10 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,10 <- queue 1 (chin)
...
proc  1 (transfer) line  15, Recv ack,12 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  15, Recv ack,99 <- queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,99 <- queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <- queue 2 (in)
proc  3 (transfer) line  21, Recv err,0  <- queue 3 (chin)
proc  1 (transfer) line  12, Recv nak,99 <- queue 1 (chin)
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer) line 11 (state 15)
proc  2 (channel)  line 28 (state 6)
proc  1 (transfer) line 13 (state 3)
proc  0 (:init:)   line 48 (state 6)
```

```
4 processes created
$ spin -n100 -r lynch | grep "proc  2"
proc  2 (channel) line 29, Recv nak,10 <- queue 2 (in)
proc  2 (channel) line 29, Recv ack,11 <- queue 2 (in)
proc  2 (channel) line 29, Recv ack,12 <- queue 2 (in)
proc  2 (channel) line 28 (state 6)
```

図 1. シミュレーションの出力

以下のような結果を得るには、-c オプション（バージョン3.0での新機能）を使います。

```
$ spin -c lynch
proc 0 = :init:
proc 1 = transfer
proc 2 = channel
proc 3 = transfer
q¥p   0   1   2   3
  1   AtoB!err,0
  1   .   chin?err,0
  2   .   chout!nak,10
  2   .   .   in?nak,10
  3   .   .   out!err,0
  3   .   .   .   chin?err,0
  1   .   .   .   chout!nak,10
  1   .   chin?nak,10
  2   .   chout!ack,10
  2   .   .   in?ack,10
  3   .   .   out!ack,10
  3   .   .   .   chin?ack,10
  1   .   .   .   chout!ack,11
  1   .   chin?ack,11
  2   .   chout!ack,11
  2   .   .   in?ack,11
  3   .   .   out!ack,11
  3   .   .   .   chin?ack,11
  1   .   .   .   chout!ack,12
  1   .   chin?ack,12
  2   .   chout!ack,12
  2   .   .   in?ack,12
  3   .   .   out!ack,12
  3   .   .   .   chin?ack,12
  1   .   .   .   chout!ack,99
  1   .   chin?ack,99
  2   .   chout!ack,99
  2   .   .   in?ack,99
  3   .   .   out!err,0
  3   .   .   .   chin?err,0
  1   .   .   .   chout!nak,99
  1   .   chin?nak,99
spin: line  13 "lynch", Error: assertion violated
-------------
final state:
-------------
#processes: 4
 79:    proc  3 (transfer) line  11 "lynch" (state 15)
 79:    proc  2 (channel) line  28 "lynch" (state 6)
 79:    proc  1 (transfer) line  13 "lynch" (state 3)
 79:    proc  0 (:init:) line  47 "lynch" (state 6)
4 processes created
```

最初の列がチャネルの id 番号、最初の行がプロセス id 番号で、各行はどのプロセスがメッセージの転送を行ったかを出力しています。

検証プログラム（The Analyzer）（Pan.html と Roadmap.html も見てください）

シミュレーションは新しい設計した結果についてざっとデバッグするには役に立ちますが、そのシステムが本当にエラーを含んでいないかどうかを証明する事はできません。 *Spin* では -a と -t オプションを使う事により、たとえ非常に大きなモデルであったとしても検証する事ができます。

プロトコルモデルに対する状態空間探索プログラムは以下の 1 行目のようにして作られる 5 つファイルからなります。

```
$ spin -a lynch
$ wc pan.[bchmt]
     99     285    1893 pan.b
   3158   10208   70337 pan.c
    356    1238    7786 pan.h
    216     903    6045 pan.m
    575    2099   14017 pan.t
   4404   14733  100078 total
```

pan.c はプロトコルの検証を行う C 言語のコードのほとんどを含みます。*pan.t* はプロトコルの制御フローをエンコードした遷移マトリクスを含みます。*pan.b* と *pan.m* は順方向の遷移と逆方向の遷移の C のコード、*pan.h* はヘッダファイルです。これらはいくつかの方法でコンパイルする事ができます（たとえば、全状態空間探索するとか、ビット状態空間を使用するとか）。

全探索（Exhaustive Search）

システムの状態空間がおおよそ100,000ほどまでならば、もっとも良い方法は、デフォルトのままプログラムをコンパイルすることです。

```
$ gcc -o pan pan.c
```

コンパイルによってできた実行プログラム *pan* を実行する事により検証する事ができます。この場合の検証は全探索を行います、つまり、可能なすべてのイベントシーケンスをテストします。もちろん、assert 違反があればそれを見つけ出します。

```
$ ./pan
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash conflicts: 0 (resolved)
```

```
(size 2^18 states, stack frames: 0/5)
```

(より新しいバーションの出力形式は、同じ情報を含んでいますが、もっと洗練されています。)

出力の最初の 1 行目は、assert 違反があった事を報告し、不変条件が満たされなかった最初の反例を教えてくれます。この違反は 61 状態が生成された後に発見されています。ハッシュ "conflicts" は、状態空間をアクセスしている間におきたハッシュの衝突回数を示しています。全探索モードでは、すべての状態はリスト（linked list）に格納して、すべての衝突を解決しています。この場合における最も関心がある部分は、3 行目でしょう。これは、シミュレータによって再現可能なエラーシーケンスが trail ファイルに作られたことを示しています。そこで以下のようにすると、

```
$ spin -t -r lynch | grep "proc  2"
```

エラーの原因がなにであったか特定する事できるでしょう。assert が適切であれば、検証プログラムはその違反を見つけてくれます。もしこのような違反が報告されなかった場合は、間違いなく assert 違反を犯すような実行シーケンスがなかったということでしょう。

オプション（Options）

検証プログラムはいくつかのオプションが指定できるように生成されます。どのようなオプションがあるかは、以下のようにするとチェックできます。

```
$ ./pan --
-cN stop at Nth error (default=1)
-l  find non-progress cycles    # when compiled with -DNP
-a  find acceptance cycles      # when not compiled with -DNP
-mN max depth N (default=10k)
-wN hash table of 2^N entries (default=18)
 ...etc.
```

最初の -c オプションの引数に 0 を設定すると、検証プログラムはエラーを見つけても状態空間探索を続けます。エラーが見つからなかった場合や -c0 オプションが指定された場合は、その検証が終了すると、実行できなかった（到達できなかった）コードに関する要旨を示します。この場合の出力は以下のようになります。

```
$ ./pan -c0
assertion violated (i == (last_i + 1))

vector 64 byte, depth reached 60, errors: 5
     165 states, stored
       5 states, linked
      26 states, matched
hash conflicts: 1 (resolved)
(size 2^18 states, stack frames: 0/6)

unreached code :init: (proc 0):
        reached all 9 states
unreached code channel (proc 1):
```

```
            line 35 (state 9),
          reached: 8 of 9 states
unreached code transfer (proc 2):
            line 24 (state 18),
          reached: 17 of 18 states
```

上記は、5 つの assert 違反があったことと 165 のユニークなシステムの状態が作られた事を示しています。状態に関しては、各状態の *ベクタサイズ*は 64 byte で、循環しない最長のシーケンスは 60 であることを示しています。チャネルプロセスと転送プロセスのどちらにも到達していない状態がひとつあることも示しています。到達していない状態は、それぞれのプロセスの do ループの直後です。どちらのループも終了しないことを意図して作られていることに注意してください。

-l オプションを指定すると、検証プログラムはデッドロックや assert 違反より non-progress ループを探します。このオプションについては "より高度な使い方" で説明します。

検証プログラムはさらに 2 つのオプションをサポートしています。デフォルトでは探索の深さは 10,000 ステップに制限されています。もし探索が最大探索深さにまで達した場合は、探索を打ち切るるため検証は部分的になります。確実に全探索を行いたい場合は、"depth reached" が最大探索深さ以内である事を確認し、もしそうでなかった場合は -m オプションを指定して検証を繰り返す必要があります。

-m オプションは、もちろん明示的に探索を打ち切るのにも使用できます。たとえば、assert を違反する実行シーケンスのうち最短のものを探したいときに使います。このように探索を打ち切る場合は、その探索深さの中での可能なすべての違反を検出する事を保証しません。

最後のオプションは -w N です。これは全状態空間の検証のスコープではなく、実行時間に影響を及ぼします。"hash table width" は通常検証プログラムが生成するシステムのユニークな状態数の予想値の対数と等しく、できれば、大きく設定しなければなりません（もしこれを小さく設定すると、ハッシュの衝突が増加し、探索に時間かかるようになるでしょう）。このデフォルト値 18 では 262,144 状態を扱うことができ、全状態空間探索検証ではこのデフォルト値のままで十分となるようにすべきです。

ビット状態空間検証（Bit State Space Analyses）

全状態空間探索で検証するときに必要となるメモリ量は簡単に計算する事ができます[4]。もし、上記のプロトコルの例のように、1 状態をエンコードするのに 64 byte 必要で、探索に使用できるシステムのメモリ量が 2MB だとすると、32,768 状態格納する事ができます。もしシステムの状態空間の中で到達可能な状態数がこれ以上になった場合、検証は失敗となります。いまのところ、*Spin* はこのトラップを避ける事ができる*唯*一の検証システムです。他の自動検証システムは（それが基盤としている形式主義に関係なく）、メモリを使い果たすとユーザに役立つ答えを返す事なく検証をやめてしまいます。

従来の検証方法では、メモリの制限に達した場合、そのカバレージは急激に下がってしまいます。たとえば、格納できる状態数の2倍の状態数をもつ状態空間を検証するとき、その探索の有効なカバレージはたったの 50% です。*Spin* ではこのようなケースでもビット状態空間法[4]を使う事により十分に良い結果を得られます。ビット状態空間を使うには以下のように検証プログラムをコンパイルします。

```
$ gcc -DBITSTATE -o pan pan.c
```

このようにコンパイルされた検証プログラムでももちろん同じように `assert` 違反を見つけ出す事ができます。

```
$ ./pan
assertion violated (i == ((last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash factor: 67650.064516
(size 2^22 states, stack frames: 0/5)
$
```

実際、中程度の大きさの問題を小さくするためにこの方法を使った場合は、全状態空間探索を比べてもほとんど違いはありません（ビット状態空間法の方が多少速く、メモリ使用量が格段に押さえられていることをのぞいてはですが）。しかし大きな問題を扱ったときには、より違いが表れます。出力の最後の 2 行に、*カバレージ*の見積もりが出るので大きなモデルを検証するときには役に立つでしょう。ビット状態空間が適応できる状態の最大数は、最後の行に書いてあります（ここでは、2^{22} byte または 3200 万ビット（状態））。その上の行は*ハッシュ要素*を示しています。これは実際の状態数で割った状態の最大数におおよそ等しいです。大きなハッシュ要素（100 以上）は信頼性の高い事を示し、そのカバレージは 99% か 100% です。ハッシュ要素が 1 に近づくと、カバレージは 0% に近づきます。

従来の検証システムが探索できなかった、もしくは、低いカバレージとなっていたケースにおいて部分的なカバレージを実現しているという点に十分に注意してください。しかし *Spin* が生成する答えが従来の自動検証システムより信頼性が低いということは *ありません*。

ビット状態検証の目標値は、ビット状態空間のメモリの最大使用量をアロケートすることによりハッシュ要素を 100 以上にすることです。最善の結果を得るには、`-w N` オプションによって状態空間のサイズを正確にあなたのコンピュータで獲得できる実際のメモリ量に適したものにすることです。デフォルトでは *N* は 22 で、これに対応する状態空間は 4MB です。たとえば、あなたのコンピュータが 128 MB のメモリを持っている場合、`-w27` と指定する事により多くの状態について調べる事ができるでしょう。

まとめ（Summary）

このマニュアルの最初の部分では、非同期データ通信に限らず Promela という名の言語による並行システムのモデルの表記法について説明しました。この言語はいくつかのちょっとかわった特徴を持っています。プロセス間の通信は、メッセージか共有変数を介して行われます。同期、非同期通信はどちらも一般化したメッセージ通信の機構の 2 つのケースとしてモデル化されます。Promela のすべての文は潜在的にモデルをブロックさせる可能性があります。それはその文が実行可能か、実行不可能かにより、ほとんどのケースはプロセスが走っている環境の状態によります。プロセス同士の相互作用とプロセス協調はこの言語の根底にあるものです。Promela の設計、*Spin* による検証とその応用範囲についての詳細は、[5] にあります。

Promela は、プログラム言語ではなく、検証用のモデリング言語を意図して作られています。例えば、手の込んだ抽象データ型は含んでいませんし、変数の基本型もわずかしかありません。検証モデルはプロトコルの実装を抽象化したものです。抽象化により、プロセスの相互作用の本質的要素を分離して調査する事ができるのです。抽象化により実装とプログラミングに関する詳細を抑え込むのです。

Promela の式、宣言、代入の構文はおおざっぱに C 言語 [6] を基礎としています。Promela は E.W. ダイクストラ [1] と C.A.R. ホーア [3] の "ガードコマンド言語" の影響を大いに受けています。しかし、それらとは重大な違いがあります。ダイクストラの言語はプロセスの相互作用を言語のプリミティブとして持っていません。ホーアの言語は、同期通信のみをベースとしています。またホーアの言語では、ガードの中で使用できる文の種類に制限があります。Promela の選択と循環文の意味は、他のガードコマンド言語と異なります。すべてのガードが偽のときは、その文は中断されるのではなくブロックされます（必要であれば同期がとられます）。

ユーザは最小限の努力をするだけで、Promela の検証モデルから洗練された検証プログラムを生成することができます。*Spin* 自身、そして生成される検証プログラムどちらも ANSI C で記述されているため、*Unix* システム間で移植が可能です。それらは、ホストコンピュータの物理的な限界を引き出すように拡大縮小でき、プロトコルの検証においてその制限の中でできうる最善の検証を実現するものです。

付録：検証スイートの構築（Appendix: Building a Verification Suite）

Exercises.html と Roadmap.html も見てください。

Spin を使った検証においてまず第一にやらなければならない仕事は、手で問題の忠実なモデルを Promela により記述することです。この言語はわざと小さく作られています。モデリングの目的は調査すべき協調問題に関するシステムの側面を抽出する事です。他の詳細部分は、捨象

します。形式的には、モデルは縮小版のシステムであり、検証する特性に関する部分だけフルシステムと等価である必要があります。いったんモデルを構築すると、それは、その特性を検証するために使われる、われわれが "検証スイート" と呼ぶ一連の構成物の基盤となります。検証スイートを構築するために、モデルに assert を仕掛ける事ができます。assert により、モデルの中の変数の値やイベントのシーケンスに関する不変条件を形式化することができます。

以下で取り上げる最初の例は、ずいぶん昔に議論され、1966 年に H. Hyman が Communications of the ACM で発表した排他制御の問題に対する解です。

```
 1 Boolean array b(0;1) integer k, i,
 2 comment process i, with i either 0 or 1, and k = 1-i;
 3 C0: b(i) := false;
 4 C1: if k != i then begin;
 5 C2: if not b(1-i) then go to C2;
 6     else k := i; go to C1 end;
 7     else critical section;
 8     b(i) := true;
 9     remainder of program;
10     go to C0;
11     end
```

この解は、Dekker の初期の解として知られるもので、2 つのプロセス（番号 0 と 1）を対象としたものです。いま、Hyman の解がクリティカルセクションに対するアクセスの排他制御を完全に保証している事を証明したいとしましょう。われわれの最初の仕事は、Promela でこの解のモデルを作る事です。ここで、この解で使われている変数に対する名前をよりわかりやすいものにします。

```
 1  bool want[2];     /* Bool array b */
 2  bool turn;        /* integer    k */
 3
 4  proctype P(bool i)
 5  {
 6    want[i] = 1;
 7    do
 8    :: (turn != i) ->
 9            (!want[1-i]);
10            turn = i
11    :: (turn == i) ->
12            break
13    od;
14    skip; /* critical section */
15    want[i] = 0
16  }
17
18  init { run P(0); run P(1) }
```

このモデルの検証プログラムを生成、コンパイルすることで、デッドロックのような主要な問題があれば、それを見つける事ができます。

```
$ spin -a hyman0
$ gcc -o pan pan.c
$ ./pan
```

```
full statespace search for:
assertion violations and invalid endstates
vector 20 byte, depth reached 19, errors: 0
      79 states, stored
       0 states, linked
      38 states, matched        total: 117
hash conflicts: 4 (resolved)
(size 2^18 states, stack frames: 3/0)

unreached code _init (proc 0):
        reached all 3 states
unreached code P (proc 1):
        reached all 12 states
```

このモデルは最初のテストにパスしました。しかし、われわれが本当に興味があるのは、このアルゴリズムが排他制御を保証しているかどうか、についてです。これにはいくつかの方法がありますが、最も簡単な方法は単に正当性を表明する assert をモデルに加えることです。

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ->
10            (!want[1-i]);
11            turn = i
12    :: (turn == i) ->
13            break
14    od;
15    skip; /* critical section */
16    cnt = cnt+1;
17    assert(cnt == 1);
18    cnt = cnt-1;
19    want[i] = 0
20  }
21
22  init { run P(0); run P(1) }
```

ここでは、グローバル変数 cnt を追加し、クリティカルセクションに侵入したときに +1、クリティカルセクションからでるときに -1 するようにしました。この変数の値がとりうる最大値は 1 であり、ひとつのプロセスがクリティカルセクション内にいるときのみ 1 になります。

```
$ spin -a hyman1
$ gcc -o pan pan.c
$ ./pan
assertion violated (cnt==1)
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 20 byte, depth reached 25, errors: 1
     123 states, stored
       0 states, linked
```

```
      55 states, matched        total: 178
hash conflicts: 42 (resolved)
(size 2^18 states, stack frames: 3/0)
```

検証プログラムは assert 違反があったと報告しています。このエラーシーケンスをチェックするために *Spin* に -t オプションを指定してその軌跡を見てみましょう。

```
$ spin -t -p hyman1
proc  0 (_init) line 24 (state 2)
proc  0 (_init) line 24 (state 3)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 9 (state 2)
proc  2 (P)     line 10 (state 3)
proc  2 (P)     line 11 (state 4)
proc  1 (P)     line 8 (state 7)
proc  1 (P)     line 12 (state 5)
proc  1 (P)     line 15 (state 10)
proc  2 (P)     line 8 (state 7)
proc  2 (P)     line 12 (state 5)
proc  2 (P)     line 15 (state 10)
proc  2 (P)     line 16 (state 11)
proc  2 (P)     line 17 (state 12)
proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 16 (state 11)
proc  1 (P)     line 17 (state 12)
spin: "hyman1" line 17: assertion violated
step 17, #processes: 3
                want[0] = 1
                _p[0] = 12
                turn[0] = 1
                cnt[0] = 2

proc  2 (P)     line 18 (state 13)
proc  1 (P)     line 17 (state 12)
proc  0 (_init) line 24 (state 3)
3 processes created
```

さて、もうひとつのエラーの発見方法を見てみましょう。もう一度モデルにクリティカルセクション内のプロセスの数に関する情報を加えます。

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ->
10            (!want[1-i]);
11            turn = i
12    :: (turn == i) ->
13            break
14    od;
15    cnt = cnt+1;
16    skip;   /* critical section */
17    cnt = cnt-1;
18    want[i] = 0
```

```
19  }
20
21  proctype monitor()
22  {
23    assert(cnt == 0 || cnt == 1)
24  }
25
26  init {
27    run P(0); run P(1); run monitor()
28  }
```

今度は、カウンタ cnt の値に関する不変条件は、独立したプロセス monitor() （名前に特に意味はないのですが）におかれました。この追加されたプロセスは、2 つのプロセスとともに走ります。このプロセスは 1 ステップを実行し終了しますが、これはシステムの実行中の *どの*タイミングでも実行する可能性があります。Promela によりモデル化されたシステム、および、*Spin* により検証されるシステムは、完全な非同期です。よって、 *Spin* による検証は 3 つのプロセスの可能なすべてのタイミングを検証の対象とするため、この assert は他の 2 つのプロセスの生存期間のあらゆるタイミングで評価されるのです。もし検証プログラムが違反を報告しなかった場合は、われわれは assert 違反を犯すような実行シーケンスが（3 つのプロセスの実行スピードによらず）まったくないことと結論づける事ができるのです。monitor プロセスによるシステムの不変条件の妥当性をチェックする方法は洗練された方法です。検証プログラムは以下のような結果を報告するでしょう。

```
$ spin -a hyman2
$ gcc -o pan pan.c
$ ./pan
assertion violated ((cnt==0)||(cnt==1))
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 24 byte, depth reached 26, errors: 1
     368 states, stored
       0 states, linked
     379 states, matched        total: 747
hash conflicts: 180 (resolved)
(size 2^18 states, stack frames: 4/0)
```

2 つのプロセスと 3 番目の monitor プロセスとの余分なインターリービングのため、探索すべきシステム状態数が増えていますが、エラーは正しく報告されています。

もうひとつの例 (Another Example)

システムの正当性がいつでもグローバルな不変条件という観点から与えられるという訳ではありません。この例はそれを示したものです。これは単純なビット交換プロトコル（alternating bit protocol）で、メッセージの損失、ゆがみをモデリングし、否定応答（negative acknowledgements）により拡張したものです。

```
 1  #define MAX       5
 2
 3  mtype = { mesg, ack, nak, err };
 4
 5  proctype sender(chan in, out)
 6  { byte o, s, r;
 7
 8    o=MAX-1;
 9    do
10    :: o = (o+1)%MAX; /* next msg */
11  again: if
12         :: out!mesg(o,s) /* send */
13         :: out!err(0,0)  /* distort */
14         :: skip       /* or lose */
15         fi;
16         if
17         :: timeout   -> goto again
18         :: in?err(0,0) -> goto again
19         :: in?nak(r,0) -> goto again
20         :: in?ack(r,0) ->
21            if
22            :: (r == s) -> goto progress
23            :: (r != s) -> goto again
24            fi
25         fi;
26  progress: s = 1-s /* toggle seqno */
27    od
28  }
29
30  proctype receiver(chan in, out)
31  { byte i;         /* actual input   */
32    byte s;         /* actual seqno   */
33    byte es;        /* expected seqno */
34    byte ei;        /* expected input */
35
36    do
37    :: in?mesg(i, s) ->
38             if
39             :: (s == es) ->
40                    assert(i == ei);
41  progress:         es = 1 - es;
42                    ei = (ei + 1)%MAX;
43                    if
44    /* send,   */   :: out!ack(s,0)
45    /* distort */   :: out!err(0,0)
46    /* or lose */   :: skip
47                    fi
48             :: (s != es) ->
49                    if
50    /* send,   */   :: out!nak(s,0)
51    /* distort */   :: out!err(0,0)
52    /* or lose */   :: skip
53                    fi
54             fi
55    :: in?err ->
56             out!nak(s,0)
57    od
58  }
59
60  init {
61    chan s_r = [1] of { mtype,byte,byte };
```

```
62    chan r_s = [1] of { mtype,byte,byte };
63    atomic {
64            run sender(r_s, s_r);
65            run receiver(s_r, r_s)
66    }
67  }
```

このプロトコルが正しくデータを転送するという命題をテストするために、このモデルにはすでにいくつかの assert が挿入されています。sender はメッセージとして MAX を法とする剰余からなる無限列を転送するように定義されています。MAX の値は重要ではなく、プロトコルの中でシーケンス番号の幅（今のケースでは 2）より大きければ何でもかまいません。たとえメッセージの損失の可能性があろうとも送信されたデータが消えたり、並べ替えられたりすることなく receiver に届けられることが検証したいのです。40 行目の assert により正確にそれを検証します。もし上記に書いた要求が満たせない可能性がある場合は、この assert の違反が検出されます。

最初の検証では、そういった問題の可能性がなく、われわれを安心させてくれます。

```
$ spin -a ABP0
$ gcc -o pan pan.c
$ ./pan
full statespace search for:
assertion violations and invalid endstates
vector 40 byte, depth reached 131, errors: 0
     346 states, stored
       1 states, linked
     125 states, matched         total: 472
hash conflicts: 17 (resolved)
(size 2^18 states, stack frames: 0/25)

unreached code _init (proc 0):
        reached all 4 states
unreached code receiver (proc 1):
        line 58 (state 24)
        reached: 23 of 24 states
unreached code sender (proc 2):
        line 28 (state 27)
        reached: 26 of 27 states
```

しかし注意してください。この結果は送信されたメッセージが正しい順番で届けられたことを意味しているのにすぎません。われわれはデータがかならず転送されるということはチェックしていません。sender と receiver が何も有効なメッセージを送らずエラーメッセージを交換してぐるぐるまわっている可能性もあります。これをチェックするために、sender と receiver がミスする事なく前進していることを示す各プロセスの状態に "プログレス状態" のラベルをはります（実際には片方のプロセスだけで十分なのですが）。

これでプログレス状態をまったく通過しないまま実行サイクルが繰り返される事がない事を検証できるようになりました。先の実行ファイルはそのまま使えないので、ノンプログレスサイクルを見つけるために検証プログラムをコンパイルしなおします。

```
$ gcc -DNP -o pan pan.c
$ ./pan -l
pan: non-progress cycle (at depth 6)
pan: wrote pan.trail
full statespace search for:
assertion violations and non-progress loops
search was not completed
vector 44 byte, depth reached 8, loops: 1
      12 states, stored
       1 states, linked
       0 states, matched        total: 13
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/1)
```

結果によると、少なくともひとつのノンプログレスサイクルがあります。最初に見つけたものは検証プログラムにより trail ファイルにその軌跡が保存されていて、われわれをそれを見る事ができます。その結果を図 2 の前半に示します。メッセージのゆがみや損失が何度も繰り返し続けられており、事実エラーですが、エラーシーケンスとしてはあまり面白くありません。ノンプログレスサイクルがいくつ存在するのか見るために、 -c フラグを使ってみます。-c オプションの引数に 0 を設定した場合は、エラーの合計数が出力されます。

```
$ pan -l -c0
full statespace search for:
assertion violations and non-progress loops
vector 44 byte, depth reached 137, loops: 92
     671 states, stored
       2 states, linked
     521 states, matched         total: 1194
hash conflicts: 39 (resolved)
(size 2^18 states, stack frames: 0/26)
```

エラーは、92 ケースあります。 -c オプションを -c1, -c2, -c3, ... のように順に使ってそのシーケンスをひとつずつ見る事ができます。しかし、最初に見つかったメッセージの損失を繰り返す事を原因とするエラーシーケンスを取り除いて、少しやるべき事を減らすことができます。そのために "progress" の 8 文字から始まる接頭語をつけたラベルをメッセージを損失している各行（13, 43, 48 行目）に張り、残るエラーサイクルを見ます（ラベルは :: の後ろに挿入します）*4。

```
$ spin -a ABP1
$ gcc -DNP -o pan pan.c

$ ./pan -l
pan: non-progress cycle (at depth 133)
pan: wrote pan.trail

full statespace search for:
assertion violations and non-progress loops
search was not completed

vector 44 byte, depth reached 136, loops: 1

     148 states, stored
       2 states, linked
```

```
        2 states, matched         total: 152

  hash conflicts: 0 (resolved)
  (size 2^18 states, stack frames: 0/26)
```

今度は、生成された軌跡によりこのプロトコルにおける真に深刻なバグを明らかにすることができます。図 2 の後半にその軌跡を示します。

```
$ spin -t -r -s ABP0
<<<<>>>>
proc  1 (sender)   line  13, Send err,0,0 -> queue 2 (out)
proc  2 (receiver) line  55, Recv err,0,0 <- queue 2 (in)
proc  2 (receiver) line  56, Send nak,0,0 -> queue 1 (out)
proc  1 (sender)   line  19, Recv nak,0,0 <- queue 1 (in)
spin: trail ends after 12 steps
step 12, #processes: 3
          _p[0] = 6
proc  2 (receiver)        line 36 (state 21)
proc  1 (sender)  line 11 (state 6)
proc  0 (_init)   line 67 (state 4)
3 processes created
$
$ spin -t -r -s ABP1
...
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  47, Send err,0,0  -> queue 1 (out)
proc  1 (sender)   line  20, Recv err,1,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
<<<<>>>>
proc  1 (sender)   line  21, Recv nak,0,0  <- queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 -> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <- queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  -> queue 1 (out)
spin: trail ends after 226 steps
...
```

図 2. エラーの軌跡 - 拡張した Alternating Bit Protocol

ひとつの（成功を示す）acknowledgment がゆがめられて転送されると、sender と receiver は無限サイクルに陥ってしまいます。sender は acknowledgment を受け取っていないので頑固に最後に送ったメッセージを繰り返し送信します、また、receiver もまた頑固に否定応答（negative acknowledgment）を送り拒否し続けます。

さらに深く掘り下げる（Digging Deeper）

このマニュアルは *Spin* の主要な機能の概要と検証で使えるより一般的なやり方について説明しているにすぎません。*Spin* には、ここで取り上げたもの以外にも、検証の問題に立ち向かうのに役に立つであろうたくさんの機能があります。

Spin ではまた "タスク" や "要求" を *never*-claim でモデル化することでシステムがそれを満たすかどうか検証することもできます。ユーザがシステムが達成すべきタスクを形式化して claim で表現すると、*Spin* は素早くその claim が成り立つ事を証明するか、反例を示すでしょう。never-claim はBüchiオートマトン*5と等価で、線形時相論理をつかった制約をモデル化可能です。新しいバーションの *Spin* では、要求を定義するのに LTL の構文を使う事ができます。*Spin* はユーザが定義した LTL をそれと等価な never-claim に変換することができます。

Spin ではまた、ユーザが定義したサブセットに探索を制限するようにシステムの状態空間を縮小する事ができます（ここでも never-claim を使用しますが、今度は状態空間を切り取るために使用します）。この方法により、たとえ全状態空間探索が実行不可能でも、システムの振る舞いに与えたエラーのパターンが含まれるかどうかをすばやく検証することができます。

これら `Promela` と *Spin* の別の使用方法についての詳細については、[5] を参考にしてください。*Spin* バージョン 2.0 と 3.0 での拡張については WhatsNew.html を見てください。

参考文献（References）

[1] Dijkstra, E.W., ``Guarded commands, non-determinacy and formal derivation of programs." *CACM* 18, 8 (1975), 453-457.
[2] Dijkstra, E.W., ``Solution to a problem in concurrent programming control." *CACM* 8, 9 (1965), 569.
[3] Hoare, C.A.R., ``Communicating Sequential Processes." *CACM* 21, 8 (1978), 666-677.
[4] Holzmann, G.J., ``Algorithms for automated protocol verification." *AT&T Technical Journal* 69, 1 (Jan/Feb 1990). Special Issue on Protocol Testing, Specification, Verification.
[5] Holzmann, G.J., *The Spin Model Checker: Primer and Reference Manual* (2003), Addison-Wesley.
[6] Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*. 2nd ed. (1988), Prentice Hall.

---

*1 訳注 : 最近のバーション（確認したのは Version 4.2.9 -- 8 February 2007）では必ずしも `init` プロセスを宣言する必要はありません。以下のようにプロセス型の宣言の頭に active をつけることでプログラム起動時にプロセスのインスタンスを生成する事ができます。

```
byte state = 2;

active proctype A()
{       (state == 1) -> state = 3
}

active proctype B()
{       state = state - 1
}
```

*2 訳注 : Version 2 で `atomic` の意味が変更されたため、`atomic` シーケンスの中でブロックしてもランタイムエラーとはなりません。もし `atomic` シーケンスの中でプロセスがブロックした場合、他のプロセスのうちひとつを非決定的に選択し実行します。その後ブロックしていた文が実行可能になると、プロセスは残りのシーケンスを不可分に実行する事ができます。（ただし、他のプロセスも実行可能な場合は、どのプロセスが実行されるかは非決定的です）。この

atomic の変更に伴って、d_step が新たに追加されました。atomic と同様、d_step で囲まれたシーケンスを不可分に実行する事ができます。しかし次に示す3つの制約があります。1. シーケンスは完全に決定的であること、2. d_step の外に出て行く goto ジャンプを含まないこと、3. d_step の外から入ってくるラベルを含まないこと、です。また、d_step シーケンスの中でブロックした場合はランタイムエラーとなります。詳しくは、WhatsNew.html の 2.1.3 d_step および 2.3.1 Blocking Atomic Sequences に説明があります。

*3 訳注 : 原文では、"For instance, if by adding a label to the process type dijkstra(), from section 1.9:" となっていますが、Gerard J. Holzmann に確認したところ、これは間違いで本文中に 1.9 節はありません（原文も修正されました。2007.06.03）。

*4 訳注 : 最近のバージョン（確認したのは Version 4.2.9 -- 8 February 2007）では :: の直後にラベルを入れると以下のようにエラーになります。

```
$ spin -a ABP0
error: ("ABP0":13) label progress_0 placed incorrectly
=====>instead of
        do (or if)
        :: ...
        :: Label: statement
        od (of fi)
=====>always use
Label:  do (or if)
        :: ...
        :: statement
        od (or fi)
```

Gerard J. Holzmann に

```
:: skip -> progress_1: out!err(0,0)
```

というようにすればいいと教えてもらいました。メッセージを損失している行（14, 52行目）およびエラーを送信している行（13, 51行目）にプログレスラベルを挿入すると、本文中で説明している真に深刻なバグのエラーシーケンスが得られます。以下にプログレスラベルを挿入したモデルを示します。

```
 1  #define MAX       5
 2
 3    mtype = { mesg, ack, nak, err };
 4
 5    proctype sender(chan in, out)
 6    {       byte o, s, r;
 7
 8            o=MAX-1;
 9            do
10              :: o = (o+1)%MAX; /* next msg */
11    again:       if
12                   :: out!mesg(o,s) /* send */
13                   :: skip -> progress_0: out!err(0,0)  /* distort */
14                   :: skip -> progress_1: skip /* or lose */
15                fi;
```

```
16                  if
17                    :: timeout   -> goto again
18                    :: in?err(0,0) -> goto again
19                    :: in?nak(r,0) -> goto again
20                    :: in?ack(r,0) ->
21                       if
22                         :: (r == s) -> goto progress
23                         :: (r != s) -> goto again
24                       fi
25                  fi;
26      progress:       s = 1-s /* toggle seqno */
27              od
28      }
29
30      proctype receiver(chan in, out)
31      {       byte i;         /* actual input   */
32              byte s;         /* actual seqno   */
33              byte es;        /* expected seqno */
34              byte ei;        /* expected input */
35
36              do
37                :: in?mesg(i, s) ->
38                   if
39                     :: (s == es) ->
40                        assert(i == ei);
41      progress:         es = 1 - es;
42                        ei = (ei + 1)%MAX;
43                        if
44                          :: out!ack(s,0) /* send,   */
45                          :: out!err(0,0) /* distort */
46                          :: skip /* or lose */
47                        fi
48                     :: (s != es) ->
49                        if
50                          :: out!nak(s,0) /* send,   */
51                          :: skip -> progress_0: out!err(0,0) /* distort */
52                          :: skip -> progress_1: skip /* or lose */
53                        fi
54                   fi
55                :: in?err(0,0) ->
56                   out!nak(s,0)
57              od
58      }
59
60      init {
61              chan s_r = [1] of { mtype,byte,byte };
62              chan r_s = [1] of { mtype,byte,byte };
63              atomic {
64                      run sender(r_s, s_r);
65                      run receiver(s_r, r_s)
66              }
67      }
```

*5 訳注 : Büchiは"ビューヒ"とか"ビュッヒ"と発音します。

Spin Online References
Promela Manual Index

(Page Updated: 30 August 2014)

Promela Grammar
Spin HomePage
