

設計モデル検証(基礎編)講座 (第1回)

第2版

2010年6月7日

トップエスイープロジェクト



設計モデル検証（基礎編）のゴール

- 検証とは何かを把握できる
 - 検証には何が必要なのかを知っている
 - 検証を行うツールとしてSPINを使える
 - 設計モデルの正しさを検証することが出来る
 - 前提・条件・範囲を明確しつつ設計を具象化し、その検証ができる
 - 何を検証し、どのような性質が保証できたのかが把握できる
- ➔ 対象システムの設計の意図を明確化し、そこに成り立つ性質・特性を正しく把握できる人材



講義計画

- 第1回: 導入: 並行分散システムの設計の難しさ, SPIN入門(1)
- 第2-4回: 基礎 - SPIN入門 (2)-(4) 担当: 田辺

- 第5-9回: 設計モデルの検証 (1)-(5)
- 第10-11回: 評価ボードによる演習 (1)-(2) 吉岡先生
- 第12-14回: 応用演習 (1)-(3) 田原先生
- 第15回: 発表・議論とまとめ



本日の内容

- 導入: 本講座の背景
 - 検証とは?モデル検査とは?テストとの違い
 - なぜ検証が必要か? (本講座により解決を目指す課題)
- SPIN入門 (1)
 - Spinによるモデル検査
 - Xspinの使い方
 - Promela の基礎
 - インターリーブ意味論
 - プロセス
 - 代入文, if文, do文, d_step文

導入：本講座の背景



検証とは?

- 真偽を確かめること。事実を確認・証明すること。(大辞林第二版)
 - ソフトウェアの検証: ソフトウェアに関する事実の確認・証明
- 英語ではverification
 - Software verification is a broad and complex discipline of software engineering whose goal is to assure that a software fully satisfies all the expected requirements.
(Wikipedia英語版)
「ソフトウェアが、**全ての期待される要求を完全に満たすこと**の確認」



検証とは?

■ 2種類の検証: 動的・静的

■ 動的検証: ソフトウェアを動作させて検証

■ 例: テスト

■ 静的検証: ソフトウェアを動作させずに検証

■ 例: プログラム解析、**形式的検証**

■ **モデル検査**は、形式的検証手法の1つ

■ 形式的検証: **数学的手段**により、**形式的に記述された**ソフトウェアの**仕様**を厳密に検証すること

■ モデル検査: ソフトウェアの**振舞いの仕様**に対し、可能な実行パターンを**網羅的**かつ**自動的**に検査することにより検証を行う手法



モデル検査とテストとの違い

項目	モデル検査	テスト
動的・静的	静的	動的
振舞いの網羅性	網羅的	部分的
検証の厳密性	厳密	不確実
利用容易性	難	易
検証可能なソフトウェアの規模	小規模	大規模
利用実績	少	多

青字は優位な項目を示す



なぜ検証が必要か？ —システムの高度分散化—

【分散化】: 従来単一システムにまとめられていた個別の機能としてノード化され、ネットワークで接続・連携

- 各機能が並行動作→並行プログラムは、逐次プログラムに比べて、以下の要因により、開発・運用・保守が困難
 - 非決定性
 - 資源共有・相互排除
 - プロセス間相互作用

→ 振る舞いの可能性が組み合わせ的に増大

【家電製品、ネットワーク接続、オープン化】

→ さまざまな例外を考慮して設計する必要がある



なぜ検証が必要か？ —システムの高度分散化—

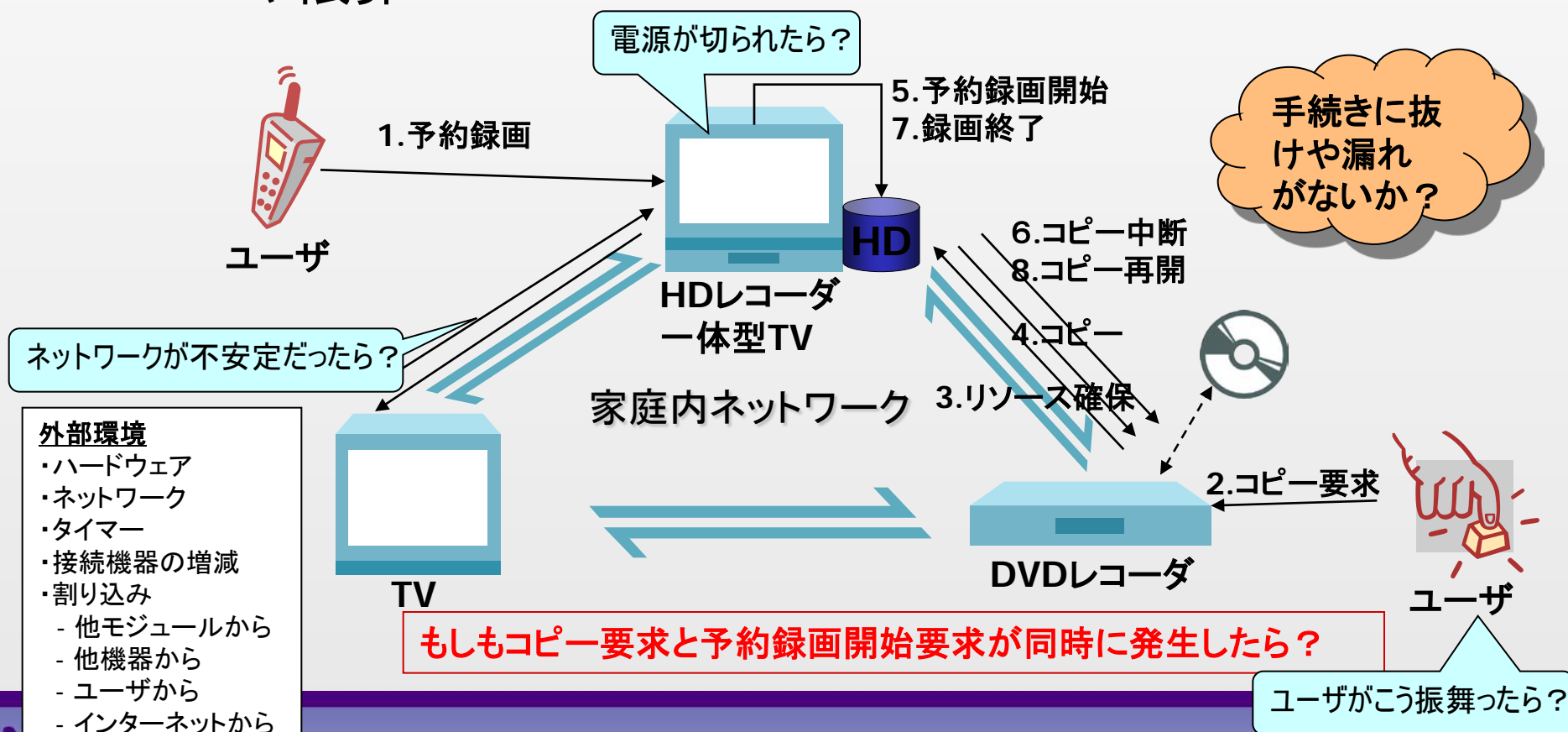
- 大規模化: **ノード数・リンク数**が爆発的に**増大**
 - 異種混合化: ノードやリンクの**種類数**も増大
 - ノード: **ハードウェア**は大型計算機から携帯電話まで、**ソフトウェア**は大企業・多企業間システムから機器組込みまで
 - リンク: インターネット、有線・無線LAN、家電機器間の特殊な接続方式など
 - オープン化: **さまざまな管理形態**のノード・リンクが**頻繁に**追加・退出
 - データ流量やアクセス数の**予測困難性**
 - 悪意のあるノードによる**セキュリティリスク**
 - **ネットワーク家電**: 高度分散化したシステムの検証の困難さが端的に現れる例
- **さまざまな例外・ユースケースを考慮して設計**する必要がある

ネットワーク家電における検証の必要性



並行分散システムにおける検証

- 多種多様な**外部環境を考慮**しないといけない
- **様々な動作パターン**に対し、人海戦術によるテストはもはや限界





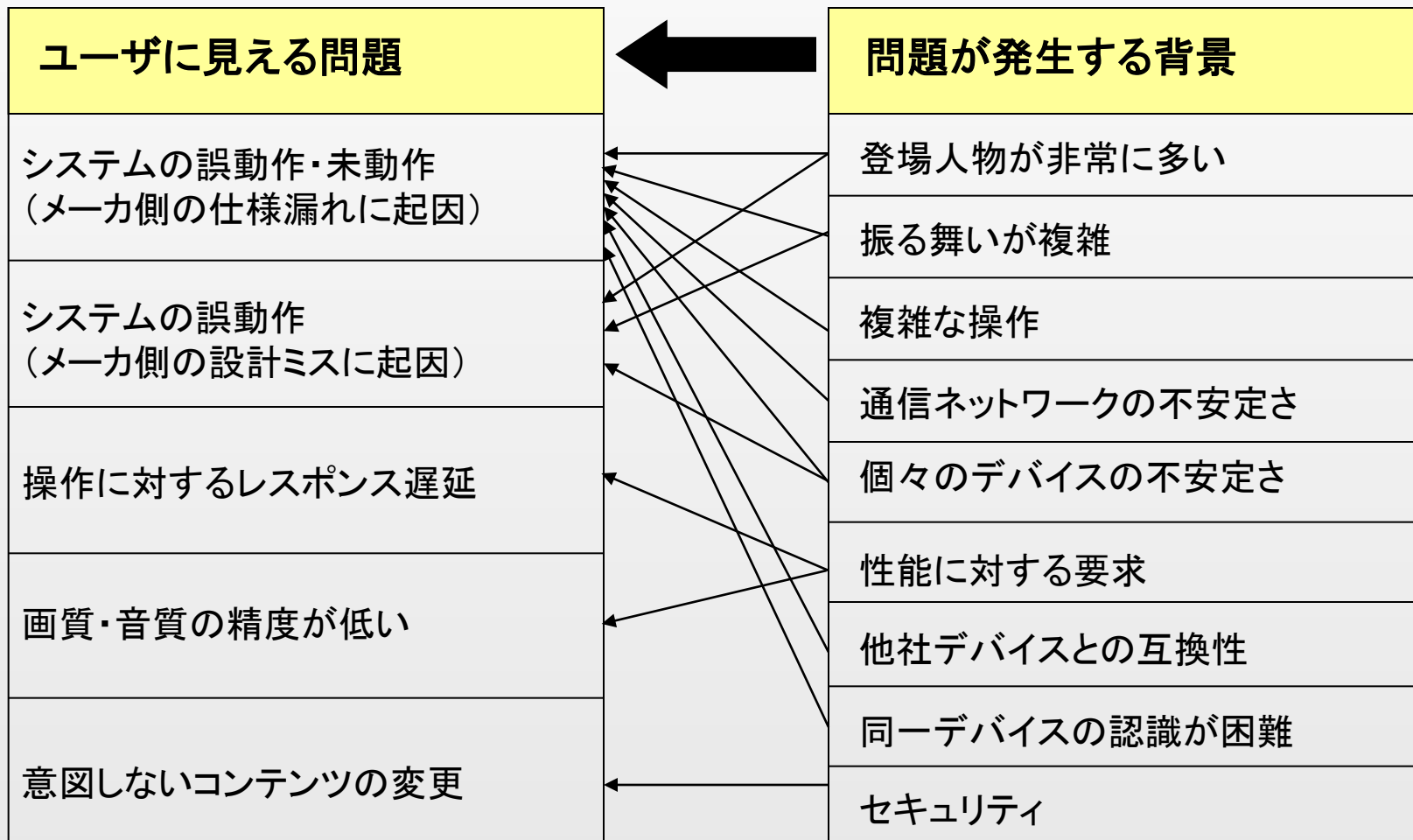
テストの限界

項目	モデル検査	テスト
動的・静的	静的	動的
振舞いの網羅性	網羅的	部分的
検証の厳密性	厳密	不確実
利用容易性	難	易
検証可能なソフトウェアの規模	小規模	大規模
利用実績	少	多

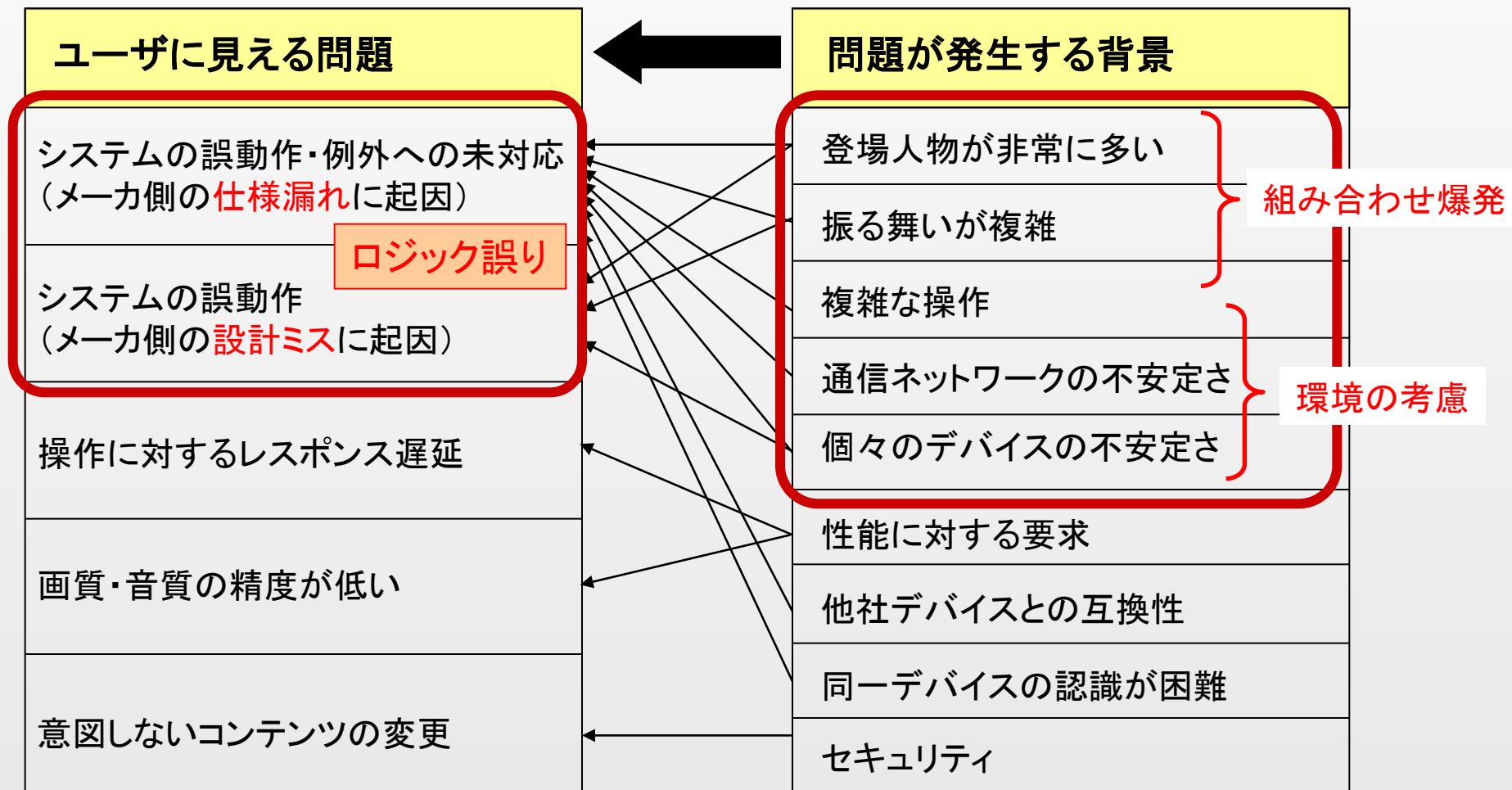
この部分の短所が致命的になりつつある



ネットワーク家電の問題



モデル検査が解決してくれる問題



SPIN入門 (1)



SPINとは

■ Simple Promela Interpreter

- Promela言語でシステムの振舞いの仕様を記述
- 各種検証: シミュレーション、デッドロック検証、LTL式検証など

■ Promela: Protocol/Process Meta Language

- チャネル通信オートマトンを記述するための言語。
 - プロセス: 並行動作する単位。
 - チャネル: プロセス間でメッセージを送受信する。



リソース

- 吉岡ほか, “SPIN による設計モデル検証”, 近代科学社, 2008.
- G. J. Holzmann, “The SPIN Model Checker: Primer and Reference Manual,” Pearson Educational, 2003.
- 中島, “SPIN モデル検査”, 近代科学社, 2008.
- <http://spinroot.com/spin/> -- Spinサポートページ
 - プログラムダウンロード（本講義では, 4.3.0を推奨）
 - オンラインリファレンス

GUI (Xspin)

■ Xspin

- Spinを使った検証のためのGUI。
- Promela記述の編集、シミュレーション・モデル検査などの実行を統一的に取り扱うことができる。

■ Xspinで便利なこと。

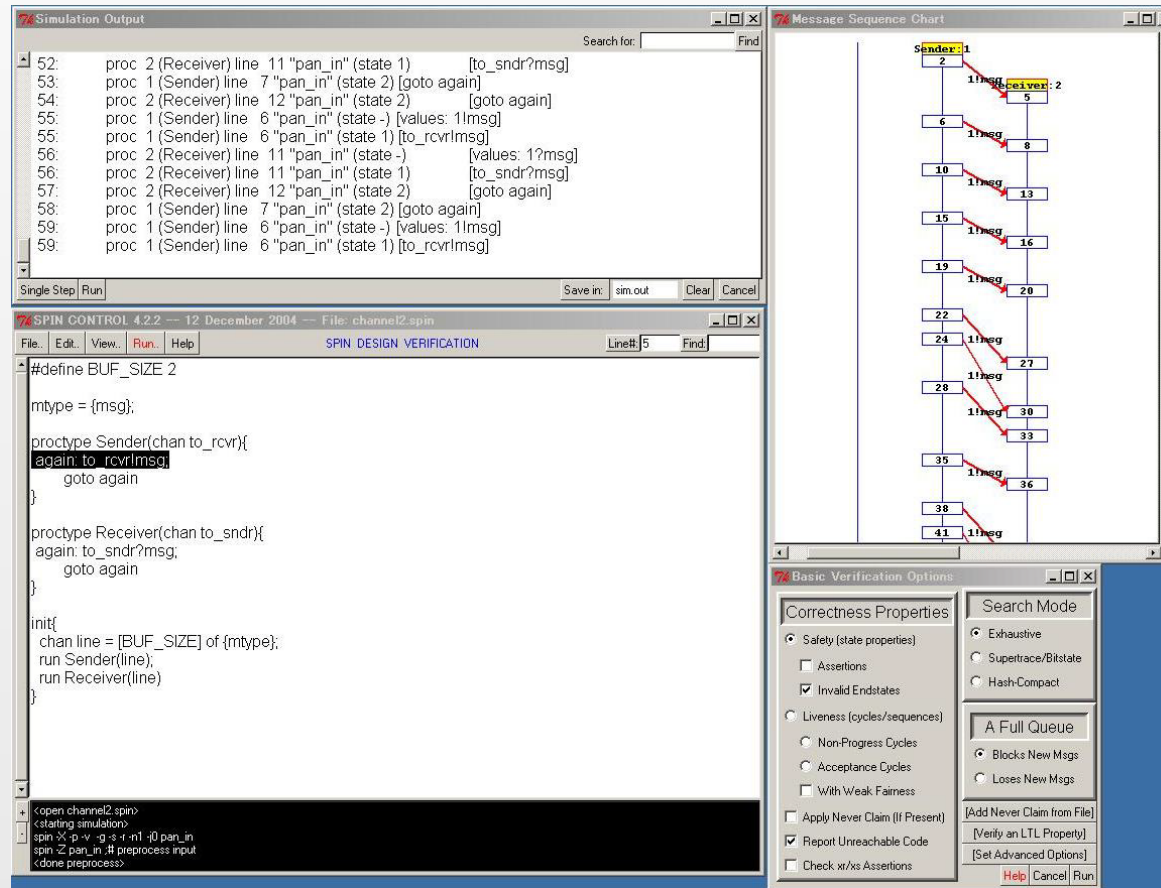
- シミュレーション実行時の変数値の追跡。
- メッセージ通信の視覚的表現。
- LTL式によるモデル検査(コピーペーストが不要)。
- モデル検査→反例解析プロセスの支援。



準備

- ドライブUに適切なフォルダを作る.
- LMS の本講座「第1回講義資料」フォルダから, p1.zip をダウンロードし, 上記のフォルダに展開する.
- (好みに応じて) スタートメニュー内にある xspin ショートカットを, デスクトップなど適切な場所にコピーし, 右クリック→プロパティ にある「作業フォルダ」に, 上述のフォルダを指定する.

Xspin



The screenshot displays the Xspin software interface, which is used for simulating and verifying Spin programs. It consists of several windows:

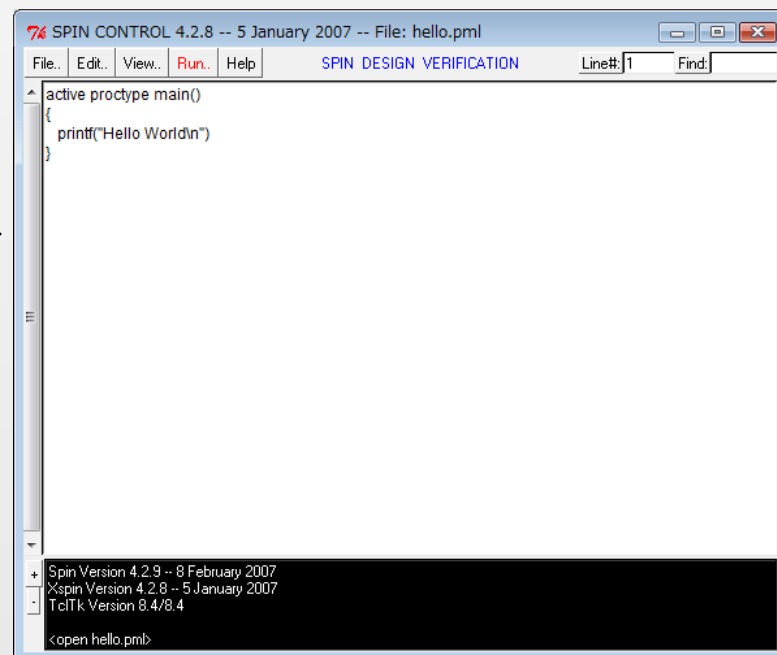
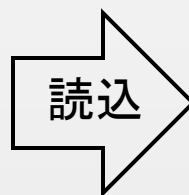
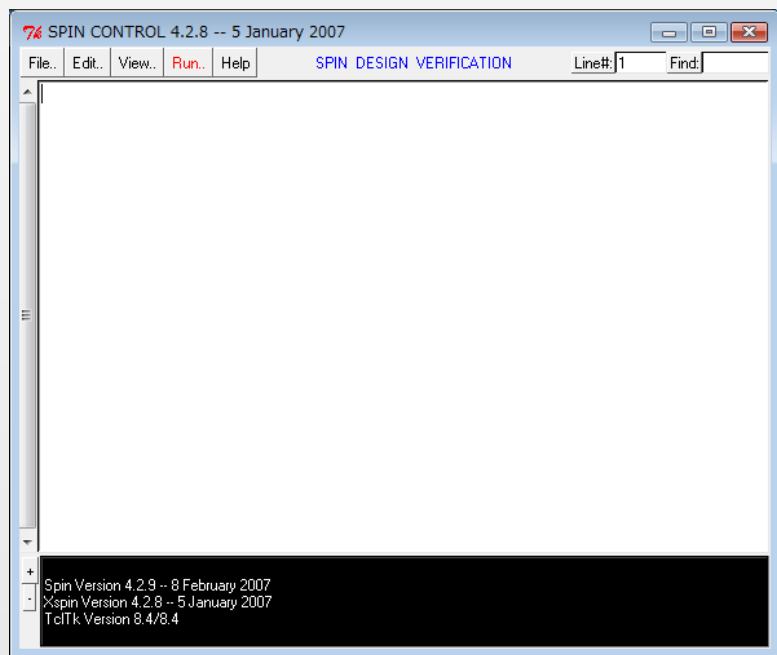
- Simulation Output:** Shows the execution log of the simulation. It includes line numbers and the state of the Sender and Receiver processes. For example, it shows the Sender sending a message to the Receiver, which then sends it back.
- SPIN CONTROL 4.2.2:** The main window for running the simulation. It includes a menu bar (File, Edit, View, Run, Help) and a status bar showing the current line number (5) and the file name (channel2.spin).
- Source Code:** Displays the Spin program code. The code defines a buffer size, a message type, and two processes: Sender and Receiver. The Sender process sends a message to the Receiver, which then sends it back.
- Message Sequence Chart:** A visual representation of the message exchange between the Sender and Receiver. It shows the sequence of messages and the state of the processes at each step.
- Basic Verification Options:** A panel for configuring the verification process. It includes options for correctness properties (Safety, Liveness), search mode (Exhaustive, Supertrace/Bitstate, Hash-Compact), and various flags for reporting and checking assertions.

基本的な使い方

- Xspinを起動
- Promela記述を別のエディタで編集・保存
 - Xspinのエディタは、環境によっては動作が不安定なことがあるため
- XspinでPromela記述を読込み/再読込み
- 構文チェック
- オプション設定
- 実行（シミュレーション/モデル検査）

基本的な使い方

- Xspinを起動。
- hello1.pml を読み込み

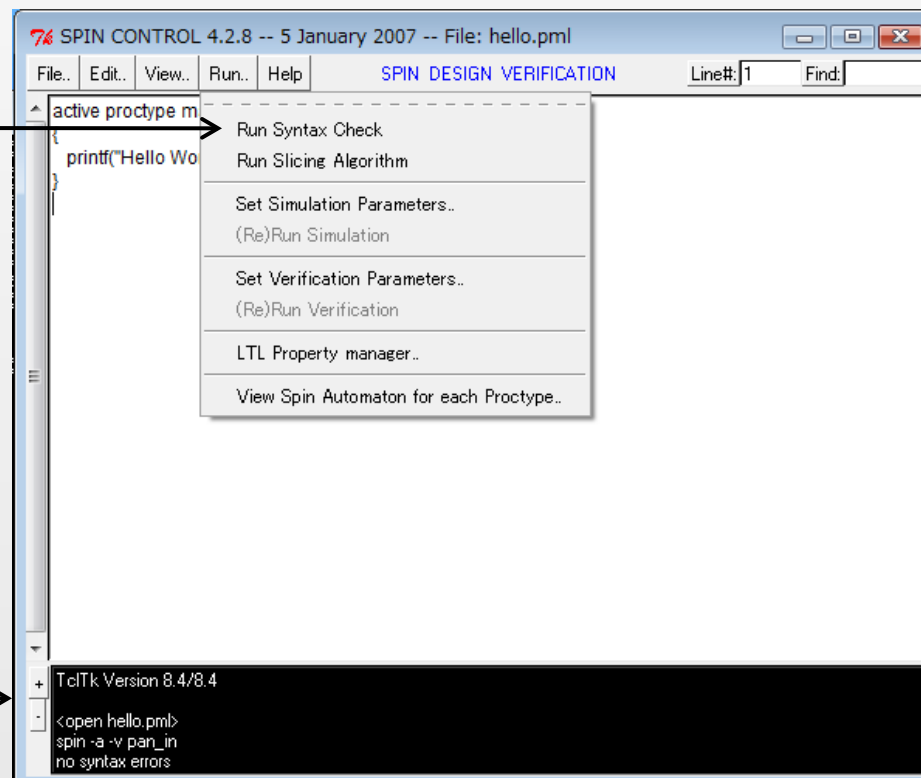


基本的な使い方

■ 構文チェック。

構文をチェックする。
実行はしない。

エラーなし
(エラーがあれば報告される)

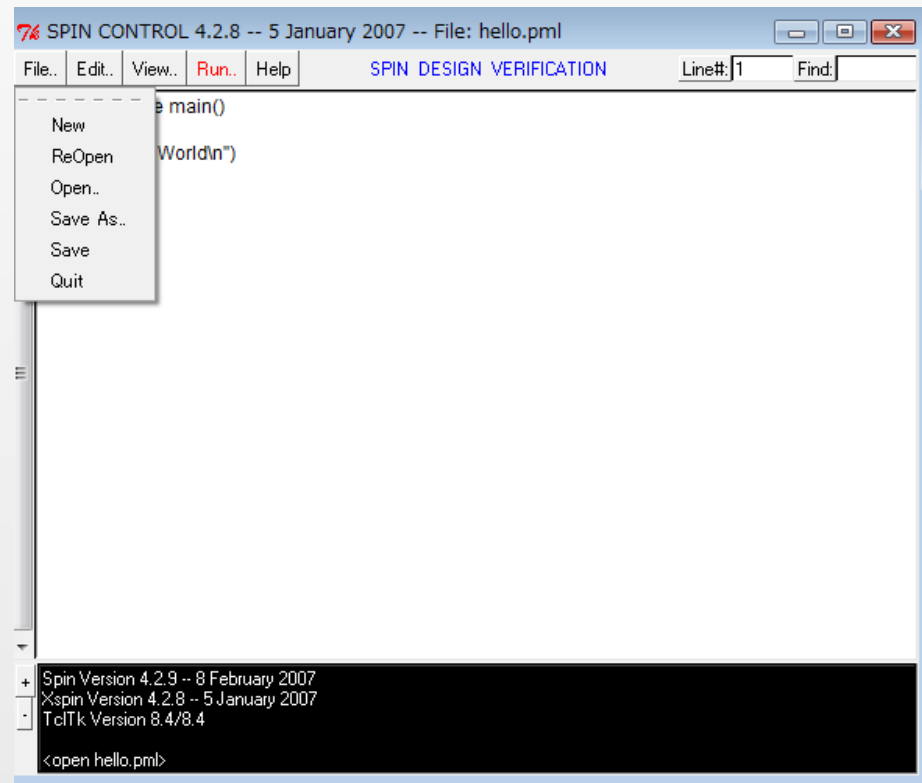


基本的な使い方

■ ファイルメニュー。

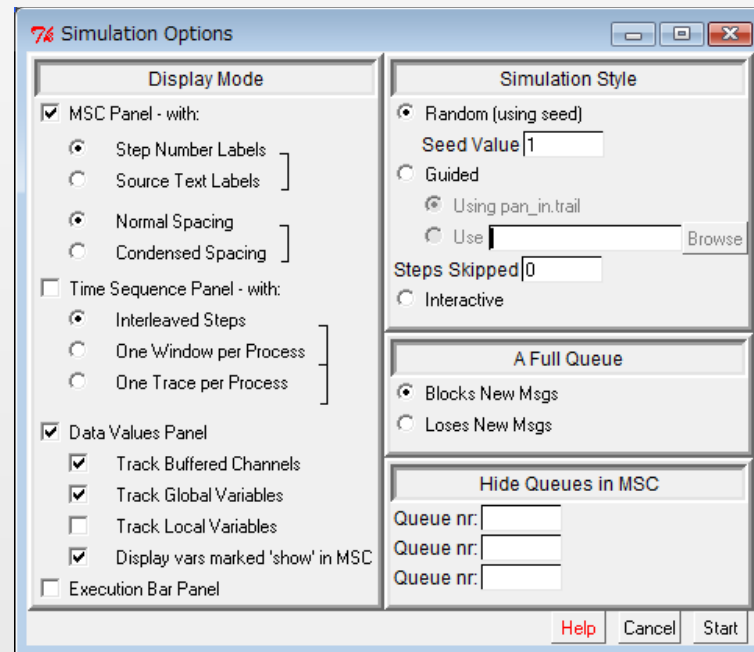
- 保存、読込み、再読込み、初期化、終了。

- 別のエディタで行った編集を反映させるには、「Reopen」を選ぶ。



シミュレーション実行

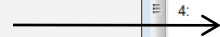
- Xspin上でシミュレーション実行する。
 - Run → Set Simulation Parameters
 - Start



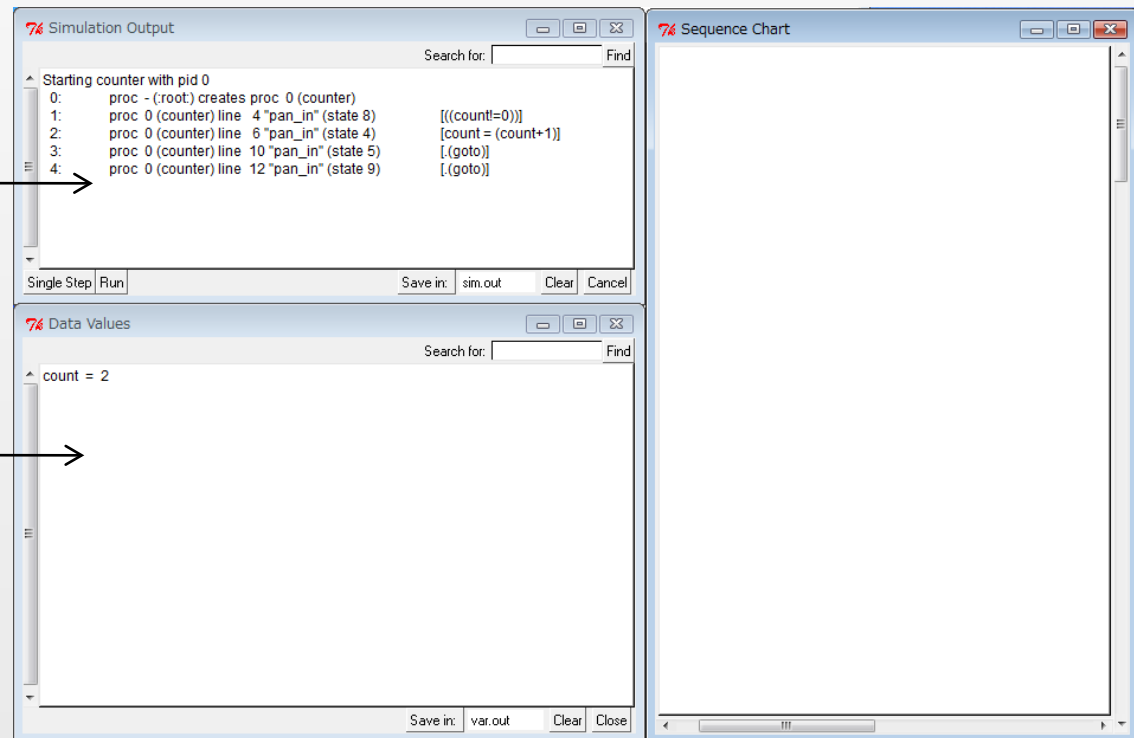
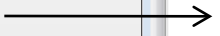
シミュレーション実行

■ Single Step または Run

実行の様子を表示

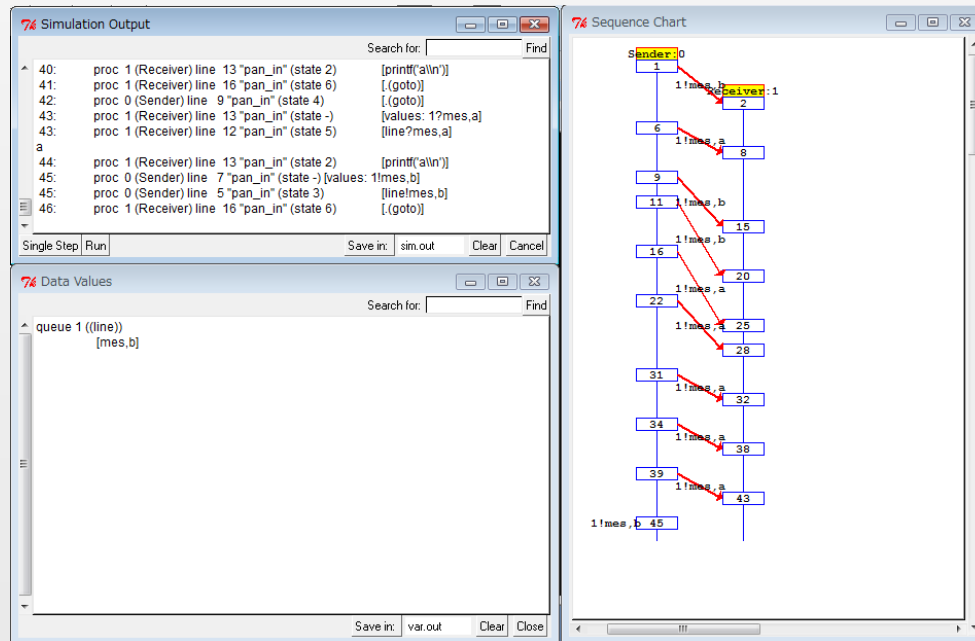


変数の値を表示



シミュレーション実行

- チャンネル通信がある場合は、通信の様子をシーケンスチャートで表示。

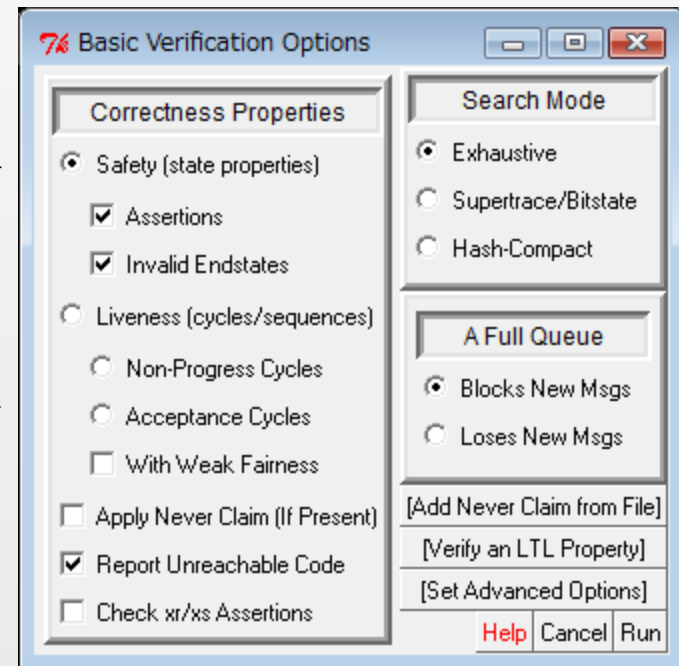


モデル検査

- Run→Set Verification Parameters
- オプションを指定
- Run

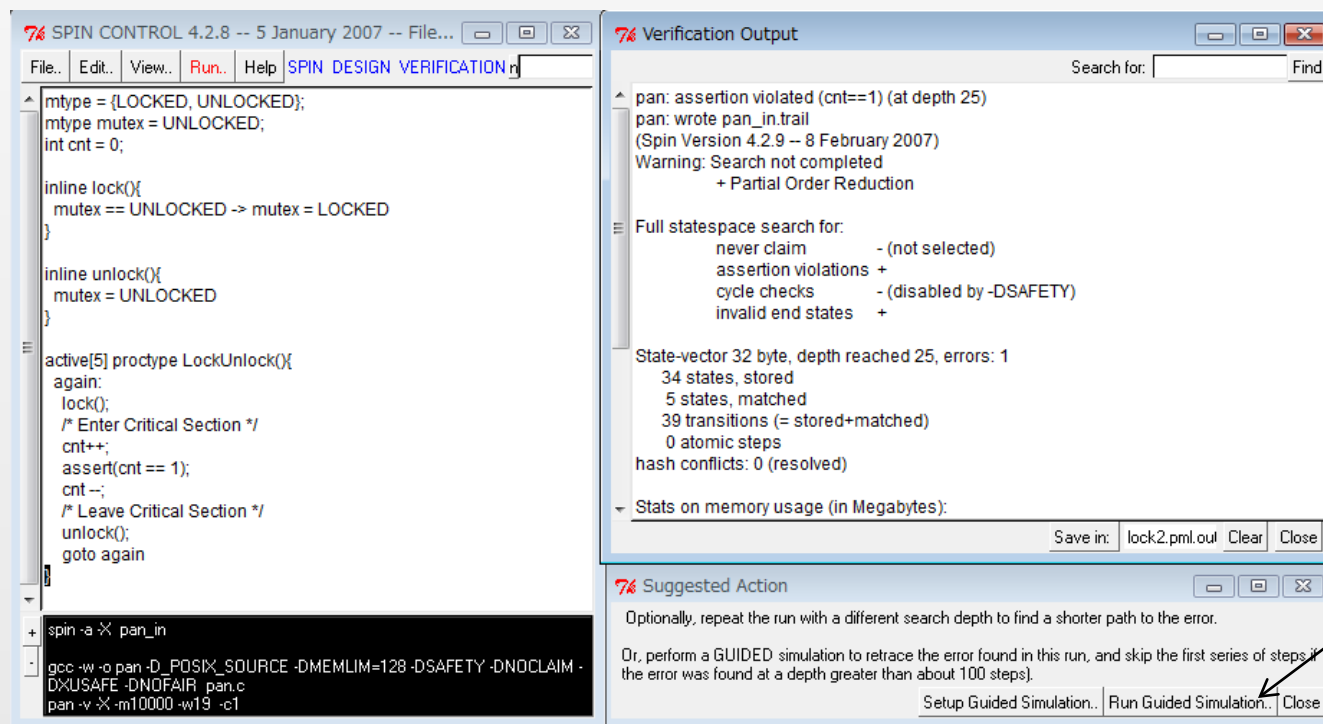
表明とデッドロックの検証 →

進行性の検証 →



モデル検査

- コンパイルして実行してくれる。
- エラーがあると報告される。



The screenshot shows the SPIN CONTROL 4.2.8 interface. The left pane displays a C code snippet for a mutex implementation. The right pane shows the 'Verification Output' window, which reports an assertion violation. Below the output, a 'Suggested Action' section provides instructions on how to further investigate the error.

```
mtype = {LOCKED, UNLOCKED};
mtype mutex = UNLOCKED;
int cnt = 0;

inline lock(){
    mutex == UNLOCKED -> mutex = LOCKED
}

inline unlock(){
    mutex = UNLOCKED
}

active[5] proctype LockUnlock(){
    again:
    lock();
    /* Enter Critical Section */
    cnt++;
    assert(cnt == 1);
    cnt--;
    /* Leave Critical Section */
    unlock();
    goto again
}
```

spin -a -X pan_in

gcc -w -o pan -D_POSIX_SOURCE -DMEMLIM=128 -DSAFETY -DNOCLAIM -DXUSAFE -DNOFAIR pan.c
pan -v -X -m10000 -w19 -c1

Verification Output

pan: assertion violated (cnt==1) (at depth 25)
pan: wrote pan_in.trail
(Spin Version 4.2.9 -- 8 February 2007)
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 32 byte, depth reached 25, errors: 1
34 states, stored
5 states, matched
39 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):

Save in: lock2.pml.out Clear Close

Suggested Action

Optionally, repeat the run with a different search depth to find a shorter path to the error.

Or, perform a GUIDED simulation to retrace the error found in this run, and skip the first series of steps if the error was found at a depth greater than about 100 steps.

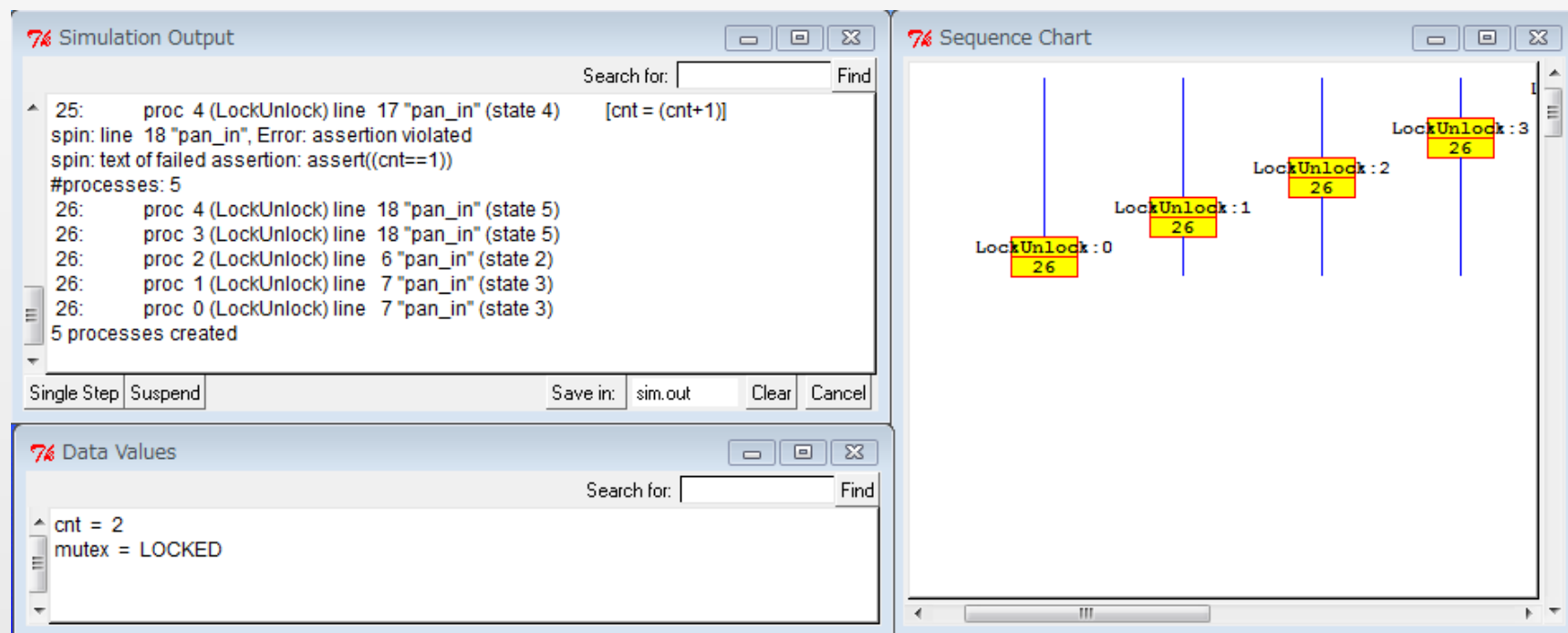
Setup Guided Simulation.. Run Guided Simulation.. Close

エラーを報告

反例に基づいた
シミュレーション
実行

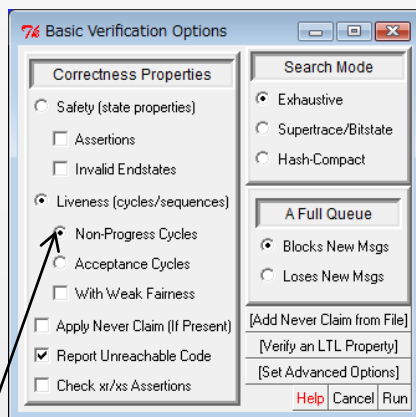
モデル検査

- 反例に基づいて実行される。

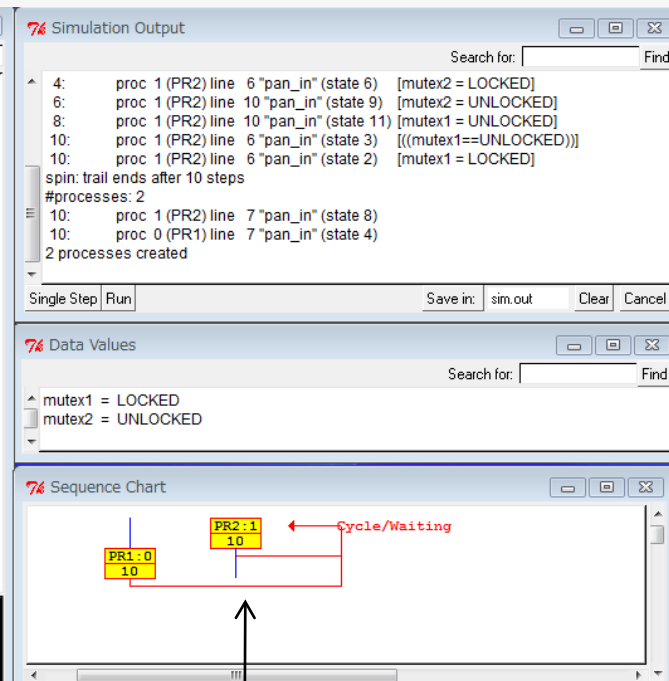
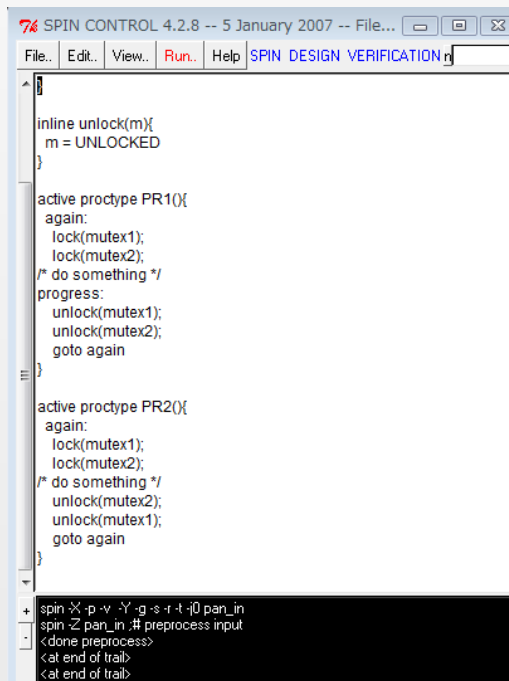


モデル検査

■ 進行性の検証の場合。



Non-Progress Cycle
(進行しないサイクルの
検出)



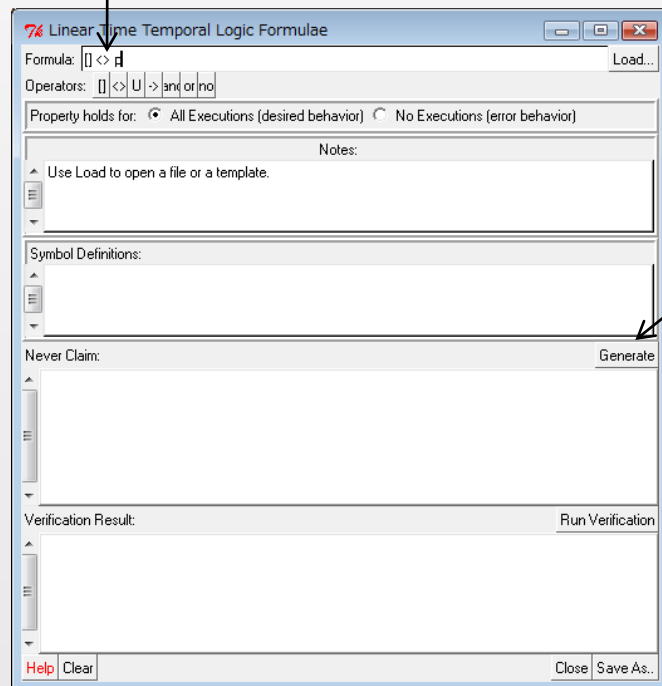
反例としてサイクルを出力

LTL式の検証

■ Run→LTL Property Manager

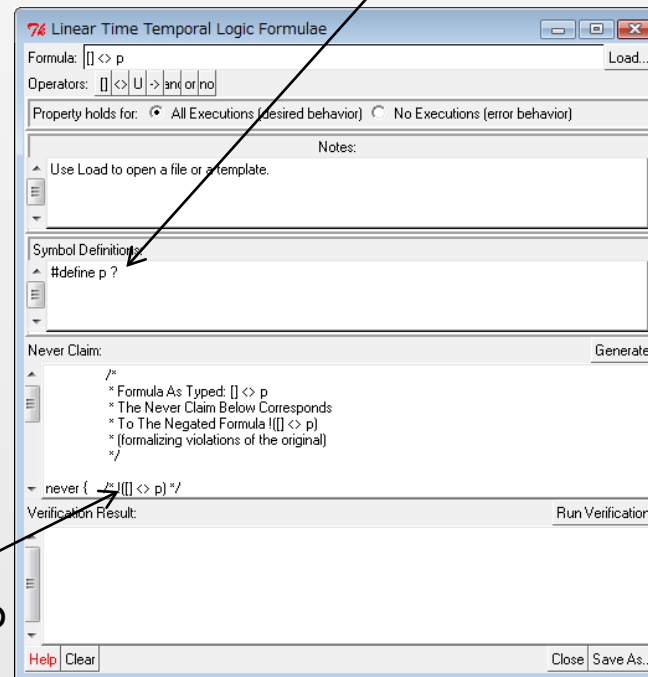
LTL式を入力
(成立してほしい式。否定をとらなくてもよい)

命題変数pを定義する。



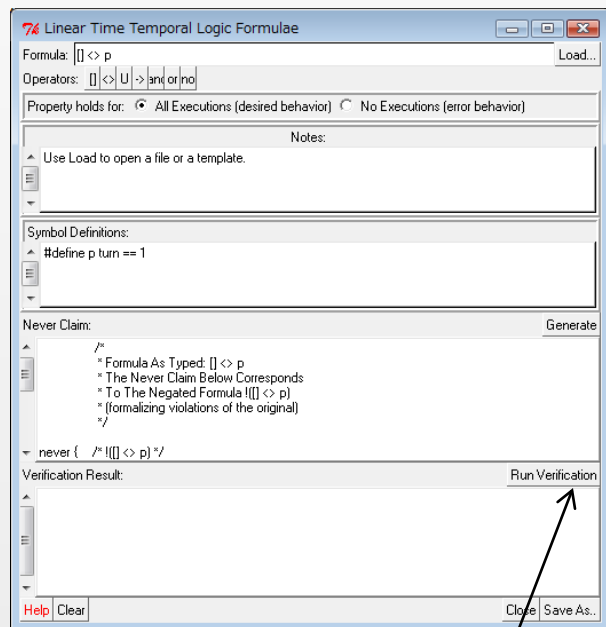
never claim
を生成

否定になっ
ている

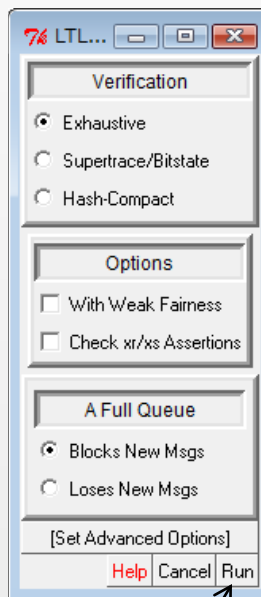


LTL式の検証

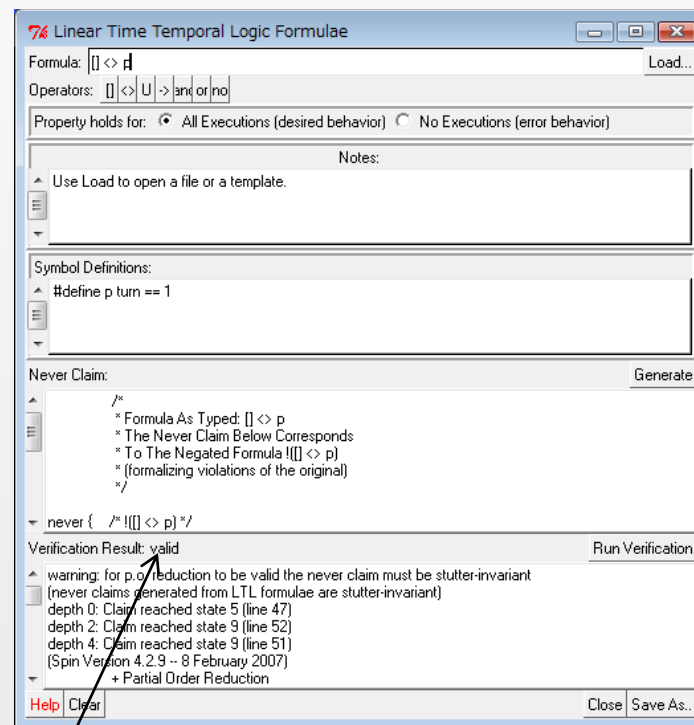
オプションの指定



検証の実行

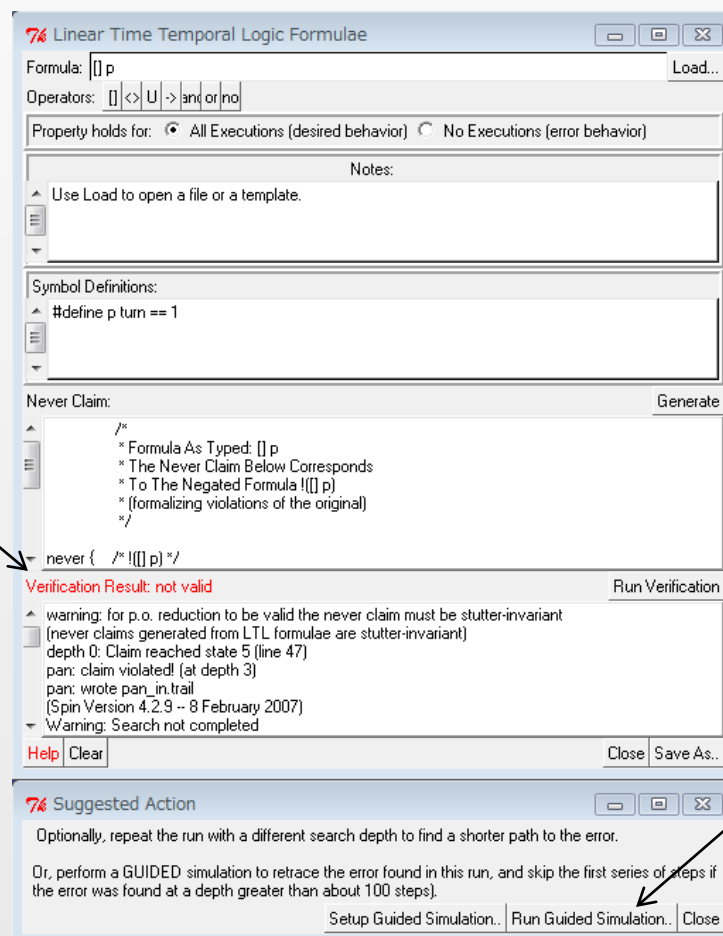


実行



LTL式は成立する。

LTL式の検証



LTL式が成立しない。

反例に基づいて実行

小さなPromelaコード例: small1.pml

```
int x;  
  
proctype Proc1() {  
    ... (略) ...  
    x = 1;  
    x = x + x;  
    assert(x == 2);  
}  
  
proctype Proc2() {  
    x = 0;  
}  
  
init {  
    run Proc1();  
    run Proc2();  
}
```

プロセス (動作単位)

- 定義
- 実行

init プロセスが, 最初に実行を開始する.

小さなProme|aコード例: small1.pml

```
int x;

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

proctype Proc2() {
  x = 0;
}

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

大域変数. 0に初期化される.

代入文.
C や Java に似ている.

assert 文 (表明)
「x == 2が成り立っているはず」
と主張している.



インターリービング意味論

- プロセスのうちで実行可能なものが、**非決定的に** 選択され、実行される.
- **シミュレーション**の場合: 各時点で1通りの選択のみがランダムに行われる.
- **モデル検査**の場合: 各時点で可能な選択をすべて行う.

可能な実行系列

```

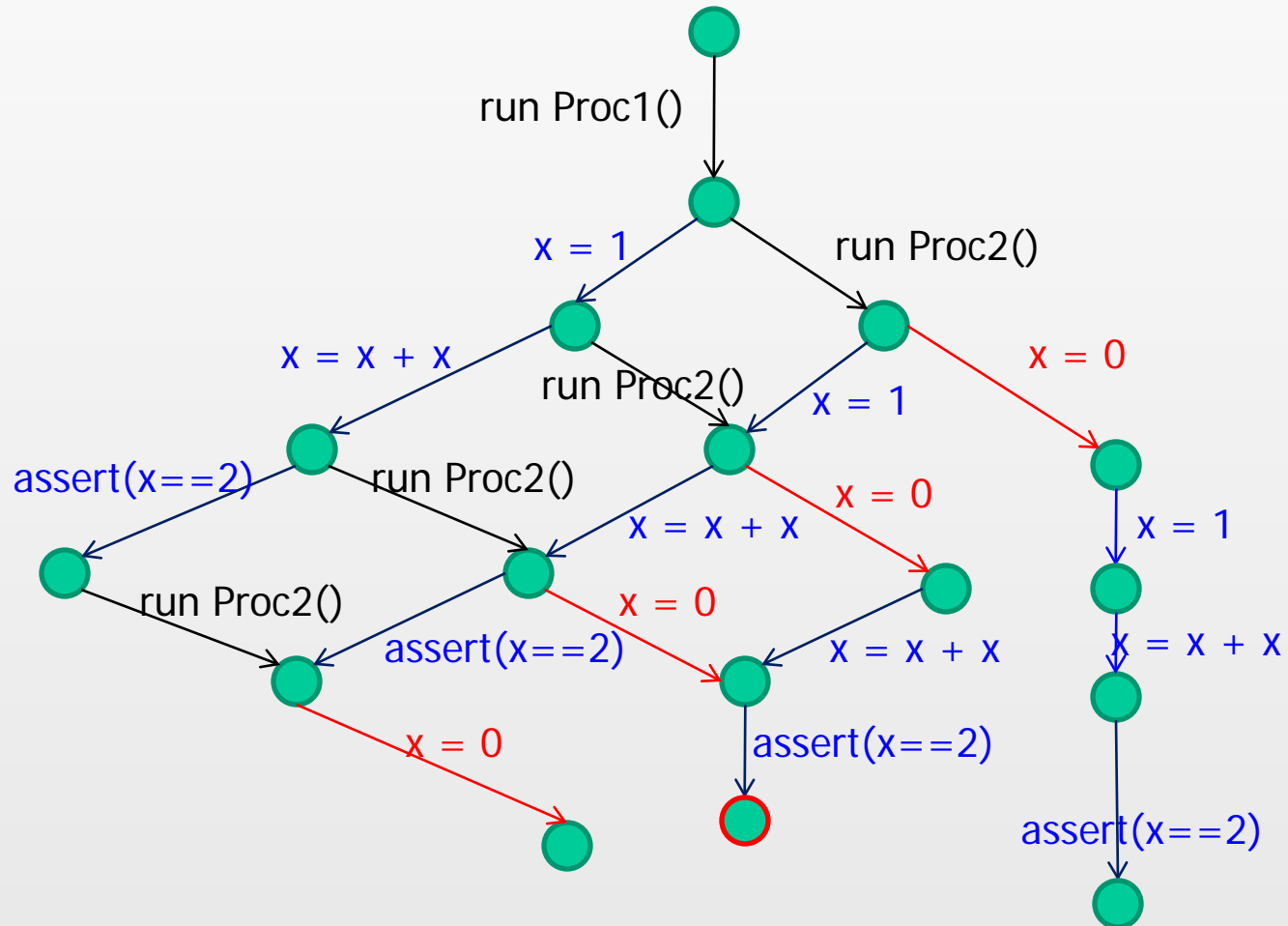
int x;

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

proctype Proc2() {
  x = 0;
}

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

```





練習

- small1.pml を用いて, シミュレーションを行え. オプション画面で, Seed Valueを変えながら複数回実行せよ. エラーは報告されるか?
- small1.pml を用いて, モデル検査を行え. エラーは報告されるか?

プロセス

```
int x;  
  
active proctype Proc1() {  
    x = 1;  
    x = x + x;  
    assert(x == 2);  
}  
  
active proctype Proc2() {  
    x = 0;  
}
```

- proctype の前にキーワード active を記述すると, そのプロセスは, 起動時から実行可能になる.

プロセス

```
byte y;  
  
proctype foo(byte x) {  
    y = x;  
}  
  
init {  
    run foo(3);  
    run foo(50);  
}
```

- 引数付きのプロセスも定義できる.

byte ... 8bitの整数型



文の実行可能性

- 文は、**実行可能**であるか、**ブロック**しているかのいずれか.
- 代入文は、常に実行可能.
- run 文も、常に実行可能. (本当はちょっと違うが、気にしない.)
- 式も文になる. 値が非0なら実行可能. 0ならブロック.

例 (expblock1.pml)

```
int x, y, z;  
  
active proctype Proc1() {  
    x = 2; y = 2;  
  
    x = x + z;  
    assert(x > y);  
}  
  
active proctype Proc2() {  
    z = 1;  
  
}
```

assert は失敗しうる.

例 (expblock2.pml)

```
int signal;  
int x, y, z;  
  
active proctype Proc1() {  
    x = 2; y = 2;  
    signal;  
    x = x + z;  
    assert(x > y);  
}  
  
active proctype Proc2() {  
    z = 1;  
    signal = 1;  
}
```

assert は失敗しない.

値について

- Promelaでは整数の変数を使うことができる。
 - bit: 0,1
 - byte: 0~255
 - short: $-2^{15} \sim 2^{15} - 1$
 - int: $-2^{31} \sim 2^{31} - 1$
- 配列
 - 「bit a[5]」、「bit a[5] = 0」、「byte hoge[2]」、....
 - a[3] = 0, hoge[1] = 10, ...
- レコード型
 - 「typedef Point{ int x; int y; }」
 - Point p; p.x = 0;
- 演算子
 - +, -, *, / : 足し算、引き算、掛け算、割り算
 - % : 余り
 - <, >, <=, >= : 比較
 - &&, ||, !, ==, != : 論理積、論理和、否定、等しい、等しくない

if文

```
if
::  choice1 -> stat1,1; stat1,2; ...
::  choice2 -> stat2,1; stat2,2; ...
::  ...
::  choicen -> statn,1; statn,2; ...
fi
```

- choice_iたち（ガードと呼ばれる）のうち1つ以上が実行可能ならば、このif文は実行可能. その場合、実行可能なガードの1つが非決定的に選択されて、実行される. その後は対応する stat_{i,1} 以下に制御が移る. (他のプロセスとインターリーブする)
- すべてのchoice_iがブロックの場合には、このif文はブロック.
- 「->」は、「;」と同じ意味である.

```
if
:: x > 0 -> x--;
:: y > 0 -> y--;
:: else -> skip;
fi
```

- ガードに **else** が指定できる.
これは, 他のすべてのガード
がブロックするときに限って
実行可能.

```
if
:: skip -> stat1;
:: skip -> stat2;
:: skip -> stat3;
fi
```

- 非決定的に stat1, stat2,
stat3 を選択するイディオム

```
if
:: stat1;
:: stat2;
:: stat3;
fi
```

- 1つ上のコードとどう違う?

do文

```
do
::  choice1 -> stat1,1; stat1,2; ...
::  choice2 -> stat2,1; stat2,2; ...
::  ...
::  choicen -> statn,1; statn,2; ...
od
```

- if文と同様に動作する. 異なるのは, 選択された枝の最後まで実行が終了したら, 再び選択が行われる (ループする) こと.
- do文の中には, **break** 文を置くことができる. これは, 常に実行可能で, 実行されると, ループを抜ける.

間違い探し (doFault.pml)

```
byte a[10];
byte i;
byte count;
.... /* a[i] の値が設定される */
do
::  if
    :: i == 10 -> break
    :: a[i] == 0 -> count = count + 1
    fi;
    i++
od
```

- $0 \leq i < 10$ かつ $a[i] == 0$ となる i の個数を数えようと思っているのだが, 期待通りには動作しない. (なぜか?)
- どう修正すればよいか?

実習: a11Num. pml

- しばらくは, Spinを使わずに机上で検討する.
- 何をしているコードか? アサーションは成立するか?
- #define N 2 にするとどうか?
- #define N 3 にするとどうか?

```
#define N 1
byte x, t1, t2;

proctype Add() {
    do
        :: t1 = x; t2 = x; x = t1 + t2
    od
}

init { x = 1; run Add(); run Add(); assert( x != N ); }
```

N = 1 の場合

x	t1	t2	proc0	proc1	proc2
0	0	0			
1			x = 1		
			run Add()		
			run Add()		
			assert(x != 1)		

N = 2 の場合

x	t1	t2	proc0	proc1	proc2
0	0	0			
1			x = 1		
			run Add()		
			run Add()		
	1			t1 = x	
		1		t2 = x	
2				x = t1 + t2	
			assert(x != 2)		

N = 3 の場合

x	t1	t2	proc0	proc1	proc2
0	0	0			
1			x = 1		
			run Add()		
			run Add()		
	1			t1 = x	
	1				t1 = x
		1			t2 = x
2					x = t1 + t2
		2		t2 = x	
3				x = t1 + t2	
			assert(x != 3)		



反例のチェック

- 確かに反例は得られるのだが、
信じられないほど長い反例である.



Spin の探索方式

- Spin は, assertion などの安全性 (safety property) の検証 では, **深さ優先探索** (DFS = depth first search) を行う.
 - 一般のLTL論理式に関する検証では, **二重深さ優先探索** (double DFS) を行う.
- はじめて見つかる反例パスを報告するので, 比較的長い反例が見つかりがち.

深さを限定した探索： 実行手順

- 探索する深さを限定するオプションがある.
 - ただし, 小さくしすぎると, 反例が見つからなくなる.
- Run → Set Verification Parameter → Set Advanced Options で, Maximum Search Depth を設定する.
- $N = 3$ について, いろいろな値を設定して, Spin からの報告がどのように変わるか実験せよ.

深さを限定した探索： 実行手順

- Maximum Search Depth の最適な値を見つけるのは、手間がかかる。Spinに探させるオプションもある。
- **-i** : 最短の反例を探す。時間がかかることがある。
Extra Runtime Options に指定する。
- **-I** : **-i**よりは効率良く短い反例を探す。最短の反例が見つからないことがある。Extra Runtime Options に指定する。
- **-DREACH** : **-i** や **-I** を指定するときには、これも指定する必要がある。Extra Compile-Time Directives に指定する。
- これらのオプションを使用して、探索を行え。他のNに対しても探索を試みよ。