SEAMAIL

Newsletter from Software Engineers Association

Vol. 14, Number 8 March, 2006

目　次

ソフトウェア技術者協会
Software Engineers Asociation

ソフトウェア技術者協会 (SEA) は，ソフトウェアハウス，コンピュータメーカ，計算センタ，エンドユーザ，大学，研究所など，それぞれ異なった環境に置かれているソフトウェア技術者または研究者が，そうした社会組織の壁を越えて，各自の経験や技術を自由に交流しあうための「場」として，1985年 12月に設立されました．

その主な活動は，機関誌 SEAMAIL の発行，支部および研究分科会の運営，セミナー／ワークショップ／シンポジウムなどのイベントの開催，および内外の関係諸団体との交流です．発足当初約 200人にすぎなかった会員数もその後増加し，現在，北は北海道から南は沖縄まで，350 余名を越えるメンバーを擁するにいたりました．法人賛助会員も 17社を数えます．支部は，東京以外に，関西，横浜，名古屋，九州，広島，東北の各地区で設立されており，その他の地域でも設立準備をしています．分科会は，東京，関西，名古屋で，それぞれいくつかが活動しており，その他の支部でも，月例会やフォーラムが定期的に開催されています．

「現在のソフトウェア界における最大の課題は，技術移転の促進である」といわれています．これまでわが国には，そのための適切な社会的メカニズムが欠けていたように思われます．SEA は，そうした欠落を補うべく，これからますます活発な活動を展開して行きたいと考えています．いままで日本にはなかったこの新しいプロフェッショナル・ソサイエティの発展のために，ぜひとも，あなたのお力を貸してください．

ＳＥＡＭＡＩＬ Ｖｏｌ．１４，Ｎｏ．８ ２００６年３月２５日発行 編集人 岸田孝一
発行人 ソフトウェア技術者協会（ＳＥＡ）
〒１６０-０００４ 東京都新宿区四谷３－１２ 丸正ビル５Ｆ
T: 03-3356-1077 F: 03-3356-1072 E-mail: sea@sea.or.jp URL: http://www.sea.jp/
印刷所 市田印刷株式会社 〒１１４-００１４ 東京都北区田端２－３－２５
定価 ５００円 （禁無断転載）

平成17年02月24-26日

第4回デザインワークショップ

ソフトウエア仕様検証技術の最新動向
PROMELA/SPINを用いて

スケジュール：

2月24日(木)

1330	受付開始
1400-1430	あいさつ，案内，その他
1430-1530	Promela模型を作るために（1）
1530-1600	お茶
1600-1700	Promela模型を作るために（2）
1800-	夕食＋雑談

2月25日(金)

930-1030	基礎をすこし
1030-1100	お茶
1100-1200	SPINで検査を
1200-1300	昼食
1300-1700	演習
1800-	夕食

2月26日(土)

930-1200	成果報告　感想と討論(SPIN Promela DW2005批判など)

デザインワークショップ　DW2005

「ソフトウエア仕様検証技術の最新動向 -- PROMELA/SPIN を用いて」

主催：ソフトウエア技術者協会

異なるプロセスがあり，それぞれが通信することで構成されるシステムを如何に設計するかは，昔から議論されてきた話題です．但し，かつては一部の人たちの知識や経験だったものが，現在では多くのソフトウェア設計者が身に付けておくべき技術となりました．組込みシステムや，分散システムが一般化したことが理由として挙げられます．

今回は，SPIN というLTL と CSP に従った模型検査が可能なツールを利用することで，日頃の我々の設計作業の中で生かし，より安全で信頼性の高いシステムを作るための手段と方法について議論したいと思います．

以下は，General Description にある概要（のごく最初の部分）を訳したものです．

> Spin[ツール]は，効率的なソフトウェアの検証を目指しています（ハードウエア検証ではありません）．ツールはシステムの仕様を記述するための高次の言語 PROMELA (a PROcess MEta LAnguage)を使用します．Spin は，分散システムの設計における論理的な設計上の誤りを見つけ出すために用いることができます．具体的な例としては ,基本ソフトウェア ,データ通信プロトコル，交換システム，並列アルゴリズム，鉄道信号プロトコルといったものを挙げることができます．ツールは仕様における論理的な一貫性を検査します．デッドロック，仕様化されていない [イベントの] 受け取り，不完全なフラグ，競合状態，プロセスの相対的な速度についての保証されていない仮定について，ツールを通して知ることができます．

課題となる問題を用意して、実際に仕様を記述し、SPIN で仕様のチェックするという実習を行います．できるかぎり，SPIN のインストールされた PC をご持参下さい．
日時：　2005 年 2 月 24 日 - 26 日（2 泊 3 日）土曜日昼解散
場所：　ウェルハートピア熱海（厚生年金ハートピア熱海）
http://www.kjp.or.jp/hp_99/
定員：　15 名

DW2005のための問題(山崎利治さん作成)

ご利用に際して

あなたの物置番号は125です. いま扉は閉まっていて施錠してあります(施錠灯が点いています).

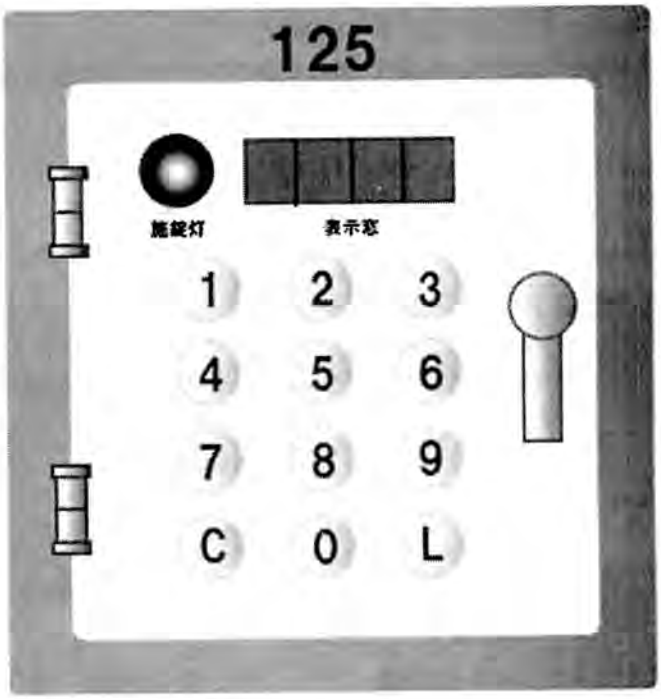


図1. 物置

物置番号の書かれた物置の扉にはパネルがあり, そこに, 施錠灯, 0から9までの押ボタン, C,Lの押ボタン, 10進数字4桁の液晶表示窓, 90度回転する取っ手が付いています.

扉の開けかた

(1) 数字ボタンを1234(施錠鍵といいます)と左から順に押してください.
(2) 押した数字を表示窓で確認してください.
(3) それが1234であれば施錠灯が消えます(開錠).
(4) そこで扉の取手を時計回りに90度廻して手前に引けば扉が開きます.
(5) 数字ボタンを押し間違えたときは, Cボタンをおして, 始からやり直してください(Cボタンを押すと表示窓の数字が消えます).

扉の閉めかたと施錠

(1) 扉を閉め取手を逆時計回りに90度廻します(機械的な施錠).
(2) Lボタンを押します(電子的な施錠).
(3) ここで施錠灯が点きます(直前に扉を開けたときのあるいは, 次項によって変更した鍵による施錠).
(4) その後は開錠手続きをしない限り, 取手を廻して引いても扉は開かないはずです.

鍵の変えかた

施錠灯が消えているときに, つぎのようにして好みの鍵にすることができます.

(1) 数字ボタンを4回押す. この4数字が新しい施錠鍵になりますので必ず表示窓で確認の上ご記憶ください.
(2) ついでLボタンを押します. そこで鍵が登録され, 表示窓の数字が消えます.

以 上

デザインワークショップ2005参加報告

佐原伸
日本フィッツ株式会社
情報技術研究所
TEL：03-3623-4683
shin.sahara@jfits.co.jp

2005年3月15日

概要

ソフトウェア技術者協会 (SEA) 主催のデザインワークショップ 2005 参加報告である。本ワークショップは、AT&T の UNIX チームが開発し、NASA が全宇宙システムの開発に使用している模型検査 (Model Checking) 言語 Promela とそのツール SPIN[2][3] を使って、並行プロセスを持つシステムのモデル（以下、模型という）化とその検査を味見しようというものであり、極めて有意義だったので報告する。

なお、本ワークショップの報告は SEA 機関誌 SEAMAIL に掲載される予定で、6月のソフトウェアシンポジウム 2005 でも関連セッションが予定されている。

1 ワークショップ日時・場所・参加者

1.1 日時

２００５年2月２４日（木）～２６日（土）

1.2 場所

ウェルハートピア熱海 (厚生年金ハートピア熱海)

1.3 参加者

(株) デンソー、九州大学２名、九州工業大学、奈良先端大学、ニル・ソフトウェア（株)、(株) SRA ２名、フリーのソフトウェア技術者２名、佐原の１１名。その内、(株) SRA の岸田氏はワークショップの記録係のため演習問題には挑戦しなかった。

2 ワークショップ要約

簡易な電子錠システムの演習問題 [8] と、ニル・ソフトウェア（株）伊藤氏による SPIN の Quick Reference の翻訳 [1] がワークショップ１～２ヶ月前に配布されていて予習できるようになっていた。

ワークショップの目的は「Promela 言語で記述した模型の検査を SPIN で行ってみる」ことで、模型構築と模型検査の重要性を実感しよう」[*1] というものであった。

当日は、日本最初のプログラマーであり形式手法にも詳しい山崎利治さんより、基礎理論にも触れながら、簡潔で分かりやすい講義が４時間あり、２日目午後から３チームに分かれて山崎さん提示の演習問題の模型構築に挑戦した。

各チームは。はじめての模型検査 (model checking) 言語とツールに苦戦して、４時間の予定を大幅に超過し、事実上徹夜で模型構築を行ったが、全チームが翌朝までに模型構築および模型検査を完了した。

2.1 講義要約

山崎さんの講義概要を以下に示す。

2.1.1 なぜ SPIN か?

実時間分散システムは、多スレッド・プログラムになり、設計の誤りを起こしやすい。竦み (deadlock)・発散・仕様の過不足・制約違反などであるが、これらの誤りは模型検査 (Model Checking) によって発見できる。

他方、並行系でなくても、オブジェクト間相互作用、つまり、系全体の挙動の記述に Promela 言語が利用できる。

従って、Java 多重スレッド、オブジェクトや成分間の協調、UML 相互作用図・交信図・時間図・状態図などの利用時に模型検査が利用できることになる。

SPIN は、教科書や教育資料が多く、ツールが公開されていて使用実績も多く、毎年ワークショップが開催されていてノウハウを吸収しやすい環境が

[*1] Promela は実システムそのものの仕様は書けず、模型の仕様記述に限定することで、模型検査を効率化している。一方、VDM[6][7] は動的な模型検査は手で行う代わりに、実システムそのものの仕様を書くことができる。

整っている。

2.1.2 模型検査とは？

システムの有限状態模型とそれが満たすべき性質を与え、自動判定する技術であり、システム検証の一方法である。SPIN では、Promela で模型を作成し、線形時間（時相）論理式 LTL(linear time temporal logic) で模型が満たすべき条件を記述し、それを Promela に変換して模型検査を行う。この検査は、かなり自動で行われるが、人手の介入も必要である。

2.1.3 時間論理式 LTL とは？

LTL は、通常の論理式を拡張して、時間を考慮した論理式を書けるようにしたものである。時間演算子には、以下のようなものがある。

表 1. 時間演算子

時間演算子	Promela での表現	意味
$p \bowtie q$	U	q がやがて真になり、それまでは p がずっと真である
$\Diamond p$	<>p	p がやがて（未来のある時点で）真になる
$\Box p$	[]p	帳簿中の明細記録の総件数を返す

2.1.4 模型検査の手順

SPIN による模型検査の手順は、以下のようになる。

- Promela で模型を作る
- 正当性を保証する仕様を LTL で記述する
- 模型検査系 SPIN で検査する
- 結果を分析する

結果の分析は、以下のように行う。

- 検査で、模型が仕様を満たせば、終了
- 満たさないときは、反例を SPIN が表示するので、検討する
- 模型と仕様を再構成し、上記手順を繰り返す

例えば、山崎さんの解答例で模型に竦み (deadlock) があるのを SPIN ツールが検出した例が下記である。

```
(Spin Version 4.2.4 -- 14 February 2005)
Warning: Search not completed
+ Partial Order Reduction
```

(注) 下記で + で表されているのが検査した性質

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

(注) エラー検出

```
State-vector 72 byte, depth reached 105, errors: 1
```

~省略~

(注) 下記がエラーの状況

```
unreached in proctype Button
line 96, state 105, "Lch!lock"
line 94, state 120, "Bch?D,d"
line 94, state 120, "Bch?L"
line 94, state 120, "Bch?C"
line 33, state 151, "D_STEP"
line 107, state 159, "-end-"
(49 of 159 states)
unreached in proctype Lock
line 115, "pan_in", state 7, "ls = 1"
line 114, "pan_in", state 9, "Lch?lock"
line 117, "pan_in", state 11, "-end-"
(3 of 11 states)
unreached in proctype Door
line 130, "pan_in", state 14, "-end-"
(1 of 14 states)
unreached in proctype user
line 141, "pan_in", state 9, "Bch!D,1"
line 142, "pan_in", state 10, "Bch!D,1"
line 143, "pan_in", state 11, "Bch!D,1"
line 144, "pan_in", state 12, "Bch!D,1"
line 149, "pan_in", state 19, "Bch!D,1"
line 149, "pan_in", state 20, "printf('send D')"
line 150, "pan_in", state 21, "Bch!D,1"
line 151, "pan_in", state 22, "Bch!D,1"
line 152, "pan_in", state 23, "Bch!D,1"
line 154, "pan_in", state 25, "Dch!OPEN"
line 153, "pan_in", state 26, "((ls==0))"
line 156, "pan_in", state 28, "Bch!D,9"
line 158, "pan_in", state 30, "Bch!L"
line 158, "pan_in", state 31, "printf('send L')"
line 157, "pan_in", state 32, "((ds==0))"
line 160, "pan_in", state 34, "Bch!D,8"
line 161, "pan_in", state 35, "Bch!D,7"
line 162, "pan_in", state 36, "Bch!D,6"
line 163, "pan_in", state 37, "-end-"
```

```
(19 of 37 states)
unreached in proctype :init:
(1 of 11 states)
        0.12 real  0.01 user 0.06 sys
```

上記のテキスト形式だけでは、見にくいので、ページ 15 のような交信図も自動的に表示され、どこでおかしくなったかが分かるようになっている。図の例では、一番右側の user プロセス (98 ステップ目) から、左側から 2 番目の Button プロセス (106 ステップ目) にメッセージ L(鍵を掛けるためのメッセージ) がチャネルを通して送られ、そこで竦み (deadlock) が発生していることが分かる。

このような模型上の欠陥を修正していけば、模型検査が終了することになる。

山崎さんの講義は、基礎理論にも触れながら、初心者が演習するための上記内容をわずか 4 時間で伝えるという優れたものだった。

以下、模型検査終了まで何とか達成した演習の状況を報告する。

2.2 演習要約

我々のチームの場合は、山崎さんもメンバーだったのと、残りの 3 人の自力では 4 時間で模型検査終了までは到底無理そうとの判断から、まず山崎さんの解答例を XSPIN という X-Window システム上の GUI ツールで検査した。

山崎さんの模型の Promela ソースを以下に示す。

Listing 1 山崎利治さんの模型

```
/* e-locker */
#define N 4
#define complete (K[0] != -1)
#define locking ls = true
#define unlocking ls = false
#define locked  (ls == true)
#define unlocked (ls == false)
#define opening ds = true
#define closing ds = false
#define opened (ds == true)
#define closed (ds == false)
#define matched (r == true)

mtype={D,L,C,OPEN,CLOSE,lock,unlock};

chan Bch = [1] of {mtype,short};
chan Dch = [0] of {mtype};
chan Lch = [0] of {mtype};

bool ls = true, ds = false,
  r = false;
short K[N], Ks[N], d, w;
byte i;

inline new()
{
  d_step {
    i = 0;
    do
    :: i < N    -> K[i] = -1; i++
    :: i == N -> break
    od
    }
}

inline ass() /* new key assignf */
{
  d_step {
    i = 0;
    do
    :: i < N      -> Ks[i] = K[i]; i++
    :: i == N -> break
    od
    }
}

inline in(d)
{
  d_step {
    assert (!complete);
    i = 0;
    do
    :: i < (N - 1)
-> K[i] = K[i+1]; i++
    :: i == (N - 1)  -> break
    od;
    K[N - 1] = d
  }
}

inline eq()
{
  bool rr;
  d_step {
```

```
    rr = true;
    i = 0;
    do
    :: i < N      ->
        rr = rr && (Ks[i] == K[i]);
       i++
    :: i == N     -> break
    od;
  r = rr
  }
}

proctype Button()
{
end:
s1: new();
  if
  :: Bch ? D(d)  -> in(d); goto s2
  :: Bch ? L(w)  -> goto s1
  :: Bch ? C(w)  -> goto s1
  fi;
s2: if
  :: complete    -> eq();
    if
    :: r == true      ->
      progress1: Lch ! unlock; goto s3
    :: r == false     -> goto s1
    fi
  :: else         -> if
    :: Bch ? D(d)     -> in(d); goto s2
    :: Bch ? C(w)     -> goto s1
    :: Bch ? L(w)     -> goto s2
    fi
  fi;
s3: if
    :: Bch ? D(d)     ->
        new(); in(d); goto s4
    :: Bch ? L(w)     ->
        progress2: Lch ! lock; ass();
        goto s1
    :: Bch ? C(w)     ->
        goto s3
    fi;
s4: if
  :: complete    -> goto s3
  :: else         -> if
    :: Bch ? D(d)     ->
        in(d); goto s4
    :: Bch ? C(w)     ->
        new(); goto s3
    :: Bch ? L(w)     ->
        goto s4
    fi
  fi
}

proctype Lock()
{
ls1: if
  :: Lch ? unlock        ->
   unlocking; goto ls2
 fi;
ls2: if
 :: Lch ? lock  ->
  locking; goto ls1
 fi
}

proctype Door ()
{
ds1: if
 ::Dch?OPEN ->
  if
  :: unlocked -> opening; goto ds2
  fi
 fi;
ds2: if
 :: Dch?CLOSE -> closing; goto ds1
 fi
}

proctype user()
{
Bch ! D(1);
Bch ! D(2);
Bch ! D(3);
Bch ! D(4);
if
::unlocked       -> Dch ! OPEN
fi;
Bch ! D(1);
Bch ! D(1);
Bch ! D(1);
Bch ! D(1);
Dch ! CLOSE;
if
::closed         ->
  Bch ! L(w); printf("send_L1");
```

```
fi;
Bch ! D(1); printf("send_D");
Bch ! D(1);
Bch ! D(1);
Bch ! D(1);
if
::unlocked          -> Dch ! OPEN
fi;
Bch ! D(9);
if
::closed            ->
  Bch ! L(w); printf("send_L")
fi;
Bch ! D(8);
Bch ! D(7);
Bch ! D(6);
}

init
{
 d_step {
  Ks[0] = 1;
  Ks[1] = 2;
  Ks[2] = 3;
  Ks[3] = 4;
 };
 atomic {
  run Button(); run Lock();
  run Door(); run user()
 }
}
```

結果は前記の通りであるが、山崎さんを含めXSPIN ツールには誰も習熟していないため、欠陥を修正するには至らなかった。[*2]

そこで、プロセスの数を減らして[*3]欠陥の修正を容易にすることと、錠と取手それぞれの開閉のみを行う縮小した模型を構築することにした。山崎さんの解答例レベルの模型は、縮小模型ができてから拡張すれば良いと判断したためである。

以下が、その縮小模型の Promela ソースである。

Listing 2 佐原の簡易版模型

```
/* Store E-Locker */
short key, digit
bool lockStatus, doorStatus

#define locking lockStatus = true
#define unlocking lockStatus = false
#define locked (lockStatus == true)
#define unlocked (lockStatus == false)
#define opening doorStatus = true
#define closing doorStatus = false
#define opened (doorStatus == true)
#define closed (doorStatus == false)
#define matched (key == digit)

mtype = {D,OPEN,CLOSE,lock,unlock}

chan ButtonCh = [1] of {mtype,short};
chan LockCh = [0] of {mtype};
chan DoorCh = [0] of {mtype};

init
{
  key = 1;
  atomic {
  run User()
  }
}

proctype User()
{
ButtonCh ! D(1); printf("try_unlock1");
if
::unlocked          ->
  DoorCh ! OPEN;
  printf("try_door_open1")
fi;
DoorCh ! CLOSE; printf("try_close1");
if
::closed            ->
  printf("try_locking");LockCh ! lock
fi;
ButtonCh ! D(1); printf("try_unlock2");
if
::unlocked          ->
```

[*2] ワークショップ後に見直したところ、模型は、Bch に引数があるメッセージと引数の無いメッセージを送っていたのだが、引数の無いメッセージを送ったときに前の引数が残っていて、その受け手がいないため詰んでいることが発見できた。そこで、引数の無いメッセージにも使わない引数を付けたところ、模型のエラーは無くなった。

[*3] プロセスが多いと再利用性は増えるものの、問題に含まれない余分な並行性も発生するため、初心者には予測できない模型の動きが多発するので、欠陥修正が難しい。

```
  DoorCh ! OPEN;
   printf("try_door_open2")
fi
}

active proctype Ctrl()
{
lockS:
if
:: ButtonCh ? D(digit)  ->
  if
  :: matched     ->
    printf("unlocking"); goto closeS
  :: else        ->
    goto lockS
  fi
fi;
closeS:
if
:: LockCh ? lock          ->
  printf("locking");
  goto lockS
:: DoorCh ? OPEN          ->
  opening; printf("opening");
  goto openS;
fi;
openS:
if
:: DoorCh ? CLOSE         ->
  closing; printf("closinglocked");
  goto closeS
fi;
}

  /*
  * Formula As Typed:
  *       locked -> <> unlocked
  * The Never Claim Below Corresponds
  * To The Negated Formula
  *
!(locked -> <> unlocked)
   * (formalizing violations
   *     of the original)
   */

never {
/* !(locked -> <> unlocked) */
accept_init:
T0_init:
  if
  :: (! ((unlocked)) && (locked)) ->
    goto accept_S3
  fi;
accept_S3:
T0_S3:
  if
  :: (! ((unlocked))) -> goto accept_S3
  fi;
}
```

この模型の最後から 14 行目の never 以降は、否定要求 (never claim) といって、検査仕様の LTL 式から生成された仕様である。ここでは「鍵が掛かっていたら (loecked) いつかは鍵が開けられる (unlocked)」という命題に反することはない、すなわち「鍵が開かなくなる」ことは絶対にないことが保証される。

つまり、検査したいことを否定要求として記述することで、「起きて欲しくないことは起きない」という保証が得られる訳である。

最後の模型は、最終日午前中の発表の４５分前に書き始めた。ここに至るまでに、３つの模型を書き (内一つは動かず)、大きな修正を５回行ったため、ほぼ徹夜となったが、そのおかげで、Promela と XSPIN にある程度習熟し始めたのである。

3 VDM 模型との比較

実は、ワークショップの前と後で、演習問題の VDM++[5] による模型を作っていた。VDM++ による並行処理模型の作り方にはまだ習熟していないし、まずは並行性を排除した模型で正しさを確認すべきだという VDMTools の生みの親 Larsen 博士の文書 [4] を信じて、節 3.1 のような VDM++ 模型を作った。

VDM++ には習熟しているので、この模型は「鍵の押し間違えの修正」機能を除いて、ほぼ問題文に忠実に作った。

結果として、SPIN 模型では発見できないであろう５個の誤りを発見した。いずれも、事後条件の間違いである。

SPIN は並行プロセスの状態遷移に内在する矛盾

検出に役立ち[*4]、VDM++ は状態遷移したときの動作 (操作や関数) に必要な制約 (事後条件) を明確化するのに役立つ。

3.1 佐原の VDM++ 模型

[*4] ただし、現実の問題では「状態の爆発」が置き、事実上検査ができなくなるので、模型簡易化 (model slicing) を行うことによって、状態数を減らす必要がある。

3.1.1 Store

Store は、電子錠の付いた物置で、かつ、そのテスト用のクラスである。本来は、テスト機能は別クラスに分離すべきであるが、手を抜いた。

new Store().test() とすることで、想定した全てのテストを行う。

class *Store* is subclass of *KeyCommon*

```
instance variables
        public 錠 : 『錠』 := new 『錠』();
        public 取手 : 『取手』 := new 『取手』();
        public 施錠灯 : 『施錠灯』 := new 『施錠灯』();
        public 表示窓 : 『表示窓』 := new 『表示窓』();
        public ボタン : 『ボタン』 := new 『ボタン』();

operations
        Init : () →o ()
        Init () △
          (  取手.Init(施錠灯);
             錠.Init(施錠灯, 取手, 表示窓, ボタン);
             施錠灯.点ける();
             取手を閉める();
             表示窓.消す()
          );
public
        解錠する : 「鍵」 →o ()
        解錠する (a 鍵) △
          (  表示窓.ボタン列を設定する(a 鍵);
             錠.解錠する()
          );
public
        施錠する : () →o ()
        施錠する () △
          (  ボタン.ボタンを設定する(L ボタン);
             錠.施錠する()
          );
public
        取手を開ける : () →o ()
        取手を開ける () △ 取手.
          開く();
public
        取手を閉める : () →o ()
        取手を閉める () △ 取手.
          閉める();
public
        施錠鍵を登録する : 「鍵」 →o ()
        施錠鍵を登録する (a 鍵) △
          (  表示窓.ボタン列を設定する(a 鍵);
             ボタン.ボタンを設定する(L ボタン);
             錠.登録する()
          );
```

以下はテスト機能である。

```
public
        test : () → 𝔹
        test () △
          (  return t1()∧t2()∧t3()∧t4()∧t5()∧
t6()
          );
public
        t1 : () →o 𝔹
        t1() △
          (  Init();
             解錠する([1,2,3,4]);
             return 施錠灯.消えている()
          );
public
        t2 : () →o 𝔹
        t2() △
          (  Init();
             解錠する([1,2,3,4]);
             取手を開ける();
             return 施錠灯.消えている() ∧
                    取手.開いている()
          );
public
        t3 : () →o 𝔹
        t3() △
          (  Init();
             解錠する([1,2,3,4]);
             取手を開ける();
             施錠鍵を登録する([1,9,1,9]);
             return 施錠灯.消えている() ∧
                    錠.鍵が一致([1,9,1,9])
          );
public
```

```
        t4 : () →o 𝔹
        t4() ≜
          (   Init();
              解錠する([1,2,3,4]);
              取手を開ける();
              施錠鍵を登録する([1,9,1,9]);
              取手を閉める();
              施錠する();
              return 施錠灯.点いている() ∧
                     表示窓.消えている()
          );
public
        t5 : () →o 𝔹
        t5() ≜
          (   Init();
              解錠する([1,2,3,4]);
              取手を開ける();
              施錠鍵を登録する([1,9,1,9]);
              取手を閉める();
              施錠する();
              解錠する([1,9,1,9]);
              return 施錠灯.消えている() ∧
                     取手.閉まっている()
          );
public
        t6 : () →o 𝔹
        t6() ≜
          (   Init();
              解錠する([1,2,3,4]);
              取手を開ける();
              施錠鍵を登録する([1,9,1,9]);
              取手を閉める();
              施錠する();
              解錠する([1,9,1,9]);
              return 施錠灯.消えている() ∧
                     取手.閉まっている()
          )
end Store
```

3.1.2 KeyCommon

共通の型や値を定義する。

```
class KeyCommon
values
public
        鍵桁数 = 4;
public
        Lボタン = 'L';
public
        Cボタン = 'C'
types
        public 「鍵」= ℕ*
end KeyCommon
```

3.1.3 『錠』

電子錠である。

```
class 『錠』 is subclass of KeyCommon
instance variables
        public 施錠鍵 : 「鍵」;
        public 取手 : 『取手』;
        public 施錠灯 : 『施錠灯』;
        public 表示窓 : 『表示窓』;
        public ボタン : 『ボタン』;
        inv 正しい鍵(施錠鍵)

functions
public static
        正しい鍵 : 「鍵」→ 𝔹
        正しい鍵(a 鍵) ≜
          len a 鍵 = 鍵桁数 post len a 鍵 = 鍵桁数
operations
public
        鍵が一致 : 「鍵」→o 𝔹
        鍵が一致(a 鍵) ≜
          return 施錠鍵 = a 鍵;
public
        Init : 『施錠灯』×『取手』×『表示窓』×『ボタン』→o ()
        Init(a 施錠灯, a 取手, a 表示窓, a ボタン) ≜
          (   施錠鍵 := [1,2,3,4];
              施錠灯 := a 施錠灯;
              取手 := a 取手;
              表示窓 := a 表示窓;
              ボタン := a ボタン
          );
public
        『錠』 : () →o 『錠』
        『錠』() ≜
          (   return self
          );
public
        『錠』 : 「鍵」→o 『錠』
        『錠』(a 鍵) ≜
          (   施錠鍵 := a 鍵;
              return self
          )
        pre 正しい鍵(a 鍵);
public
```

```
        施錠する : () →^o ()
        施錠する () ≜
          (   if 取手.閉まっている ()
              then (   表示窓.消す();
                       施錠灯.点ける()
                   )
              else skip
          )
        pre
表示窓.点いている ()  ∧  施錠灯.消えている ()  ∧
取手.閉まっている () ∧ ボタン.L()

post
表示窓.消えている ()  ∧  施錠灯.点いている ()  ∧
取手.閉まっている ();
public
        解錠する : () →^o ()
        解錠する () ≜
          (   if 表示窓.点いている () ∧ 鍵が一致 (表
示窓.内容) ∧ 正しい鍵 (施錠鍵)
              then (   表示窓.消す();
                       施錠灯.消す()
                   )
              else skip
          )
        pre             表示窓.点いている ()      ∧
施錠灯.点いている () ∧ 正しい鍵 (施錠鍵)

post
表示窓.消えている () ∧ 施錠灯.消えている ();
public
        登録する : () →^o ()
        登録する () ≜
          施錠鍵 := 表示窓.内容
        pre             表示窓.点いている ()      ∧
施錠灯.消えている () ∧ ボタン.L()

post
鍵が一致 (表示窓.内容)
end『錠』
```

3.1.4 『施錠灯』

```
class『施錠灯』is subclass of KeyCommon
instance variables
        public 点灯 : 𝔹;

operations
public
        点ける : () →^o ()
        点ける () ≜
          点灯 := truepost 点いている ();
public
        消す : () →^o ()
        消す () ≜
          点灯 := falsepost 消えている ();
public
        点いている : () →^o 𝔹
        点いている () ≜
          return 点灯post 点灯;
public
        消えている : () →^o 𝔹
        消えている () ≜
          return ¬ 点灯post ¬ 点灯
end『施錠灯』
```

3.1.5 『表示窓』

```
class『表示窓』is subclass of KeyCommon
instance variables
        public 内容 :「鍵」:= [];

operations
public
        設定する :『ボタン』'「ボタン」 →^o ()
        設定する (a ボタン) ≜
          if len 内容 ≥ 鍵桁数
          then skip
          else 内容 := 内容 ↷ [a ボタン]
        pre『ボタン』'数字 (a ボタン)

post
if len 内容 ≥ 鍵桁数
            then 内容 = ←内容
            else 内容 = ←内容 ↷ [a ボタン];
```

以下の操作は、ボタン列の最後から鍵桁数分だけを内容に設定する。Drop は、自作ライブラリで定義されている関数で、Drop(n)(列) と呼ぶと、列から頭の n 個を削除した列を返す。

```
public
        ボタン列を設定する :『ボタン』'「ボタン」* →^o
()
        ボタン列を設定する (a ボタン列) ≜
          内容 := FSequence'Drop[ℕ]
                    (
                     len 内容 + len a ボタン列 −
鍵桁数)
                    (
                     内容 ↷ a ボタン列)
```

pre 『ボタン』`数字列 (a ボタン列)

post
内容 =
$FSequence`Drop[\mathbb{N}]$
(
$\overleftarrow{\text{len 内容}}$ + len a ボタン列 − 鍵桁数)
(
$\overleftarrow{\text{内容}} \curvearrowright a$ ボタン列);
public
消す : () $\xrightarrow{o}$ ()
消す () $\triangleq$
内容 := []post 消えている ();
public
点いている : () $\xrightarrow{o}$ $\mathbb{B}$
点いている () $\triangleq$
return 内容 $\neq$ [] post 内容 $\neq$ [];
public
消えている : () $\xrightarrow{o}$ $\mathbb{B}$
消えている () $\triangleq$
return 内容 = [] post 内容 = []
end 『表示窓』

3.1.6 『ボタン』

class 『ボタン』 is subclass of $KeyCommon$
types
public 「ボタン」 = $\mathbb{N}$ | char
inv ボタン $\triangleq$ 数字(ボタン) $\vee$ ボタン $\in$ {'C','L'}
instance variables
public 内容 : 「ボタン」 := 'L';

functions
public static
数字 : 「ボタン」 → $\mathbb{B}$
数字 (a ボタン) $\triangleq$
a ボタン $\in \{0, \ldots, 9\}$;
public static
数字列 : 「ボタン」* → $\mathbb{B}$
数字列 (a ボタン列) $\triangleq$
$\forall b \in$ elems a ボタン列 · 数字 (b)
operations
public
C : () $\xrightarrow{o}$ $\mathbb{B}$
C () $\triangleq$
return 内容 = C ボタン;
public
L : () $\xrightarrow{o}$ $\mathbb{B}$
L () $\triangleq$
return 内容 = L ボタン;
public
ボタンを設定する : 「ボタン」 $\xrightarrow{o}$ ()
ボタンを設定する (a ボタン) $\triangleq$
内容 := a ボタン
end 『ボタン』

3.1.7 『取手』

class 『取手』 is subclass of $KeyCommon$
instance variables
public 開閉 : $\mathbb{B}$:= false;
public 施錠灯 : 『施錠灯』;

operations
public
$Init$: 『施錠灯』 $\xrightarrow{o}$ ()
$Init$ (a 施錠灯) $\triangleq$
施錠灯 := a 施錠灯;
public
開く : () $\xrightarrow{o}$ ()
開く () $\triangleq$
if 施錠灯.消えている ()
then 開閉 := true
else 開閉 := falsepost if 施錠灯.消えている ()
then 開閉 = true
else 開閉 = false;
public
閉める : () $\xrightarrow{o}$ ()
閉める () $\triangleq$
開閉 := falsepost 閉まっている ();
public
開いている : () $\xrightarrow{o}$ $\mathbb{B}$
開いている () $\triangleq$
return 開閉post 開閉;
public
閉まっている : () $\xrightarrow{o}$ $\mathbb{B}$
閉まっている () $\triangleq$
return ¬開閉post ¬開閉
end 『取手』

4 まとめ

4.1 他チームとの比較

最終日の発表と討議により、他チームの進め方や考え方あるいは失敗から、良い教訓を学ぶことができた。

例えば、伊藤さんのチームは、以下のようなしっかりしたプロセスで模型作成を行い、他人に開発過程を見せることができる模型作成を行っていた。

- 問題領域にあるオブジェクトを認識する
- それをベースに、プロセスを見つけ、その間のチャネルとメッセージを考える
- 重要なプロセス群を見つけ、そこをまず模型化する

野中さんのチームは、模型検査を一番精密に行っていたように思うが、野中さんの失敗例が一番おもしろかった。すなわち、野中さんの最初の模型は「正しい鍵とは異なる鍵を使って、いつかは鍵が開いてしまう」例が提示されたそうだ。これは、鍵が掛かっているときにも、鍵の変更ができてしまう模型を書いたための失敗だが、もっと複雑なシステムでは、このような単純な失敗を人手で見つけることは難しいので、SPIN の威力を感じた例であった。

我々のチームは、行き当たりばったりで「3人中誰かは模型作成できるであろう」という方針を取った。大規模システムの開発には向かないが、4時間で模型検査まで行くには、この方式も良いだろうと思ったのである。

多くの試行錯誤を行ったが、結果として他チームより多くの版の Promela ソースを書き、SPIN の初歩の経験を積むには良かった。囲碁や将棋の世界では「下手なうちに考えるな。多くのゲームをし、試行錯誤を行って、その世界の感覚をつかめ」という上達のための極意があるが、それをそのまま使ってみたわけである。

4.2 模型検査は必須

「このような模型検査系を使わずに、どうして並行処理プログラムを設計し得るのでしょうか？」という奈良先端科学技術大学院大学蔵川先生の言葉が、今回のワークショップの意義を表している。

並行処理プログラムの作成は、開発現場で行われているような、経験や根性といった「体で覚える」方式の開発では、必ず欠陥が発生することを実感した。理論と定理と、それらを支援する言語とツールを使い、その上に立脚した経験をもとに開発することが必要だろう。

4.3 なぜ現場で使わないのか？

しかし、既にかなり使い込まれていて、無料で使用でき、理論的にもしっかりしている模型検査ツールが、なぜか日本ではほとんど使われていない。

今回のようなワークショップを、仮に UML を対象として行った場合、ほとんどの参加者は模型構築に至らないだろう。反対に、今回は参加者のほとんどが模型構築に成功した。そこが、UML のような意味の曖昧な図形言語を核とした手法と、形式手法の違いであろう。とっつきにくいが、結局は早いのだ。しかも、あとで見直すことが容易である。

今回のワークショップをきっかけとして、模型検査や形式手法を日本で普及させなければならない、という確信を持った。

5 ワークショップ後の改良モデル

ワークショップの後、改良した山崎さんの模型を以下に示す。筆者のモデルは、まだ改良していないので、次回 SEAMAIL で報告することとしたい。

Listing 3 山崎利治さんの最終模型

```
/* $Id: E-locker.pml,v 1.10 2005/03/15$
 *
 * e-locker */

#define N 2
#define locking ls = true
#define unlocking ls = false
#define locked (ls == true)
#define unlocked (ls == false)
#define opening ds = true
#define closing ds = false
#define opened (ds == true)
#define closed (ds == false)
#define eq (Ks == K)
#define kfull (i == N)
```

```
mtype={D,L,C,OPEN,CLOSE,lock ,unlock};

chan Bch = [1] of { mtype, bit };
chan Dch = [0] of { mtype };
chan Lch = [0] of { mtype };

bool ls = true, ds = false;
short K, Ks;
byte i;
bit d,t;



inline new()
{ d_step{ K = 0; i = 0 }}

inline in(d)
{ d_step{ K = 2*K+d; i++ }}


proctype Button()
{
 end:
 bs1: new();
   if
   :: Bch ? D(d)  -> in(d); goto bs2
   :: Bch ? C(t)  -> goto bs1
   :: Bch ? L(t)  -> goto bs1
   fi;
 bs2: if
   :: kfull ->
     if
     :: eq ->
         progress1: Lch ! unlock;
         goto bs3
     :: else    -> goto bs1
     fi
   :: else  ->
     if
     :: Bch ? D(d) -> in(d); goto bs2
     :: Bch ? C(t) -> goto bs1
     :: Bch ? L(t) -> goto bs2
     fi
   fi;
 bs3: if
   :: Bch ? D(d) ->
        new(); in(d); goto bs4
   :: Bch ? L(t) ->
     progress2:
     if
     :: kfull ->
          Lch ! lock; Ks = K; goto bs1
     :: else  -> goto bs3
     fi
   :: Bch ? C(t) -> goto bs3
   fi;
 bs4: if
   :: kfull -> goto bs3
   :: else  -> if
     :: Bch ? D(d) -> in(d); goto bs4
     :: Bch ? C(t) -> new(); goto bs4
     :: Bch ? L(t) -> goto bs4
     fi
   fi
}

proctype Lock()
{
  ls1: if
    :: Lch ? unlock ->
         unlocking; goto ls2
    :: Lch ? lock   -> goto ls1
    fi;
  ls2: if
    :: Lch ? lock
-> locking; goto ls1
    :: Lch ? unlock -> goto ls2
    fi
}

proctype Door()
{
  ds1: if
    :: Dch ? OPEN  ->
      if
      :: unlocked -> opening; goto ds2
      :: else     -> goto ds1
      fi
    :: Dch ? CLOSE -> goto ds1
    fi;
  ds2: if
    :: Dch ? CLOSE
-> closing; goto ds1
    :: Dch ? OPEN   -> goto ds2
    fi
}
```

```
proctype user()
{ Bch ! D(1);
  Bch ! C(t);
  Bch ! D(1);
  Bch ! D(1);
  unlocked; Dch ! OPEN;
  Bch ! D(1);
  Bch ! C(t);
  Bch ! D(0);
  Bch ! D(0);
  Bch ! L(t);
  Dch ! CLOSE;
  Bch ! D(0);
  Bch ! D(0);
  unlocked; Dch ! OPEN
}

proctype watch()
{ do
  :: timeout -> printf("deadlock␣?")
  od
}

proctype user0()
{ do
  :: Bch ! D(0)
  :: Bch ! D(1)
  :: Bch ! C(t)
  :: Bch ! L(t)
  :: Dch ! OPEN
  :: Dch ! CLOSE
  od
}


trace{
  do
  :: Lch ? unlock; Lch ? lock
  od
}

init
{ Ks = 3;
  atomic{
    run Button(); run Lock();
    run Door(); run user();
    run watch()}
}
```

参考文献

[1] 伊藤昌夫（翻訳）Gerth. Promela 簡介, Feb. 2005.

[2] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, Sep. 2003.

[3] Gerard J.Holzmann 著, 水野忠則・東野輝夫・佐藤文明・太田剛訳. コンピュータプロトコルの設計法. カットシステム, Nov. 1997.

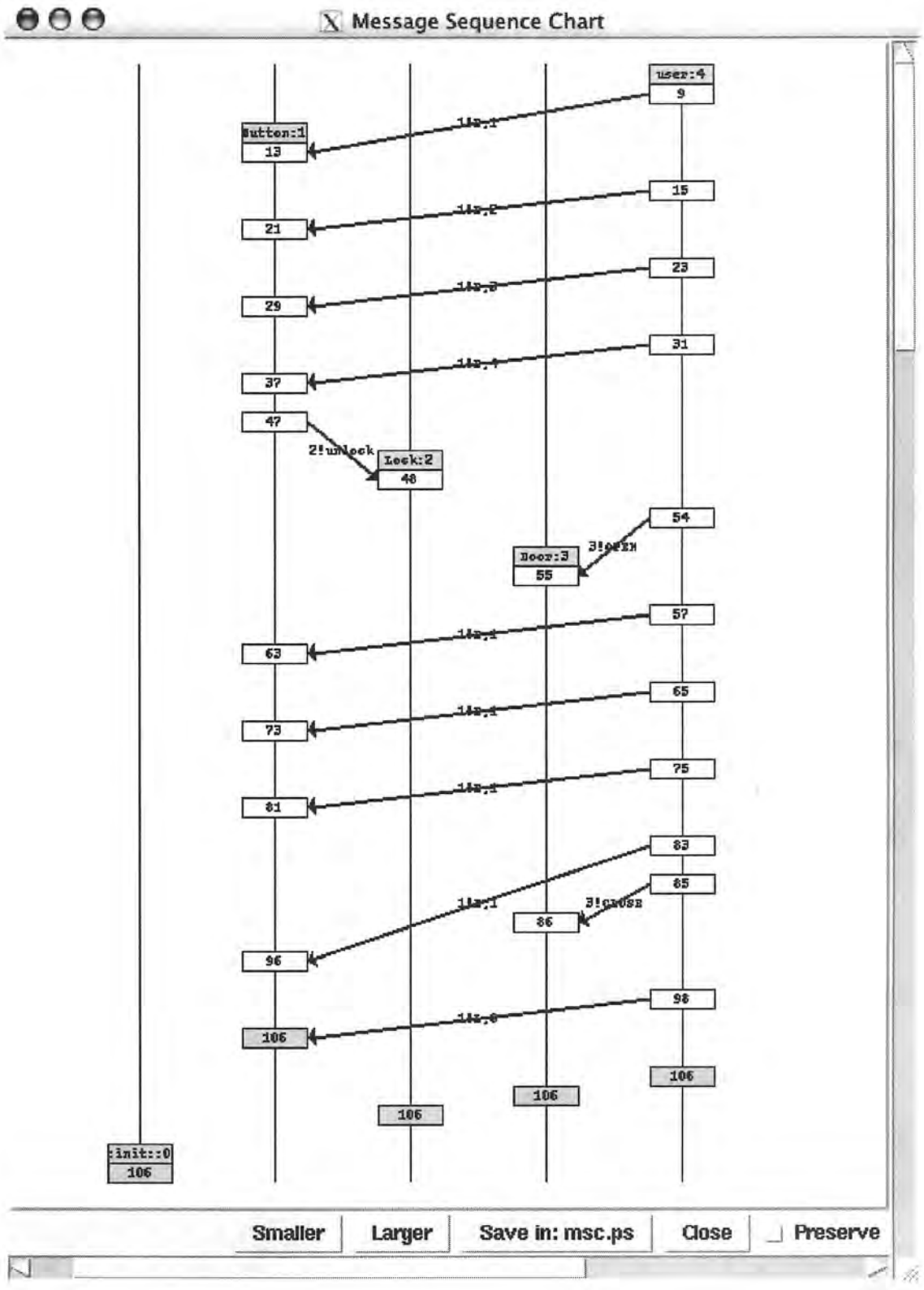
[4] IFAD. Development guidelines for real-time systems using vdmtools. Technical report, CSK, 2000.

[5] IFAD. *The IFAD VDM++ Language.* CSK, v6.8 edition, 2001.

[6] Cliff Jones. *Systematic Software Development using VDM.* Prentice Hall International, 1990.

[7] ジョン・フィッツジェラルド, ピーター・ゴーム・ラーセン. VDM++ 関数型回帰テスト支援ライブラリ. 岩波書店, 2003.

[8] 山崎利治. Dw2005 のための問題, Feb. 2005.


Message Sequence Chart
user:4
9
Button:1
13
15
21
23
29
31
37
47
2!unlock
Lock:2
48
54
Door:3
55
57
63
65
73
75
81
83
85
86
96
98
106
106
106
106
:init::0
106
Smaller
Larger
Save in: msc.ps
Close
Preserve

課題「電子錠物置」

山崎利治

以下は私の DW2005 の演習課題の解答です．肝心の Promela/SPIN の理解不足に，それに輪をかけた耄碌によって拙く虫までいるといった代物ですが，おもちゃの小問題でも並行プロセスになれていないと手を焼くことになる見本としてご笑覧ください．じつに汗顔のいたりです．

課題の検討

課題の Promela 模型を構成するために (逐次的には進まないが) つぎのような作業を行った．

1. 事象・作用を洗い出し
2. 利用事例の検討
3. 状態変数と状態述語術語の検討
4. Lamport による TLA 風の仕様記述
5. Statechart を書く
6. 検査項目の洗い出し
7. Promela コード作成

事象・作用を洗い出し

これは物置の利用者の操作と，それらに対する物置側の応答がすべてで「ご利用に際して」から汲み取れる．模型を簡略化するためにつぎのように考えた．

1. 押し釦はすべてそのまま採用する．
2. 取っ手は廻すを捨象，「開ける」と「閉める」に簡略化．
3. 施錠灯は施錠中は点灯，開錠中は消灯として無視，ただし，利用者の挙動としては「消灯を確認して」（鍵を変更する）があるので記憶しておく．
4. 上の利用者の操作に対して対応する物置の作用を用意する．
5. あとで気が付いて加えた物置の作用がある．

利用事例の検討

つぎのような利用事例が想定できる．

1. 開錠：：施錠・閉扉状態で数字釦を N 回押して施錠鍵を構成し開錠する.
2. 開扉：：開錠状態で「開ける」と扉が開く.
3. 鍵の変更：：開錠状態（施錠灯は消灯）において「消灯を確認して」数字釦を N 回押して新しい鍵を構成する. ここで L 釦を押すとこの鍵によって施錠される.
4. 閉扉：：「閉める」と扉が閉まる.

状態変数と状態述語術語の検討

模型を構成するために物置の状態変数と必要な状態述語を考える. つぎが想定できる (詳細は「TLA 風の仕様記述」参照).

定数

鍵の桁数 N, 10 進数字集合 Digits, 数字釦を押した回数の変域 I , 交信信号集合 M.

状態変数

(1) 鍵の状態 ls, 扉の状態 ds, 施錠鍵 Ks(初期値 kk), 押した数字による構成鍵 K(初期値 0), 数字釦を押した回数 i(初期値 0).

(2) 利用者と物置間の交信の媒体となる待行列 qB(釦関係), qD(取っ手関係「開ける」,「閉める」).

(3) できるだけプロセスを自明なものにするためにいくつかのプロセスをつくることにする. 鍵プロセス, 扉プロセス, 釦プロセスがそれである. 補助的に待行列 qL を用意する.

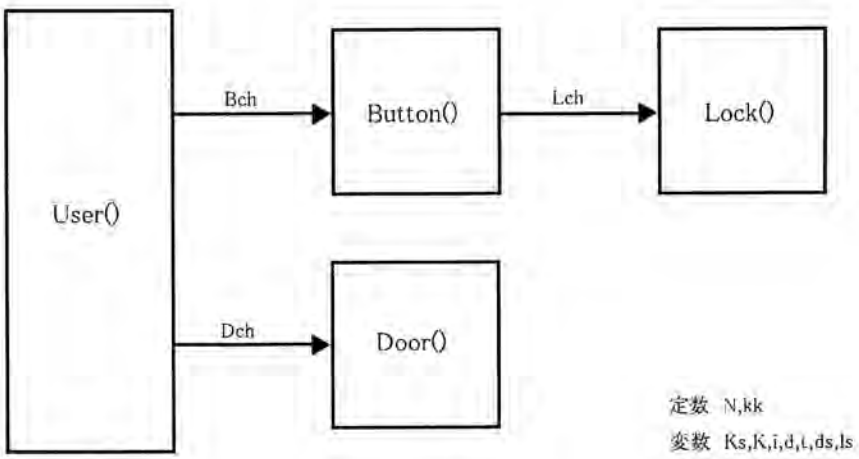


図 1 プロセスと通信路

状態述語

施錠中 locked, 開扉中 opened, 施錠灯消灯中 indoff, 構成鍵の初期状態 newk, N 桁構成鍵を構成した kfull, 施錠鍵と構成鍵が合致した keq.

TLA 風の仕様記述

つぎが上のように考えて書いた課題の仕様である (本来記述すべき安全性や活性はここには書いていない．つまり，模型部分だけである)．プロセス User と Locker が交信を繰り返し，その目的を果たすわけである．User がその行為として，SendB(d), SendL, SendC, SendO, SendCl を行う．また，RcvXx はそれに答える Locker の行為で，さらにいくつかの副次的な Locker の行為がある (SendU, SendLk, RcvU, RcvL, Akfull).

```
------------------------------------------
e-locker
------------------------------------------
CONSTANTS
 byte N;
 Digits == {0..9};
 I == {0..N};
 M == {D, C, L, OPEN, CLOSE, lock, unlock};
 short kk;
------------------------------------------
VARIABLES
 bool ls, ds;
 short Ks, K;
 i : I;
 d : Digits;
 qB : queue(M*Digits);
 qD : queue(M);
 qL : queue(M);
------------------------------------------
STATE PREDICATES
 locked == (ls==true);
 opened == (ds==true);
 indoff == (ls==false);
 newk == (i==0 && K==0);
 kfull == (i==N);
 keq == (Ks==K);
```

```
-----------------------------------------
INIT ==
    Ks == kk
 &&  locked
 &&  !opened
 &&  newk
 &&  qB==newq
 &&  qD==newq
 &&  qL==newq
-----------------------------------------
ACTIONS

 SendB(d) ==  exists d:Digits. qB'==enq(qB,(D,d));

 SendL    ==  qB'==enq(qB,(L,0));

 SendC    ==  qB'==enq(qB,(C,0));

 SendO    ==  qD'==enq(qD,OPEN);

 SendCl   ==  qD'==enq(qD,CLOSE);

 SendU    ==  qL'==enq(qL,unlock);

 SendLk   ==  qL'==enq(qL,lock);

 RcvD(x:M*Digits)  == c1 || c2 || c3 || c4
  where
  pp == !emp(qB) && qB'==deq(qB)
     && exist d:Digits.(x==front(qB) && x==(D,d)
  c1 == pp && locked && newk && K'==10*K+d && i'==i+1
  c2 == pp && locked && !newk && K'==10*K+d && i'==i+1
        && (i'!=N || Akfull)
  c3 == pp && !locked && kfull && K'==d && i'==1
  c4 == pp && !locked && !kfull && K'==10*K+d && i'==i+1;

 Akfull  == locked && kfull &&
```

```
          && ((keq && SendU)||(!keq && K'==0 && i'==0));

 RcvC  == (!emp(qB) && front(qB)==(C,0))->
        (K'==0 && i'==0  && qB'==deq(qB));

 RcvL  == (!emp(qB) && front(qB)==(L,0) && !locked)->
    (SendLk && K'==0 && i'==0 && qB'==deq(qB));

 RcvLk == !emp(qL) && front(qL)==lock && (locked || ls'==!ls)
     && qL'==deq(qL);

 RcvU == locked &&(!emp(qL) && front(qL)==unlock ->
     (locked || ls'==!ls) && qL'==deq(qL);

 RcvO == (!emp(qD) && front(qD)==OPEN && (opened || ds'==!ds)
     && qD'==deq(qD);

 RcvCl == (!emp(qD) && front(qD)==CLOSE && (!opened || ds'==!ds)
      && qD'==deq(qD);
-----------------------------------------
 Next == SendB(d) || SendL || SendC || SendO
     || SendCl || SendU || SendLk || Send
     || RcvB(x) || RcvL || RcvC || RcvU
     || RcvLk || RcvO || RcvCl || Akfull
-----------------------------------------
SPEC == INIT && []Next && WF(Next)
-----------------------------------------
```

* queue は先入先出待行列でふつうの演算 newq,enq,deq,front,(is)emp(ty) をもつ.

* ACTIONS で ' を付けた変数は ACTION の実行後の値を表す. また ' を付けた変数以外の変数値は ACTION によって変わらない.

Statechart を書く

以下は課題の statechart である. Promela コードを書くために便利と思ったからである. そこでプロセスとして, Button, Lock, Door を書いた. User は自明であるから書かなかった.

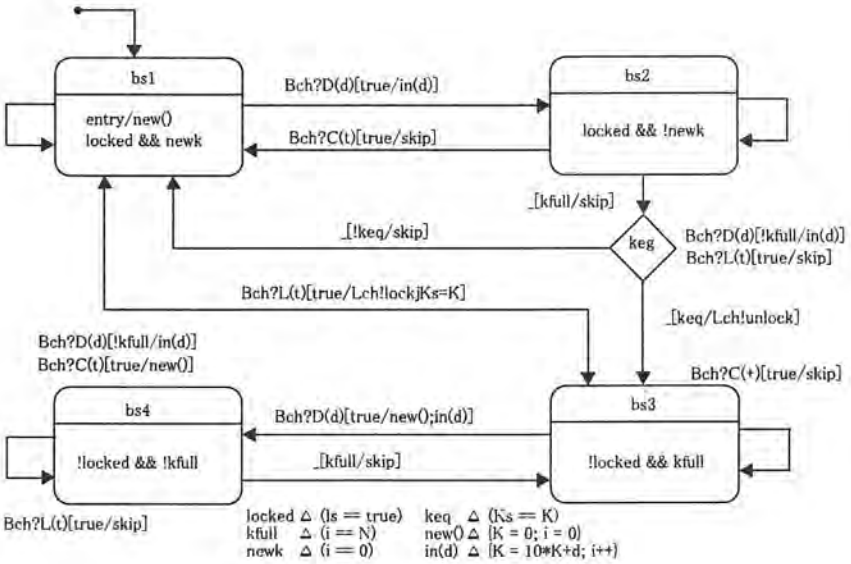


図 2　釦の状態遷移図

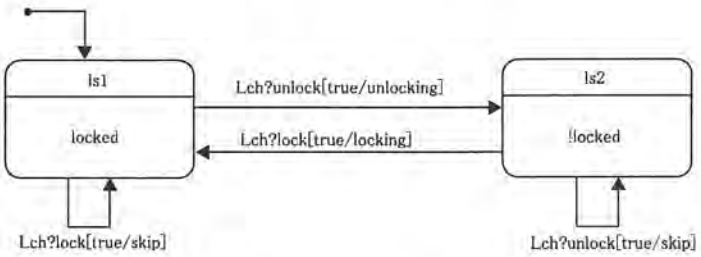


図 3　鍵の状態遷移図

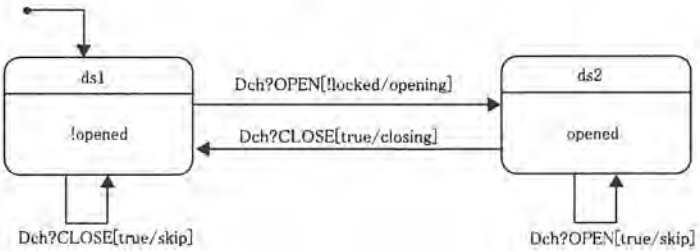


図 4　扉の状態遷移図

検査項目の洗い出し

安全性の主張は []Next としたものと，それらによって状態変数の型が不変であることが考えられるが，この検査は省略する．

活性についての主張にはさまざま確かめたい．たとえば，利用事例で記した項目を検査したい．

1. 鍵を知っていれば開錠でき扉も開けられる．
2. 鍵を知らなければ開錠できず，したがって扉を開けられない．
3. 開錠状態において「消灯を確認して」数字釦を N 回押して新しい鍵を構成できる．ここで L 釦を押すとこの鍵によって施錠される．

そこで，

1. opened に!locked が先行する．
2. !locked に keq が先行する．

などと考え，

1. opened に!locked が先行する．
   `<>opened -> (locked U (!locked && !opened))`
2. !locked に keq が先行する．
   `<>opend -> (!keq U (keq && !opened))`

を検査してみる．

Promela コード

つぎが先の statechart を忠実にコード化したものである．ただし，模型を小さくするために 10 進数字を 2 進数字にし，桁数も 2 とした．コード中に検査向きに実際には模擬実行させないプロセスもある．init プロセスを書き換えて実行させてみようと思ったからである．プロセス，通信路，大域変数，状態述語などは図示の通りである．

```
/* e-locker */

#define N 2
#define kk 3
#define locking ls = true
#define unlocking ls = false
#define locked (ls == true)
#define unlocked (ls == false)
```

```
#define opening ds = true
#define closing ds = false
#define opened (ds == true)
#define closed (ds == false)
#define keq (Ks == K)
#define kfull (i == N)


mtype = { D, L, C, OPEN, CLOSE, lock, unlock };

chan Bch = [1] of { mtype, bit };
chan Dch = [0] of { mtype };
chan Lch = [0] of { mtype };

bool ls = true, ds = false;
short K, Ks;
byte i;
bit d,t;

inline kinit()
{Ks=kk}

inline new()
{ d_step{ K = 0; i = 0 }}

inline in(d)
{ d_step{ K = 2*K+d; i++ }}


proctype Button()
{
 end:
 bs1: new();
      if
      :: Bch ? D(d)  -> in(d); goto bs2
      :: Bch ? C(t)  -> goto bs1
      :: Bch ? L(t)  -> goto bs1
```

```
      fi;
 bs2: if
      :: kfull ->
          if
          :: keq  -> progress1: Lch ! unlock; goto bs3
          :: else -> goto bs1
          fi
      :: else  ->
          if
          :: Bch ? D(d) -> in(d); goto bs2
          :: Bch ? C(t) -> goto bs1
          :: Bch ? L(t) -> goto bs2
          fi
      fi;
 bs3: if
      :: Bch ? D(d) -> new(); in(d); goto bs4
      :: Bch ? L(t) ->
           progress2:
             if
             :: kfull -> Lch ! lock; Ks = K; goto bs1
             :: else  -> goto bs3
             fi
      :: Bch ? C(t) -> goto bs3
      fi;
 bs4: if
      :: kfull -> goto bs3
      :: else  -> if
                  :: Bch ? D(d) -> in(d); goto bs4
                  :: Bch ? C(t) -> new(); goto bs4
                  :: Bch ? L(t) -> goto bs4
                  fi
      fi
}

proctype Lock()
{
  ls1: if
```

```
        :: Lch ? unlock -> unlocking; goto ls2
        :: Lch ? lock   -> goto ls1
        fi;
  ls2: if
        :: Lch ? lock   -> locking; goto ls1
        :: Lch ? unlock -> goto ls2
        fi
}

proctype Door()
{
  ds1: if
        :: Dch ? OPEN  ->
             if
             :: unlocked -> opening; goto ds2
             :: else     -> goto ds1
             fi
        :: Dch ? CLOSE -> goto ds1
        fi;
  ds2: if
        :: Dch ? CLOSE  -> closing; goto ds1
        :: Dch ? OPEN   -> goto ds2
        fi
}

proctype User()
{ Bch ! D(1);
  Bch ! C(t);
  Bch ! D(1);
  Bch ! D(1);
  unlocked; Dch ! OPEN;
  Bch ! D(1);
  Bch ! C(t);
  Bch ! D(0);
  Bch ! D(0);
  Bch ! L(t);
  Dch ! CLOSE;
```

```
  Bch ! D(0);
  Bch ! D(0);
  unlocked; Dch !  OPEN
}

proctype watch()
{ do
  :: timeout -> printf("deadlock ?")
  od
}

proctype User0()
{ do
  :: Bch ! D(0)
  :: Bch ! D(1)
  :: Bch ! C(t)
  :: Bch ! L(t)
  :: Dch ! OPEN
  :: Dch ! CLOSE
  od
}


trace{
  do
  :: Lch ? unlock; Lch ? lock
  od
}

init
{ kinit();
  atomic{
    run Button(); run Lock(); run Door(); run User();
    run watch()}
}
```

検査の実際

Promela 模型ができたら，まずは模擬実行である．小さな課題だからと侮っていたら散々であった．実際は佐原さんに熱海で動かしていただいたのであるが．もっと以前に野中さんに SPIN を動かせるようにしていただいていたのに怠慢至極であった．熱海から帰って考え直して書いたものが上である．

正しい利用形態では動くようなので，望みの性質をもつかを確かめることにする．

1. 出鱈目な釦操作で竦みを起こさないか？
2. opened に!locked が先行する．
3. !locked に keq が先行する．

1. の検査のために User0() を書いた．
2. の検査は LTL 式から否定要求を生成して行う．通信路 Lch の交信の順序を確かめるために跡表明もつけてみた．これらの検査については不勉強で SPIN が出す説明が読めていない．みなさまの助けを請わなければならない．

つぎは検査実施の一例である．

```
#define o (ds==true)
#define k (Ks==K)
/*
 * Formula As Typed: <>  o  ->  ( ! k  U  (k  &&   ! o))
 * The Never Claim Below Corresponds
 * To The Negated Formula !(<>  o  ->  ( ! k  U  (k  &&   ! o)))
 * (formalizing violations of the original)
 */

never {    /* !(<>  o  ->  ( ! k  U  (k  &&   ! o))) */
T0_init:
if
:: ((o)) -> goto accept_S8
:: ((k) && (o)) -> goto accept_all
:: ((o)) -> goto accept_S21
:: (! ((k))) -> goto T0_init
:: ((k) && (o)) -> goto accept_S27
fi;
accept_S8:
if
```

```
:: (((! ((k))) || ((o)))) -> goto accept_S8
:: ((k) && (o)) -> goto accept_all
fi;
accept_S21:
if
:: ((o)) -> goto accept_S8
:: ((k) && (o)) -> goto accept_all
:: (((! ((k))) || ((o)))) -> goto T0_init
:: ((k) && (o)) -> goto T0_S27
fi;
accept_S27:
if
:: ((o)) -> goto accept_all
:: (1) -> goto T0_S27
fi;
T0_S27:
if
:: ((o)) -> goto accept_all
:: (1) -> goto T0_S27
fi;
accept_all:
skip
}

#ifdef NOTES
Use Load to open a file or a template.
#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be
 stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.2 -- 12 December 2004)
+ Partial Order Reduction

Full statespace search for:
trace assertion     +
never claim          +
```

```
assertion violations + (if within scope of claim)
acceptance   cycles  + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 40 byte, depth reached 0, errors: 0
       1 states, stored
       0 states, matched
       1 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.000  equivalent memory usage for states (stored*(State-vector + overhead))
0.289  actual memory usage for states (unsuccessful compression: 601500.00%)
State-vector as stored = 288712 byte + 8 byte overhead
2.097  memory used for hash table (-w19)
0.320  memory used for DFS stack (-m10000)
0.152  other (proc and chan stacks)
0.084  memory lost to fragmentation
2.622  total actual memory usage

unreached in proctype Button
line 32, "pan.___", state 4, "D_STEP"
line 35, "pan.___", state 9, "D_STEP"
line 42, "pan.___", state 15, "Bch?D,d"
line 42, "pan.___", state 15, "Bch?C,t"
line 42, "pan.___", state 15, "Bch?L,t"
line 50, "pan.___", state 19, "Lch!unlock"
line 49, "pan.___", state 23, "((Ks==K))"
line 49, "pan.___", state 23, "else"
line 35, state 30, "D_STEP"
line 54, state 36, "Bch?D,d"
line 54, state 36, "Bch?C,t"
line 54, state 36, "Bch?L,t"
line 47, state 38, "((i==2))"
line 47, state 38, "else"
line 32, state 44, "D_STEP"
```

```
line 35, state 48, "D_STEP"
line 65, state 52, "Lch!lock"
line 65, state 53, "Ks = K"
line 64, state 57, "((i==2))"
line 64, state 57, "else"
line 60, state 61, "Bch?D,d"
line 60, state 61, "Bch?L,t"
line 60, state 61, "Bch?C,t"
line 35, state 70, "D_STEP"
line 32, state 76, "D_STEP"
line 72, state 80, "Bch?D,d"
line 72, state 80, "Bch?C,t"
line 72, state 80, "Bch?L,t"
line 70, state 82, "((i==2))"
line 70, state 82, "else"
line 78, state 84, "-end-"
(26 of 84 states)
```

以下省略

振り返って

伊藤，野中両大徳の誘いに乗って楽しい熱海を過ごせたわけですが，みなさまに大変ご迷惑をお掛けしてしまった．おかげで並行プロセスはいい加減には書けないことが身に沁みて解った．

デザインワークショップ2005参加報告

三好 健吾
九州大学大学院　システム情報科学府　情報工学専攻
修士課程2年
TEL：092-642-3865
kengo@ale.csce.kyushu-u.ac.jp

平成17年4月19日

1　ワークショップの要約

本資料は、2005年2月24日(木)～26日(土)にウェルハートピア熱海において、ソフトウェア技術者協会(SEA)が主催したデザインワークショップ2005の参加報告である。

1.1　ワークショップの目的

本ワークショップの目的は、モデル記述言語Promela(Process Meta Language)とその検証ツールであるSPINを実際に使用し、その有効性を検討することである。

1.2　ワークショップの課題

本ワークショップでは、課題として四桁の開錠鍵を持つ物置を題材とし、その物置のモデルを作成し性質を検証することを課題とした。

2　モデルの解説

本ワークショップにおいて私が作成したモデルの解説を行う。なお、付録Aに作成したモデルを掲載する。

2.1　モデル構築の方針

課題となるモデルの作成にあたり、まず最初に山崎利治さんが考案されたステートチャート図を基に、物置を構成要素とその機能で区別した。今回は、物置がボタン入力部、施錠部、ドア部から成ると考え、この3つに区別した。その構成要素それぞれを`Button`プロセス、`Lock`プロセス、`Door`プロセスとしてモデルを構成した。以上の3つのプロセスが完成した後に、物置のユーザの動作をプロセスとして定義した。最後に、モデルの初期状態を`init`プロセスで定義した。

また、通信路として、ボタン入力に関するメッセージのための通信路 Bch、ドアの開閉に関するメッセージのための通信路 Dch、施錠・開錠命令に関するメッセージのための通信路 Lch を定義した。

2.2 各プロセスの説明

2.2.1 Button プロセス

Button プロセスは、ボタンの入力部の動作を意味するプロセスである。このプロセスは、以下の四つの状態を取ると仮定する。

- S1

  施錠され、ユーザが開錠鍵の入力を行う状態

- S2

  施錠され、ユーザから開錠鍵の入力が完了した時に鍵の照合、開錠命令の送信を行う状態

- S3

  開錠され、施錠命令を送信する状態

- S4

  開錠され、ユーザからの入力を受けて開錠鍵の変更を行う状態

このプロセスでは、プロセス起動時に施錠された状態 (lock_lamp == 1) であるなら、状態 S1 に遷移する。S1 においては、入力した開錠鍵が四桁未満ならば鍵の入力の後再度入力の桁を調べ、四桁の入力が完了していれば状態 S2 に遷移する。

状態 S2 に遷移した場合、まず入力された開錠鍵の照合が行われる。入力が保存されている開錠鍵と一致した場合開錠命令を送信する。

開錠された状態 (lock_lamp == 0) であるなら、状態 S3 に遷移し、Lock ボタンが押された場合に施錠命令を送信する。Lock ボタンではなく数字のボタンが押された場合、状態 S4 に遷移し、開錠鍵の変更を行う。

状態 S1、S2 においては、通信路 Lch にメッセージ LOCK が、状態 S3、S4 においては通信路 Lch にメッセージ UNLOCK が受信されず残っている状態である。これらのメッセージが受信されるのは、開錠命令および施錠命令送信の直前である。これにより、他のプロセスの動作に左右されることなく開錠鍵の入力ができるようにしている。

Button プロセスにおける開錠鍵入力操作および入力消去操作は、input_key、clear という inline マクロとして定義されており、プロセス内で呼び出されている。このうち inline マクロ input_key については、数字ボタン以外の入力を受け付けず、Lock ボタン、Clear ボタンが押された場合には inline マクロ clear が実行され、入力を消去するようにしている。

2.2.2 Lock プロセス

Lock プロセスは、鍵の電子的な開錠および電子的な施錠を行うプロセスである。このプロセスは、施錠時である LS1 と開錠時である LS2 という二つの状態を取る。

開錠は、状態 LS1 において通信路 Lch からメッセージ UNLOCK を受信することで行われ、施錠は、状態 LS2 において Lock ボタンが押され、物置の表示窓に何も表示されていない場合に行われる。

2.2.3 Door プロセス

Door プロセスは、ドアの開閉を行うプロセスである。このプロセスは、ドアが閉じた状態 DS1 と開いた状態 DS2 という二つの状態を取る。ドアの開閉はユーザによって行われ、開錠時にユーザを意味するプロセスからのメッセージを受信することで行われる。

2.2.4 User プロセス

物置のユーザの動作を User プロセスとして定義する。今回定義したプロセスは以下の通りである。

- User01

  初期状態の開錠鍵を使って物置の開閉を行うプロセスである。初期状態の鍵"1234"を入力後ドアを開閉し、施錠を行うという動作を繰り返す。

- User02

  開錠鍵の変更を行うプロセスである。まず初期状態の鍵"1234"を入力して開錠し、その後開錠鍵を"5555"に変更する。その後ドアの開閉を行い、施錠する。その後は変更後の鍵を用いて開錠と施錠を繰り返す。

- User03

  鍵の変更を繰り替えすプロセスである。初期状態の鍵"1234"と変更後の鍵"5555"とを交互に使用し、開閉を行う。

- User04

  User03 プロセスの動作に、鍵の入力間違いによるクリアと、鍵の入力途中に誤って Lock ボタンを押すという動作を追加している。

3 SPIN による検証

SPIN を用いて、このモデルの検証を行った。Simulation においては思惑通りの動作が行われていることを確認できた。しかし、Verification においては、検証オプションを変更した場合にデッドロックが発生した。これは、本来想定していない順序でメッセージの送受信が行われたことが原因である。

4 考察

今回のモデル構築の際には、山崎さんが考えられたステートチャートを参考にさせて頂いた。そのため、構成要素の切り分けおよび状態の把握をスムーズに行うことができた。また今回は、atomic、d_step 等の命令を極力使わず、メッセージの送受信のみで危険区域と非危険区域との区別

をつけようと試みた。そう試みた理由は、atomic 等の命令を用いた場合、モデルにおけるデッドロックの危険性を簡単に排除できる反面、SPIN を用いることで検証可能な並行性の問題を発見できないと考えたからである。今回の課題では、3 にも述べた通り、各プロセスを並行に走らせた時に予想外のメッセージの送受信が発生した。このことから、各プロセスにおいて他のプロセスの影響を受けてはならない危険区域の切り分けが不十分だったと考えられる。

5 感想

今回の課題は、残念ながら期限内に never claim による検証を行うことができなかった。この点は非常に反省すべき点である。また、4 でも述べた通り、危険区域と非危険区域の切り分けが不十分であった。今後モデル構築の際には危険区域についても十分な検討を行いたい。

今回 DW2005 に参加させて頂いて、PROMELA の基礎やモデル構築のノウハウを学ぶことができ、非常に有意義な時間を過ごすことができた。

A DW2005 課題 物置の PROMELA モデル

```
/* 定数定義 */

#define N 4             /* 施錠鍵は 4 桁 */
#define OPN 0           /* ドア開いた状態 */
#define CLS 1           /* ドア閉じた状態 */
#define HOR 0           /* 取っ手水平（開いた状態）*/
#define VER 1           /* 取っ手垂直（閉じた状態）*/
#define BLANK -1        /* 表示窓空白 */



/* 大域変数定義 */

 /* データ型 */

int index;              /* 値を表示する表示窓の番号 */
int buff[N];            /* ユーザが指定した鍵を格納する領域 */
int win[N];             /* 表示窓 */
bit lock_lamp;          /* 施錠灯 */
bit door;               /* ドア */
bit handle;             /* 取っ手 */

mtype {D, C, L, LOCK, UNLOCK, OPEN, CLOSE};
 /* メッセージタイプ
        D : 数字の入力                  L : Lock ボタン押下
```

```
        C : Clear ボタン押下              LOCK : Lock プロセスへの施錠命令
    UNLOCK : Lock プロセスへの開錠命令    OPEN : Door プロセスへの開扉命令
    CLOSE : Door プロセスへの閉扉命令 */

 /* 通信路 */

chan Bch = [1] of { mtype, short };          /* ボタン押下時の通信路 */
chan Dch = [1] of { mtype };                 /* ドア開閉時の通信路 */
chan Lch = [1] of { mtype };                 /* 開錠・施錠時の通信路 */

/* inline マクロ */

inline input_key()      /* 鍵入力 */
{       if
        :: (!(Bch?[L(1)]) && !(Bch?[C(1)]) && nempty(Bch))
                    -> Bch?D(win[index]);
                       index++
        :: Bch?[L(1)] -> Bch?L(1);clear()
        :: Bch?[C(1)] -> Bch?C(1);clear()
        fi
}

inline clear()          /* 入力クリア */
{       index = 0;
        do
        :: index != N -> win[index] = BLANK;
                         index++
        :: index == N -> index = 0;break
        od
}

/* プロセス型定義 */

        /* ボタンの入力に関するプロセス Button プロセス */
proctype Button(){
end:do
        :: if
           :: (Lch?[LOCK] && (lock_lamp == 1)) ->
                /* 鍵入力 */
S1:             if
                :: index == N -> index = 0
                :: else -> input_key();
                           goto S1
```

```
                fi;
                /* 鍵の照合・開錠命令の送信 */
S2:             if
                :: index != N -> if
                                 :: win[index] == buff[index];
                                        -> index++;
                                           goto S2;
                                 :: else -> clear();goto S1;
                                 fi;
                :: index == N -> Lch?LOCK;
                   clear();
                   Lch!UNLOCK;
                fi;
   :: (Lch?[UNLOCK] && (lock_lamp == 0)) ->
                /* 施錠命令の送信 */
S3:              if
                 :: Bch?[L(1)] -> Lch?[LOCK] -> Bch?L(1)
                 :: empty(Bch) -> skip
                 :: (!(Bch?[L(1)]) && !(Bch?[C(1)]) && nempty(Bch)) ->
                /* 鍵の変更 */
S4:                        if
                           :: index == N -> index = 0
                           :: index != N -> input_key();
                                            goto S4
                              fi;
                              Bch?[L(1)] -> Bch?L(1);
NEW_KEY:                      if
                              :: index != N -> buff[index] = win[index];
                                               index++;
                                               goto NEW_KEY
                              :: index == N -> clear()
                              fi
                     fi
          fi;
    od
}

   /* 鍵の開錠・施錠に関するプロセス Lockプロセス */
proctype Lock()
{
   /* 電子的な開錠 */
end:do
   :: Lch?[UNLOCK] ->
```

```
LS1:      lock_lamp = 0;
   /* 電子的な施錠 */
      (Bch?[L(1)] &&
       (win[0] == (-1)) && (win[1] == (-1))
        && (win[2] == (-1)) && (win[3] == (-1))) ->
LS2:     Lch?UNLOCK;
         lock_lamp = 1;
         Lch!LOCK
    od
}

   /* ドアの開閉に関するプロセス Door プロセス */
proctype Door()
{  do
   :: if
      :: ((lock_lamp == 0) && Dch?[OPEN]) ->
         /* ドアを開く */
DS1:        door = OPN;Dch?OPEN
      :: ((lock_lamp == 0) && Dch?[CLOSE]) ->
         /* ドアを閉じる */
DS2:        door = CLS;Dch?CLOSE
      fi;
   od
}

/* テストケース  User プロセス */
proctype User01(){
end:do
    :: Bch!D(1);
       Bch!D(2);
       Bch!D(3);
       Bch!D(4);
       handle = HOR;
       Dch!OPEN;
       skip;
       handle = VER;
       Dch!CLOSE;
       Bch!L(1)
    od
}

proctype User02(){
   Bch!D(1);
```

```
    Bch!D(2);
    Bch!D(3);
    Bch!D(4);
    Bch!D(5);
    Bch!D(5);
    Bch!D(5);
    Bch!D(5);
    Bch!L(1);
    handle = HOR;
    Dch!OPEN;
    skip;
    handle = VER;
    Dch!CLOSE;
    Bch!L(1);
end:do
    :: Bch!D(5);
       Bch!D(5);
       Bch!D(5);
       Bch!D(5);
       handle = HOR;
       Dch!OPEN;
       skip;
       handle = VER;
       Dch!CLOSE;
       Bch!L(1)
    od
}

proctype User03()
{  do
   :: Bch!D(1);
      Bch!D(2);
      Bch!D(3);
      Bch!D(4);
      Bch!D(5);
      Bch!D(5);
      Bch!D(5);
      Bch!D(5);
      Bch!L(1);
      handle = HOR;
      Dch!OPEN;
      skip;
      Dch!CLOSE;
```

```
        handle = VER;
        Bch!L(1);
        Bch!D(5);
        Bch!D(5);
        Bch!D(5);
        Bch!D(5);
        handle = HOR;
        Dch!OPEN;
        skip;
        Dch!CLOSE;
        handle = VER;
        Bch!D(1);
        Bch!D(2);
        Bch!D(3);
        Bch!D(4);
        Bch!L(1);
        Bch!L(1)
    od
}

proctype User04()
{  do
    :: Bch!D(1);
       Bch!D(1);
       Bch!C(1);
       Bch!D(1);
       Bch!L(1);
       Bch!D(1);
       Bch!D(2);
       Bch!D(3);
       Bch!D(4);
       Bch!D(5);
       Bch!D(5);
       Bch!D(5);
       Bch!D(5);
       Bch!L(1);
       handle = HOR;
       Dch!OPEN;
       skip;
       handle = VER;
       Dch!CLOSE;
       Bch!L(1);
       Bch!D(5);
```

```
        Bch!D(5);
        Bch!D(5);
        Bch!D(5);
        handle = HOR;
        Dch!OPEN;
        skip;
        handle = VER;
        Dch!CLOSE;
        Bch!D(1);
        Bch!D(2);
        Bch!D(3);
        Bch!D(4);
        Bch!L(1);
        Bch!L(1)
    od
}

init{
    /* 初期状態 */
    index = 0;
    lock_lamp = 1;
    buff[0] = 1;
    buff[1] = 2;
    buff[2] = 3;
    buff[3] = 4;
    win[0] = BLANK;
    win[1] = BLANK;
    win[2] = BLANK;
    win[3] = BLANK;
    Lch!LOCK;
    door = CLS;
    handle = VER;

    run Button();
    run Lock();
    run Door();
/* run User01();
    run User02();
    run User03(); */
    run User04()
}
```

デザインワークショップ DW2005 課題の回答

野中　哲
(有) トゥルーロジック

2005 年 4 月 26 日

1　はじめに

デザインワークショップ 2005(以下 DW2005) は、2005 年 2 月 24 日（木）より 2 泊 3 日の日程で、ウエルハートピア熱海で開催された。日程の前半は、山崎さんによる、モデル検査および PROMELA に関する講義で、後半は事前に示されていた課題 [1] を PROMELA で記述して、SPIN を使用して検証を行った。参加者は 11 名で、このうち学生が 3 名、教育機関から 1 名、その他が一般企業からの参加者であった。

2　モデル作成

課題について PROMELA で作成したモデルのソースコードを付録 A に示す。モデルの簡単な説明は以下の通りである。

1. safe がロッカーのプロセス。全体が 1 つのイベントループになっており、keypad, handle からの入力を契機に処理を行う、イベントドリブン形式になっている。
2. グローバル変数 lock_status が、鍵の状態を表す。施錠灯を兼ねている。
3. グローバル変数 door_status はドアが開いているかどうかの状態を示す。ロックされているが、ドアは閉まっているという状態がある。
4. safe のローカル変数 currentKey が現在ロッカーに設定されている鍵。
5. safe のローカル変数 buf はユーザーのキー入力のバッファで、この値が前面に表示されているものとする。これは長さ 4 のキューで、キーが 4 つ以上入力されると古いものから消えていく。(シフトレジスタのような動作をする)
6. user と safe は 2 つのチャネルでロッカーと交信する。keypad が数字等のキーパッドで、handle が物理的にあけるハンドルである。
7. user... がユーザーを模するプロセス

課題では言及されていないが、モデル作成に必要ないくつかの仕様を仮定した。それらを以下に列挙する。

1. ロックが解けたときは、現在のバッファをクリアする。
2. ドアが開いた状態ではキー入力は無視する。
3. 新鍵設定時に 4 桁に数字が足りないときは、エラーにする。
4. 新しい鍵を設定したときは、現在のバッファをクリアする。

3 検証

3.1 ロッカーが満たすべき仕様

まずロッカーが満たすべき仕様としては、以下のようなものが考えられた。

1. 正しい鍵を入力すると開く
2. 誤った鍵を入力すると開かない
3. 新しい鍵を設定できる
4. 新しい鍵を設定すると、新しい鍵で開く
5. 新しい鍵を設定すると、古い鍵では開かない

SPIN による検証にはシミュレーションモードとベリフィケーションモードというものがある。前者では、並列プロセスをランダムにインターリーブさせながら実行を行い、後者では並列プロセスが取りうる全ての状態で問題が発生しないかどうか検証をする。

3.2 正しい鍵を入力するとドアをあけることができるか

最初に正しい鍵を入力するユーザープロセスを定義し、ドアを開けられるかどうかの検証を行った。これはシミュレーションモードで検証を行ったが、ここで気が付いたのは、ドアを正しく開けるにはロッカープロセスとユーザープロセスの間で正しく同期をとらなければいけなという点である。

ロッカーの正しいあけ方

1. 正しいキーを入力
2. ユーザーは施錠灯が消える事（解錠）を確認（ここで同期をとる）
3. レバーを回す

すなわち正しい鍵を入力しても、それを受け取ったロッカーの処理が完了し、解錠されなければ、レバーを回してもドアは開かないのである。これは、実際にシミュレーションを行った結果ドアが開かないというケースがある事がわかり問題を発見した。ロッカーと人間の間では、現実的に実際にこのような問題が発生する事は少ないであろうが、一般的なソフトウエア設計のケースでは、このように並列プロセス間の同期は充分注意が必要なので、モデルチェッキングの有効性を感じた。

3.3 誤った鍵でドアが開く事はないか

次におこなったのは、誤った鍵を入力したにもかかわらずドアが開いてしまうことはないか、という点の検証である。これには SPIN のベリフィケーションモードを用いた。まず「**ドアは永遠に開かない**」という性質の記述は、TLT 式を用いて次のように記述した。

```
□!p
#define p (lock_status == K_UNLOCKED)
```

次に誤った鍵しか入力できないユーザーとして、「ランダムに数字キーを押すが、4 だけは絶対に押さないユーザープロセスを定義した。ロッカーの鍵の初期値は「1234」であるのでこのユーザープロセスは、どう頑

張っても正しい鍵は入力出来ない。したがってドアは開かないはずである。

```
proctype user5(){
 do
   :: keypad ! NUM(0);
   :: keypad ! NUM(1);
   :: keypad ! NUM(2);
   :: keypad ! NUM(3);
   :: keypad ! NUM(5);
   :: keypad ! NUM(6);
   :: keypad ! NUM(7);
   :: keypad ! NUM(8);
   :: keypad ! NUM(9);
   :: keypad ! LOCK(0)
 od
};
```

以上のように定義を行ったのち、SPIN のベリフィケーションモードで検証を行った。実行結果は以下である。

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
pan: claim violated! (at depth 375)
pan: wrote pan_in.trail
(Spin Version 4.2.1 -- 8 October 2004)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     - (not selected)
        invalid end states      - (disabled by never claim)

State-vector 84 byte, depth reached 375, errors: 1
<<<以下省略>>>
```

最初のモデルでは検証は失敗した。すなわち「ドアは永遠に開かない」という表明は否定され、反例が発見されたのである。これは「正しくない鍵でもドアが開いてしまう」ということを意味する。SPIN では、この時見つかった反例の動作ーシーケンスが出力され、この通りにシミュレーションを再実行することで判例を再現することが出来る。これによってわかった事は、仕様の理解に不十分な点があったという点である。すなわち

- 新しい鍵を設定できる

という仕様は実は誤りで、正しくは

- ロックされているときは、新しい鍵は設定できない
- ロックされていないときは、新しい鍵を設定できる

でなければいけない。そうでなければ、現在ロッカーが施錠中でその鍵の値を知らなくても、新しい鍵の設定方法さえ知っていれば、施錠中のロッカーに新しい鍵を設定(上書き)して、その鍵を使いドアを開けることが出来てしまう。これではロッカーの役目を果たさない。課題の文章の中には「施錠灯が消えているときに,つぎのようにして好みの鍵にすることができます」と記述されているが、下線の条件部分に対する考慮がすっぽりと抜け落ちていたわけある。

問題の原因がわかったので「新しい鍵を設定できるのは鍵が開いている場合のみ」という条件チェックをモデルへ追加した。これで再度検証を行い次の出力が得られた。

```
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
(Spin Version 4.2.1 -- 8 October 2004)
+ Partial Order Reduction

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

State-vector 84 byte, depth reached 32817, errors: 0
<<<以下省略>>>
```

このように、表明が正しいことが確認され、不正な鍵ではドアは開かないことが確認された。今回の課題について行った検証は以上である。

4 演習を通じての感想

4.1 仕様作成時における誤りについて

前セクションでとりあげた「新しい鍵を設定できるのは鍵が開いている場合のみ」という部分の仕様の誤りについて大変興味深い例だと思った。これはモデル作成時のコーディングエラーではなく、明らかに仕様作成時の誤りである。[*1]そしてモデルはその**誤った仕様どおり作成されている。**このため、仕様の誤りがそのままモデルにも引き継がれたわけである。従って実際に製品の製造を行い、仕様書に沿ったテストのみしか行わなければ、この欠陥はテスト工程でも発見できない。市場に製品が出荷され実際にユーザーが使用してみて初め

[*1] 今回の演習ではモデル作成時に物理的に仕様書を実際に書くという行為は行わなかったが当然意識の中では仕様書は存在する。

て問題があきらかになるというわけだ。これはもちろん実際の製品ならば巨額の損失を引き起こす可能性をはらんでいる。このような仕様作成時のエラーが LTL 式を利用した仕様の検証で発見できたということは、モデルチェッキングの有効性を示す大きな例だと感じた。

4.2 形式手法に関する誤解

形式的手法、時間論理などといいう言葉からは、なにかしら作成したモデルや仕様が数学的で完全であるという印象を受ける。それらの手法を使うと、まるで魔法のように完璧なモデルや仕様書ができてしまうような錯覚にとらわれがちであるが、このような簡単な演習を通じてでも、それがまったくの幻想であることが良くわかる。

モデルをかいてみたものの、それらを検証する段階になると、これらをどうやって体系的、網羅的に検証していくかを考えることは、試行錯誤の連続で、手探りの作業であった。やはり、新たにものを作るという作業は経験と知識とひらめきを用いて、迷い苦しまないと達成できない。製品開発のための汎用的解法やアルゴリズムが存在するわけではないのである。以前 SIGFM のフォーラムで九州大学の荒木先生が紹介されていた「形式的手法に関する 7 つの神話」[2] にも同様の指摘があったように記憶している。

形式的手法をつかうと対象を出来る限り厳密に記述しようという努力が要求されるが、この過程が非常に重要なのではないかと思う。出来上がった結果のモデルが完全無欠のものであるという保障はどこにもないが、形式的なモデルを作り上げていく過程では、多くの問題や考慮の抜けを発見できる。形式的モデルを作成することは、いままで自然言語行や、その他の図法、記法などのツールを使用して行ってきた仕様作成とは、また別の角度から、開発の対象システムに光を当て眺める機会を得ることである。モデルチェッキングのみで全ての問題が解決されるというのは全くの幻想であるが、既存の手法やツールを補完し、それらでは発見が難しかった問題点の早期発見につながることも事実である。形式手法に対する過大な期待が払拭される一方、その有効性も実感できたのは大きな収穫であった。

4.3 プロセスの並列性について

私の書いたロッカープロセスは、いわば一つの巨大なイベントループになっており、ロッカー内では並列性はない。私が問題を一見したときの印象では、これはごく自然な設計のアプローチであった。これは、私が長年 Macintosh のユーザーアプリケーションを開発してきた経験と無関係ではないと思う。Window システム上のアプリケーションでは、イベントループを中心に置いた、イベントドリブン形式のコーディングスタイルになるのはごく自然である。モデルを詳細化してゆく間に、このメインイベントループが、どんどん巨大化してしまい、読みにくい、扱いにくいという不満を感じてはいたが、これは主に PROMELA ではサブルーチンが書けないためであると考えていた。[*2] このため、ワークショップで、他の人たちが、ロッカーをもっと細かいプロセスに分解しているモデルにしているのを見たときは、新鮮な驚きを感じた。しかしながら、振り返ってみると、そもそも `user` と `safe` の間の通信チャネルを `handle` と `keypad` の二つに分離したのは、キーパッドとハンドルは独立しているので、並列に操作可能という事が念頭にあった。それにもかかわらず、ロッカーを 1 つのプロセスにしてまってはなんのために、2 つを別のチャネルにしたのかわからない。この部分に関しては、次の機会があれば、きちんと考慮するようにしたいと思う。

[*2] これはモデルが詳細に過ぎる事が無いよう意図的にこのように設計されているということだ。

4.4 抽象化ということ

ホルツマンの教科書 [3] にも指摘されているが、初心者の犯しやすい誤りのうち最も多いものは「詳しすぎるモデルを書いてしまう」という事であるという。私の書いたモデルにも、これは当てはまる。数字キーが 10 個あるという事や、鍵長が 4 であるという事は、モデルについては本質的ではない。当然キーの種類が少なく、また長さも短い方が検証の際は、時間もメモリ使用量も節約できる。この点については改善の余地が大いにあると思う。[*3]

4.5 その他

これはモデルチェッキングとは直接関係はないことであるが、シミュレーションをしている最中に、ロッカーの仕様に関して次のようなことを考えた。ユーザーは、ロックが解けたのに気がつかず、ひたすらキーを押し続けるかもしれない。このような行為が続くと、気がつかないうちに新しい鍵でロックしてしまう可能性もある。そうすると、このユーザーは（おそらく）新しく鍵を設定したことを意識していないので、その鍵がわからず、2 度とロッカーを開けられなくなってしまう。これを防ぐためには「新しい鍵を設定するためには、ロックが解除され、ハンドルを回して一度ドアを開く必要がある」という仕様を追加するが良いのではないだろうか。このような点に気がつくのは非決定的な実行が記述でき、それをランダム実行するシミュレーションが可能な SPIN/PROMELA の隠れたご利益と言える。

[*3] ただこの点に関しては何も考えずに作ったとき、どのあたりまで検証可能なのかツールの実力を試してみたい、という気持ちもあった。当然なにも考えないと、すぐにメモリー不足で検証不能という結果にはなったが。

付録 A　ロッカーのモデルの PROMELA コード

```
/* assignment for DW2005 $Id: dw2005.pml,v 1.7 2005/02/01 05:22:11 nonaka Exp $ */

#define K_OPEN 0
#define K_CLOSED 1

#define K_UNLOCKED 0
#define K_LOCKED 1


#define KEYLENGTH 4
mtype = {NUM,CAN,LOCK};

chan keypad = [0] of { mtype, int};
chan handle = [0] of { bit }
bit lock_status = K_LOCKED; /* initial state locked, means locked */
bit door_status = K_CLOSED; /* initial state door closed  */

inline clear_buf(){
  d_step {
    buf_len = 0;
    buf[0] = -1;
    buf[1] = -1;
    buf[2] = -1;
    buf[3] = -1
  }
}

inline open_the_door(){
  lock_status == K_UNLOCKED;
  assert(lock_status == K_UNLOCKED);
  handle ! K_OPEN;
  door_status == K_OPEN;
  assert(door_status == K_OPEN);
}

proctype safe(){
```

```
  int key;
  int buf[KEYLENGTH];
  int buf_len = 0;
  int currentKey[KEYLENGTH]; /* initial key value is "1234" */
  d_step {
    currentKey[0] = 1;
    currentKey[1] = 2;
    currentKey[2] = 3;
    currentKey[3] = 4;
  }
end0:
  do
    :: keypad ? NUM(key) ->
       d_step {
 printf("Number input:%d.\n",key);
 if
   :: door_status == K_CLOSED ->
      /* the door is close, accumulate keys into buf */
      /* display them on LED */
      buf[0] = buf[1];
      buf[1] = buf[2];
      buf[2] = buf[3];
      buf[3] = key;
      if
:: buf_len <= 3 -> buf_len++;
:: else -> skip
      fi;
      if
:: lock_status == K_LOCKED ->
   /* the door is currently locked */
   if
     :: ((buf[0] == currentKey[0]) && (buf[1] == currentKey[1]) &&
 (buf[2] == currentKey[2]) && (buf[3] == currentKey[3]))
-> lock_status = K_UNLOCKED;
printf("Unlocked.\n");
clear_buf() /* light is off [open] */
     :: else -> skip
   fi
:: else -> skip
      fi
```

```
   :: door_status == K_OPEN ->
      /* if the door is open, ignore the key input */
      printf("Door open and key input ignored.\n");
      skip
 fi
       }
       /* accumulate the data into the buffer */
    :: keypad ? CAN(_) ->
       /* clear the buffer */
       clear_buf()
    :: keypad ? LOCK(_) ->
       d_step {
 printf("Lock Key pressed.\n");
 if
   :: lock_status == K_UNLOCKED ->
      /* Lock the door and set new key */
      if
:: buf_len == 0 -> {
  /* use current key and lock */
  lock_status = K_LOCKED;
  printf("Locked with the current Key.\n")
}
:: buf_len == KEYLENGTH /* set net key and lock */
   -> {
     currentKey[0] = buf[0];
     currentKey[1] = buf[1];
     currentKey[2] = buf[2];
     currentKey[3] = buf[3];
     lock_status = K_LOCKED;
     clear_buf();
     printf("Locked with the new key.\n")
   }
:: else -> {
  printf("Error: insufficient new key. Try again!\n");
  clear_buf();
}
      fi
   :: else -> skip
 fi
       }
```

```
    :: handle ? K_CLOSED ->
       /* handle closed */
       d_step {
 if
   :: door_status == K_OPEN -> {
     door_status = K_CLOSED;
     printf("Door closed.\n");
   }
   :: else -> skip
 fi

       }
    :: handle ? K_OPEN ->
       /* handle \opened
        check if unlocked
if off, open the door, clear the buffer */
       d_step {
 if
   :: (lock_status == K_UNLOCKED) -> {
     door_status = K_OPEN;
     printf("Door opened.\n");
   }
   :: else -> skip
 fi
       }
  od
}

init {
  run user5();
  run safe();
}
```

参考文献

[1] 山崎利治,DW2005 のための問題,2005

[2] J. A. Hall, Seven Myths of Formal Methods, IEEE Software, Vol.7, No.5, pp.11-19, 1990.

[3] Gerard J. Holzmann,THE SPIN MODEL CHECKER Addison-Wesley, 2004

デザインワークショップ2005参加報告

九州工業大学　堂園 隼人

２００５年４月２０日

ソフトウエア技術者協会(SEA)主催のデザインワークショップ2005(以下DW2005)の参加報告を行う。

1. DW2005の概要

開催日時：2005年2月24日（木）～26日（土）
場所：ウェルハートピア熱海
テーマ：ソフトウエア仕様検証技術の最新動向 -- PROMELA/SPINを用いて

2. 報告の概要

本ワークショップでは、実時間分散システムの設計および検証を行うことのできるモデルチェッカSPIN[1]を味見してみた。

ワークショップにあたり、山崎氏から事前に課題として、簡単な電子錠が付いた「物置」[2]が提示された。この物置は、電子錠、入力キー、取っ手、表示窓、施錠灯、ドアから構成されている。これらをプロセスに割り付け、PROMELA（PRocess MEta LAnguage）によりモデルを記述し、SPINを用いて検証を行った。

ワークショップでは、山崎氏より講義を受けた後、グループで演習を行い、最終日に発表と意見交換を行った。

本報告では、ワークショップ後に改めて作成したモデルについて報告する。

3. 作成したモデル

3.1. プロセス

検証モデルを作成する上で、物置の各部分をプロセスに割り付けた。物置自体をあらわすシステム内部のプロセスを表1に、環境（ユーザ）をあらわすシステム外部のプロセスを表2に示す。

表 1 システム内部のプロセス

プロセス名	役割
keypad	数字キーの入力をまとめた 4 桁の数字と、Lキーを e_key に送るプロセス
display	表示窓と施錠灯のプロセス
door	扉のプロセス
e_key	電子錠のプロセス

表 2 システム外部のプロセス

プロセス名	役割
key_pusher	keypad に入力を送る
door_opener	door にドアの操作を送る

3.2. チャネル

プロセス間の通信を行うために、表3のようにチャネルを設定した。なお、これらのチャネルはすべて同期チャネルとして設定した。

表 3 チャネル

チャネル名	送信元	送信先	送信内容
n	key_pusher	keypad	"1 桁の数字"、"C キー"、"L キー"
handle	door_opener	door	"取っ手の開閉"
d	keypad,e_key	display	"表示する数字"、"施錠灯の点消灯"
f	keypad	e_key	"入力された 4 桁の数字"

3.3. 共有変数

メッセージ型や局所変数として設定せず、大域変数として設定したものは以下の２つである。これらは、チャネルによる通信で表現するとモデルの記述が冗長になるため、大域変数として設定した。

・扉の状態
・電子錠の状態

3.4. 局所変数

局所変数については、付録の中で説明している。

4. 独自の仕様

今回、モデルを作成する上でいくつかの独自の仕様を導入した。

4.1. 要求仕様であいまいだった点

- 入力が１～３桁のときＬキーが押されたら、表示を消して終わる。
- 数字キーの入力は取っ手の操作により中断されない。

4.2. その他の点

- シミュレーションで、電子錠の最初のキーワードがマッチしやすいように、１～９、C、Ｌキー以外に"1234"を送るようにした。
- 簡単化のために、取っ手を回す動作と開閉の動作をまとめた。

5. キー入力に対する keypad の動作

上記の仕様の追加により、キー入力に対する keypad の動作は以下のようになる。

(ア) 数字キーの入力

① 数字が入力されると keypad は現在の入力値について覚えておく。

② 4 桁目の数字が入力されたら、e_key に 4 桁の数字をまとめて送り、キーワードとの一致を調べる。

③ 4 桁の数字の送信後、表示をクリアして、①に戻る。

(イ) C,L キーの入力

- C キーが押されると、いつでも入力をクリアする。
- Ｌキーが押されると、入力された数字が０，４桁のときはＬキーが押されたことを e_key に送る。１～３桁のときには、e_key には何も送らず入力をクリアする。

5. 調べた項目

作成したモデルに対して以下の項目について検査を行い、妥当性を確認した。

- 「ロック状態で正しいキーワードが入力されたら、アンロック状態になる」
- 「扉が開いている間は、物置はロックされない」
- 「デッドロック状態には陥らない」
- 「アンロック状態のとき、１～３桁の数字を入力してＬボタンを押すと、数字が消えるだけである」

6. DW２００５以前と以後のモデル

DW2005 に参加するにあたり事前に作成したモデルでは、倉庫と環境をそれぞれ１つのプロセスとして設定していた。それは、倉庫全体を１つの状態推移系として容易にイメージすることができたからである。

DW2005 を終えて、本報告を作成するにあたりモデルを再構築した。その際、オブジェクト間の相互作用を見るために、倉庫をいくつかの子プロセスに分割したり、変数をできるだけ局所変数として設定したりするなどの変更を行った。

7. まとめ

ワークショップを通して、PROMELA によるモデルの構築と、その検証ツールである SPIN への理解を深めることができた。

今回のワークショップのように、演習を行ったり、いろいろな方々の作成されたモデルを見比べ、意見を交わしたりする機会は大変有意義なものであった。

付録　作成した Promela モデル

```
#define DOOR_OPEN 1
#define DOOR_CLOSE 0
#define EKEY_UNLOCK 1
#define EKEY_LOCK 0
#define LAMP_ON 1
#define LAMP_OFF 0

mtype = {inputNum,c,l};          /*キー入力*/
mtype = {redisplay,lamp};        /*表示*/
mtype = {sOpen,sClose};          /*ドアの開閉*/
mtype = {num4};                  /*4 桁の数字*/

/*共有変数*/
show int door_state = DOOR_CLOSE; /*ドアの状態*/
show int e_key_state = EKEY_LOCK; /*電子錠の状態*/

/*チャネル*/
chan d = [0] of {mtype,int};     /*表示・ランプ用*/
chan n = [0] of {mtype,int};     /*キー入力用*/
chan handle = [0] of {mtype};    /*取っ手用*/
chan f = [0] of {mtype,int};     /*入力数字 4 個組み,L ボタン*/

/*キー入力を受け取るプロセス*/
active proctype keypad(){
int i;              /*入力*/
int all_nums;       /*全入力*/
int num_point;      /*入力総数*/

zero:
skip;
d_step{
  i = -1;
  all_nums = -1;
  num_point = 0;
}
```

```
  d!redisplay(all_nums);

if
::n?inputNum(i) -> all_nums = i; num_point++;
                   d!redisplay(all_nums); goto one;
::n?c(_) -> goto zero;
::n?l(_) -> f!l(-1); goto zero;
fi;

one:
if
::n?inputNum(i) -> all_nums = all_nums * 10 + i;
                   d!redisplay(all_nums); num_point++; goto two;
::n?c(_) -> goto zero;
::n?l(_) -> goto zero;
fi;

two:
if
::n?inputNum(i) -> all_nums = all_nums * 10 + i;
                   d!redisplay(all_nums); num_point++; goto three;
::n?c(_) -> goto zero;
::n?l(_) -> goto zero;
fi;

three:
if
::n?inputNum(i) -> all_nums = all_nums * 10 + i;
                   d!redisplay(all_nums); num_point++;
                   f!num4(all_nums); goto zero;
::n?c(_) -> goto zero;
::n?l(_) -> f!l(-1); goto zero;
fi;
}
```

```
/*表示窓・施錠灯のプロセス*/
active proctype display(){
int d_num = -1;              /*表示窓の数字*/
bool lock_lamp = 1;          /*施錠灯*/

end:
do
::d?redisplay(d_num) ->
  if
  ::(d_num < 0)   -> printf("¥n");
  ::(d_num >= 0) -> printf("%d¥n",d_num);
  fi;
::d?lamp(LAMP_ON) -> lock_lamp = LAMP_ON;
::d?lamp(LAMP_OFF) -> lock_lamp = LAMP_OFF;
od;
}

/*電子鍵のプロセス*/
active proctype e_key(){
int keyword = 1234;          /*現在のキーワード*/
int input = -1;              /*keypad から送られた 4 桁の数字*/

close:
e_key_state = EKEY_LOCK;
if
::f?num4(input) -> if
  ::(input == keyword) -> d!lamp(LAMP_OFF); goto open;
  ::(input != keyword) -> goto close;
  fi;
::f?l(_) -> goto close;
fi;

open:
e_key_state = EKEY_UNLOCK;
input = -1;
```

```
open2:
if
::f?num4(input) -> goto open2;
::f?l(_) -> if
  ::(door_state == DOOR_OPEN) -> goto open;
  ::(door_state == DOOR_CLOSE) -> d!lamp(LAMP_ON); if
    ::(input >= 0) -> keyword = input; goto close;
    ::(input < 0)  -> goto close;
    fi;
  fi;
fi;
}

/* ドアのプロセス*/
active proctype door(){
close:
door_state = DOOR_CLOSE;
if
::handle?sOpen  -> if
  ::(e_key_state == EKEY_UNLOCK) -> goto open;
  ::(e_key_state == EKEY_LOCK) -> goto close;
  fi;
::handle?sClose -> goto close;
fi;

open:
door_state = DOOR_OPEN;
if
::handle?sOpen  -> goto open;
::handle?sClose -> goto close;
fi;
}
```

```
/*キーを入力するプロセス*/
active proctype key_pusher(){
do
::atomic{
/*最初のキーワード 1234 がシミュレーションで
早くおこるようにするため*/
   n!inputNum(1);
   n!inputNum(2);
   n!inputNum(3);
   n!inputNum(4);
}
::n!inputNum(1);
::n!inputNum(2);
::n!inputNum(3);
::n!inputNum(4);
::n!inputNum(5);
::n!inputNum(6);
::n!inputNum(7);
::n!inputNum(8);
::n!inputNum(9);
::n!inputNum(0);
::n!c(-1);
::n!l(-1);
od;
}

/*ドアを開閉するプロセス*/
active proctype door_opener(){
do
::(door_state == DOOR_CLOSE) -> handle!sOpen;
::(door_state == DOOR_OPEN) -> handle!sClose;
od;
}
```

参考文献

[1] Gerard J. Holzmann. The Spin Model Checker: Primer and Reference Manual.
[2] 山崎 利治　DW2005 のための問題
[3] 山崎 利治　ソフトウエア仕様検証技術の最新動向　PROMELA／SPIN を用いて

デザインワークショップ2005 参加報告

張　暁晶
九州大学 大学院システム情報科学府 情報工学専攻
修士課程1年
TEL : 092-642-3869
zhang@ale.csce.kyushu-u.ac.jp
2005 年4 月4 日

一、ワークショップの要約

ソフトウェア技術者協会(SEA)主催のデザインワークショップ 2005 の参加報告である。ワークショップでは、金庫を模型検査言語 PROMELA により模型化し、SPIN というツールで模型検査する演習を行った。演習では、参加者が 3 チームに分かれてそれぞれ模型を構築した。張は、同チームの（株）ニルソフトの伊藤さんと（株）デンソーの中里さんとともに金庫の模型化に挑戦した。

伊藤チームでは、模型化する対象の本質を見出し、まずそこだけを模型として構築した。金庫の場合においてはドアの開閉と鍵の開閉が一番肝要だと考え、最初の模型をドアと鍵の二つのプロセスにより構成した。演習後に張がチームメンバの指導の下、この模型にパスワードの照合や、ボタン入力などの機能を付け加え、問題文に近い金庫の模型を作り上げた。以下に作成した金庫模型を説明する。なお、PROMELA 模型は付録に添付する。

二、模型の解説

張が作成した模型では、金庫の要素を大きく鍵、ドア、液晶表示窓の三つに分けた。それぞれを独立したプロセスとして作成した。

まず液晶表示窓プロセス lcd はユーザのボタン入力を受け付ける。つまり液晶表示窓を表す変数 window を入力バッファとして使っている。このプロセスはボタン入力の通信路 chanbutton からメッセージを受信し、そのメッセージが数字ボタンが押されたことによるものであれば、バッファにためていき、CLEAR ボタンによるものであればバッファをクリアする。LOCK ボタンによるもので、バッファが 4 桁とも空であれば「施錠しろ」イベントを起こし、4 桁とも埋まっていれば「パスワードを変更しろ」のイベントを起こす。ただし入力を受け付けるのはドアが閉まっている時のみである。

次にドアプロセス door はドアの状態を変更する。ユーザからドアの状態を変更するイベントが発生した時イベントの種類、すなわち開けるか閉めるかによってドアの状態を変化させる。通信路 chandoor から、「ドアを開けろ」イベントを受信すると、ドアが閉まっている状態でかつ鍵が開いていることを確認して、ドアの状態を開いている状態に変更する。「ドアを閉めろ」イベントが来た時も同様であるが鍵の確認はしない。

次に鍵プロセス key は鍵の状態を変更する。通信路 chanlock から lcd プロセスが起こした３種類のイベントを受信する。まず「開錠しろ」イベントが来るとドアが閉まっていて鍵が閉まっていることを確認して、次に入力バッファとパスワードを照合して一致すれば鍵の状態を開いている状態に変更する。一致しなかったり入力桁数が足りない場合は何もしない。次に「施錠しろ」イベントが来た場合ドアが閉まっていて鍵が開いていることを確認して、LOCK ボタンが押されたら鍵の状態を閉まっている状態に変更する。最後に「パスワードを変更しろ」イベントが来た場合、ドアが閉まっていて鍵が開いていることを確認して、パスワードを入力されたものに変更して鍵の状態を閉まっている状態にする。

以上が金庫の構成要素のプロセスで、外界から金庫に命令する、つまり通信路を通じて信号を送る実体として、ユーザプロセスを設けた。ユーザは、「ドアを開けろ」、「ドアを閉めろ」のイベント、数字ボタンと CLEAR ボタンと LOCK ボタンを押す動作を、ランダムにいずれかを起こすことができる。イベントというものをユーザの意思として捕らえており、いずれも inline 関数として定義した。

上記のモデルを SPIN を使ってシミュレーションした結果、期待した動作が行われた。

「ドアが開いている状態と鍵がかかっている状態は同時に成立できない」という性質を検証しようと Never Claim を記述したが、SPIN による Verification では開始直後にシミュレーションが停止し、モデルにまだバグが残っていることが確認できた。今後の作業としては、Never Claim に抵触しないようモデルを修正する必要があると思われる。

三、模型検査の結果

今回 XSPIN という GUI の SPIN を用いて作成した金庫模型の模型検査を行った。SPIN ではシミュレーションと Verification ができ、シミュレーションでは模型の一通りの動作（確率的なだが）を見ることができる。また、Verification では模型がある性質を満たしているかどうかを検証できる。

今回の模型をシミュレーションにかけた結果、ほぼ要求を満たす動作をしていたと確認できた。ユーザプロセスの最初で 1234 と登録させているパスワードを入力させている。その結果金庫が開錠されたことが確認できた。また、開錠された状態で LOCK ボタンを押した場合金庫が再び施錠されることも確認できた。鍵がかかっていない状態でユーザがドアを開けるとドアが開いたことも確認できた。

Verification のほうでは、模型の満たすべき性質として、「ドアが開いている状態と、鍵がかかっている状態は同時に起こりえない」という性質を SPIN で検証しようとしたが、返された結果として、今回作成した模型ではそのようなことは起こりうるとあった。これは模型に欠陥が存在することを意味している。張の経験不足により現在のところはまだこの欠陥を修正するに至っていない。

四、感想

今回のワークショップで初めてPROMELA言語とSPINを使って模型検査を体験した。また、今回の演習で模型検査のための模型の作り方を主に考えてきた。模型のモジュール化にプロセスという概念を用いているところに馴染みがなかったので、モジュールの切り分け方が難しいと感じた。そのため、模型化の対象の本質を明確にするため、シンプルな模型を作成し後に詳細化する手法を取り入れた。

付録：張が作成した金庫模型

```
#define KEY_LOCKED 1    /* 鍵が閉まっている */
#define KEY_UNLOCKED 0  /* 鍵が開いている */
#define DOOR_CLOSE 1    /* ドアが閉まっている */
#define DOOR_OPEN 0     /* ドアが開いている */
#define KEY_LENGTH 4    /* パスワードの桁数 */

/* 以下 never claim 用 */
#define open (door_status==DOOR_OPEN)    /* ドアが開いている状態 */
#define locked (key_status==KEY_LOCKED)  /* 鍵が閉まっている状態 */

mtype={NUM,C,L};
/* 信号の型は数字ボタン、CLEAR ボタン、LOCK ボタン */



chan chandoor=[10] of {int};          /* ドア状態を流す通信路 */
chan chanlock=[10] of {int};          /* 鍵状態を流す通信路 */
chan chanbutton=[200] of {mtype,int}; /* ボタン入力を流す通信路 */

show bit door_status = DOOR_CLOSE;
/* ドアの状態（初期値は閉まっている） */

show int E_changedoor = 0;
/* ドア状態変更に関するイベント */
/* 0：イベントなし */
/* 1：「ドアを閉めろ」イベント発生 */
/* 2：「ドアを開けろ」イベント発生 */

show bit key_status = KEY_LOCKED;
```

```
/* 鍵の状態（初期値は閉まっている） */

show int E_changelock = 0;
/* 鍵状態変更に関するイベント */
/* 0：イベントなし */
/* 1：「施錠しろ」イベント発生 */
/* 2：「開錠しろ」イベント発生 */
/* 3：「パスワード変更しろ」イベント発生 */

show int window[KEY_LENGTH];
/* 液晶表示窓（ユーザ入力バッファ） */
/* 各桁には-1（無表示）もしくは0-9が入る */

show int password[KEY_LENGTH];
/* 金庫に設定されているパスワード */

inline opendoor(){ /* ドアを開ける */
   d_step {
      E_changedoor = 2;
      chandoor!E_changedoor;
      /* 「ドアを開けろ」イベントの発生を通信路chandoorに通知 */
      E_changedoor = 0;
   };
};

inline closedoor(){ /* ドアを閉める */
   d_step {
      E_changedoor = 1;
      chandoor!E_changedoor;
      /* 「ドアを閉めろ」イベントの発生を通信路chandoorに通知 */
      E_changedoor = 0;
   };
};

inline sejyo(){ /* 施錠する */
   d_step {
```

```
        E_changelock = 1;
        chanlock!E_changelock;
        /* 「施錠しろ」イベントの発生を通信路 chanlock に通知 */
        E_changelock = 0;
    };
};

inline kaijyo(){ /* 開錠する */
    E_changelock = 2;
    chanlock!E_changelock;
    /* 「開錠しろ」イベントの発生を通信路 chanlock に通知 */
    E_changelock = 0;
};

inline henko(){ /* パスワードを変更して施錠する */
    E_changelock = 3;
    chanlock!E_changelock;
    /* 「パスワード変更しろ」イベントの発生を */
    /* 通信路 chanlock に通知 */
    E_changelock = 0;
};

inline clear(){ /* 液晶表示窓をクリアする */
    d_step {
        window[0] = -1;
        window[1] = -1;
        window[2] = -1;
        window[3] = -1;
    };
};

active proctype lcd() priority 100{ /* 液晶表示窓のプロセス */
    int but;
    do
    ::if
    ::chanbutton?NUM(but) ->    /* 数字ボタンが押されていたら */
```

```
        d_step {
            if
            :: door_status == DOOR_CLOSE ->
                window[0] = window[1];
                window[1] = window[2];
                window[2] = window[3];
                window[3] = but;
                if
                ::(window[0]!=-1&&window[1]!=-1&&window[2]!=-1&&window[3]!=-1)->
                    kaijyo();
                ::else->skip;
                fi
            :: door_status == DOOR_OPEN -> skip;
            /* ドアが開いている時は入力を無視 */
            fi;
        };
    ::chanbutton?C(0) ->    /* CLEAR ボタンが押されていたら */
        clear();
    ::chanbutton?L(0) ->    /* LOCK ボタンが押されていたら */
        if
        ::(window[0]!=-1&&window[1]!=-1&&window[2]!=-1&&window[3]!=-1)->henko();
        ::(window[0]==-1&&window[1]==-1&&window[2]==-1&&window[3]==-1)->sejyo();
        ::else->clear();
        fi;
    fi;
    od;
}

active proctype door() priority 100{ /* ドアのプロセス */
    int x;
OPEN: /* ドアが開いている状態 */
    do
    ::chandoor?x ->
        if
        :: (x==1 && door_status==DOOR_OPEN) ->
            door_status=DOOR_CLOSE;goto CLOSE;
```

```
        /* 「ドアを閉めろ」イベントが発生したら開いているドアを閉める */
        :: else -> skip;
        fi;
    od;
CLOSE: /* ドアが閉まっている状態 */
    do
    ::chandoor?x ->
        if
        :: (x==2 && key_status==KEY_UNLOCKED &&
            door_status==DOOR_CLOSE) ->
            door_status=DOOR_OPEN;goto OPEN;
        /* 「ドアを開けろ」イベントが発生したら */
        /* 鍵が開いていれば、閉まっているドアを開ける */
        :: else -> skip;
        fi;
    od;
);

active proctype key() priority 100{ /* 鍵のプロセス */
    int x;
    int but;
    d_step {/* 初期パスワードは1234 */
        password[0] = 1;
        password[1] = 2;
        password[2] = 3;
        password[3] = 4;
    };
LOCK: /* 鍵が閉まっている状態 */
    do
    ::chanlock?x ->
        if
        :: (x==2 && door_status==DOOR_CLOSE && key_status==KEY_LOCKED) ->
        /* 「開錠しろ」イベントが発生したら */
        /* ドアが閉まっていて鍵も閉まっていればパスワード照合に進む */
            if
            ::(window[0] == password[0] &&
```

```
            window[1] == password[1] &&
            window[2] == password[2] &&
            window[3] == password[3]) ->
            /* 入力がパスワードと一致したら開錠する */
               key_status = KEY_UNLOCKED;clear();goto UNLOCK;
         ::else -> skip
         fi;
      :: else -> skip;
      fi;
   od;
UNLOCK: /* 鍵が開いている状態 */
   do
   ::chanlock?x ->
      if
      ::(x==1 && door_status==DOOR_CLOSE && key_status==KEY_UNLOCKED) ->
      /* 「施錠しろ」イベントが発生したら */
      /* ドアが閉まっていて鍵が開いていれば次に進む */
         key_status = KEY_LOCKED;goto LOCK;
      ::(x==3 && door_status==DOOR_CLOSE &&
         key_status == KEY_UNLOCKED) ->
      /* 「パスワード変更しろ」イベントが発生したら */
      /* ドアが閉まっていて鍵が開いていれば次に進む */
         password[0] = window[0];
         password[1] = window[1];
         password[2] = window[2];
         password[3] = window[3];
         key_status = KEY_LOCKED;
         clear(); /* パスワードを変更して施錠する */
         goto LOCK;
      fi;
   od;
};

active proctype user(){ /* ユーザのプロセス */
   chanbutton!NUM(1);
   chanbutton!NUM(2);
```

```
    chanbutton!NUM(3);
    chanbutton!NUM(4);
    /* ユーザはランダムに以下の操作を行う */
    do
    ::opendoor();
    ::closedoor();
    ::chanbutton!NUM(0);
    ::chanbutton!NUM(1);
    ::chanbutton!NUM(2);
    ::chanbutton!NUM(3);
    ::chanbutton!NUM(4);
    ::chanbutton!NUM(5);
    ::chanbutton!NUM(6);
    ::chanbutton!NUM(7);
    ::chanbutton!NUM(8);
    ::chanbutton!NUM(9);
    ::chanbutton!C(0);
    ::chanbutton!L(0);
    od;
};

never {    /* !<>(open&&locked) */
/* 鍵が掛かってるときにドアは開かない */
T0_init:
    if
    :: ((locked) && (open)) -> goto T0_init
    :: (1) -> goto accept_all
    fi;
accept_all:
    skip;
};
```

Promela 及び Spin についてのノート

伊藤　昌夫 (nil)

2005 年 1 月 5 日

そして，本来的時間性にあっては，<将来>と<既在>と<瞬間的現在>が
緊密に結びつき，しかも<将来>が圧倒的な優位に立つ．
「ハイデガー拾い読み」　木田元

1　概要

反応系（基本的にシステムは停止せず，イベント駆動するシステム）を考える上で，イベントの順序列や並行プロセスを記述することは必須である．PROMELA(PROcess MEta LAnguage) は CSP(Communicating Sequential Processes) に基づいている．最初に時間論理と CSP について概観し，次に Promela と実行系である SPIN（Simple Promela INterpreter）について見る[*1]　SPIN は，線形時間論理 (LTL:Linear Temporal Logic) を扱うことができる．なお，この短い文でそれぞれの手法についての全体を知ることはできない．あくまで，ホンの糸口を与えるだけである．詳細は，参考文献を当たること．

2　時間論理と CSP

2.1　時間論理

時間論理（temporal logic）は，様相論理 (modal logic) の特殊な型であり [7]，Kripke 構造[*2] の性質を記述するのに用いることができる．即ち，どのような実行順でふるまうべきか，或いはどういうふるまいが不正であるかを示す時に使用する．Kripke 構造は，$\mathcal{K} = \langle P_r, S, S_0, T, l \rangle$ で示される．ここで，P_r は，原子命題の組みであり，S は，状態の組み，S_0 は初期状態である．$T \subseteq S \times S$ は，全ての遷移を示し，$l : S \to \mathcal{P}(Pr)$ は与えられた状態 s において，原子命題が正当であることを示す状態ラベル化関数である．

時間論理として特殊なものに線形時間論理 (LTL) がある，線形時間論理とは，離散的で，任意の点は，必ず一つの可能な未来を持っている．また，それ以前の時がない開始時刻がある．今，$p \in P_r$ ならば，線形時間論理式 ϕ は次のように再帰的に定義することができる．

$$\phi ::= p \mid true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\phi \mid F\phi \mid G\phi \mid \phi U \phi.$$

X, F, G, U は時間演算子と呼ばれ，経路 π の性質を示す．

[*1] Promela も Spin も共に略語であるが，以後はそれぞれ Promela, Spin と記述する)．

[*2] 有限状態機械の一種であり，グラフで表現される．グラフ上でノードは，系の到達可能な状態を表し，エッジは状態遷移を表す．

- X：現時点の直後の状態において真である (○ とも書かれる).
- F：何れかの将来において真になる.（◇ とも記述される).
- G：現時点以降，全てにおいて真である（□）とも記述される).
- U：$\phi_1 U \phi_2$ としたとき，経路上のある点で ϕ_2 が真になるならば，それ以前の状態 ϕ_1 も真であるならば，真である[*3].

$K, \pi \models p$ は，"構造 K において，論理式 p は時系列 π において正しい"ということを表す．これを用いて，次のような定義をすることができる.

- $K, \pi \models true$
- $K, \pi \models p$ *iff* $p \in l(x_0)$, ここで，$p \in P_r$ であり，x_0 は π の最初の状態である.
- $K, \pi \models \neg\phi$ *iff not* $K, \pi \models \phi$
- $K, \pi \models \phi_1 \wedge \phi_2$ *iff* $K, \pi \models \phi_1 and K, \pi \models \phi_2$
- $K, \pi \models \phi_1 \vee \phi_2$ *iff* $K, \pi \models \phi_1 or K, \pi \models \phi_2$
- $K, \pi \models X\phi$ *iff* $K, \pi^1 \models \phi$
- $K, \pi \models F\phi$ *iff* 例えば，$K, \pi^1 \models \phi$ を満足する $i \geq 0$ が存在する
- $K, \pi \models G\phi$ *iff* 全ての $i \geq 0$ に対して，$K, \pi^i \models \phi$ が成り立つ
- $K, \pi \models \phi_1 U \phi_2$ *iff* $i \geq 0$ を満足する i に対して，$K, \pi^i \models \phi_2$ であり，
  かつ，$j \in 0, ..., i-1$ である全ての j に対して，$K, \pi^j \models \phi_1$ が成り立つ

システムには幾つかの性質 (*property*) がある．システムの性質を知ることができるというのが，時間論理に従って記述することの利点である．性質には幾つかの区分がある．例えば，悪いことは決して起こらないという安全性（*safety*）及び，何時かは良いことが起こるという活性（*liveness*）である．例えば，障害物があるときにロボットアームを伸ばしてはいけないという不変条件が維持されることは仕様上の安全性であり，デッドロックを起こさないことというのは設計上の安全性に関わる性質である．また，意図しない終了をしない，あるイベントが生起したときには，対応するアクションを実行しなくてはいけないというのは仕様上或いは設計上の活性に関わるシステムの性質である．ここでは，[1] に従って時間論理の構造化から分類した例を挙げる.

- 安全性（*safety*）: $\Box p$
- 保証性（*guarantee*）: $\Diamond p$
- 拘束性（*obligation*）: $\Box p \vee \Diamond q$
- 応答性（*response*）: $\Box\Diamond p$ or $\Box(p \rightarrow \Diamond q)$
- 永続性（*persistence*）: $\Diamond\Box p$
- 反応性（*reactive*）: $\Box\Diamond p \vee \Diamond\Box q$

一般に，LTL を用いて，実務者が仕様を記述するには訓練を必要とする．そのためにデザインパターンに類似したパターンも提案されている [8]．証明すべきシステムの性質とは，逆にいえば，（それが適切に分類されていれば）容易に仕様として記述できたところで，後は証明器に委譲することができることになる．その観点からの分類が以下のものである.

[*3] 上記の *until* 演算子は，強い *until* と呼ばれるときもある．弱い *until*（通常 W と示される）では，$\phi_1 U(\phi_2 \mid \Box\phi_1)$ と等価になる．即ち，ϕ_1 が真であれば，ϕ_2 の値には関係しない.

生起 (occurence)	不在 (absence)
	普遍（universality）
	存在（existence）
	時間境界付存在（bounded existence）
順序（order）	先行（precedence）
	反応（response）
複合（compound）	先行連鎖（precedence chains）
	反応連鎖（response chains）
	ブーリアン（boolean）

例えば，「プロセス A が値を更新してから（updateA）でないと，プロセス B は読ん（readB）ではいけない」，という仕様は，反応パターンを使って次のように書ける．

$$\Box(updateA \wedge \Diamond readB)$$

2.2 CSP

CSP で扱うプロセスは，データ構造とサービスをカプセル化したコンポーネントである．但し，概念的なオブジェクト指向とは異なり，データ構造ばかりか，サービスも隠蔽されている．他に対して開かれているのはチャネルを介して行う通信のみである．

CSP*4は，広い意味でプロセス代数の一つと呼ばれる．基本プロセスを表す定数記号及びプロセスを合成するための演算記号の集合を定め，それらからなる代数項としてプロセスを定義する [3][4]．CSP では，観測可能なイベントを用いて，複数のプロセスからなるシステムを記述する．プロセス間のやりとりはアクションとして記述する．なお，プロセスのふるまいに影響を与えるイベントの集合をアルファベットと呼ぶ．この時，プロセスが起動したイベントと，他から受け付けるイベントは区別しない．

次のプロセス PHONE において，

$$PHONE = ring \rightarrow answer \rightarrow STOP.$$

プロセス PHONE は次のような有限個の軌跡を持つ．

$$\{\langle\rangle, \langle ring\rangle, \langle ring, answer\rangle\}$$

一方で次のように再帰表現されるとき，その軌跡は無限である．

$$CLOCK = tick \rightarrow CLOCK$$

今，アルファベット $\{a, b, c\}$ を持つプロセス P を考える．また同様に，アルファベット $\{b, c, d\}$ を持つプロセス Q を考えると次図のように表現することができる ([3] pp.54-55)．

(I)

図 1 独立した 2 つのプロセス

これら 2 つが並行して協調動作する場合，同じイベント同士をつなぐことによって，次のような接続線図を書くことができる．ここで，図 1 の b はメッセージの通信路と考えることもできる．従って，通信は次のように対として考えることができる．今，チャネルの名前を c とし，メッセージを v とし：

$$c.v$$

P がチャネル c を通して通信できる全てのメッセージの集合，通信から要素チャネル及びメッセージを取り出す関数はそれぞれ次のようになる．

$$\alpha c(P) = \{v \mid c.v \in \mathrm{ran}\, \alpha P\},\ channel(c.v) = c,\ message(c.v) = v$$

(II)

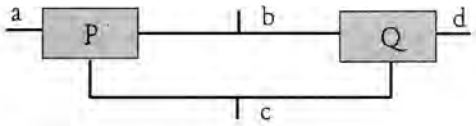


図 2　並行して動作するプロセス

次に，$\alpha R = \{c, e\}$ となるプロセス R が追加された場合，次図となる．

(III)

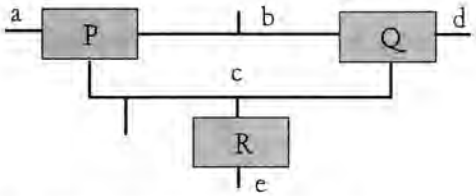


図 3　並行して動作する 3 つのプロセス

しかし，これを詳細を隠蔽した次図のように単一プロセスとして記述することもできる．

(IV)

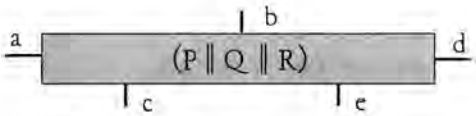


図 4　単一プロセスとしての表現

*4 Communicating Sequential Processes の略

3 Promela と Spin

3.1 Promela

下記に例を示す．文法とについては，[5] を参照のこと．

```
1  mtype = { msgtype }                          /*メッセージ型の宣言*/
2  chan name = [0] of {mtype, byte};            /*ランデブーポートの宣言*/
3  active proctype A()                          /*プロセスの宣言A*/
4  {     name!msgtype(124);
5        name!msgtype(121);
6  }
7  active proctype B()                          /*プロセスの宣言B*/
8  {     byte state;
9        name?msgtype(state);
10 }
11 init                                         /*プロセスの生成*/
12 { atomic {run A(); run B() }
13 }
```

この例では，プロセス A は最初のメッセージをバッファサイズがゼロであるランデブーポートに送信したところで（state が 124 を持つ），実行不能になる．バッファサイズが 2 以上の場合，プロセス A はプロセス B の終了を待たずに停止することができる．バッファサイズが 1 の時は，最初のメッセージを送信後，B がポートから値を取得したところで，A は再度送信する．そこで，両方とも停止状態となる．

3.2 Spin

Spin は，並行システムの論理的な一貫性（特に，データ通信プロトコル）を分析するためのツールである [2].

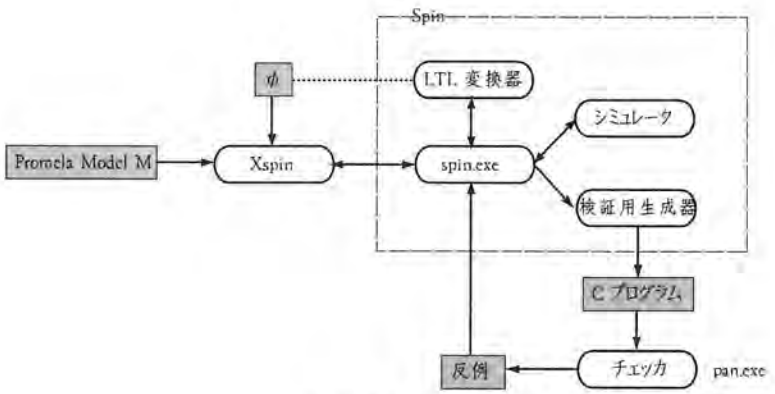


図 5 Spin 全体図

先に述べた時間論理に関して，Promela は直接的には関与しない．図 1 に示すように，Spin は LTL を理解

するパーザを持つ ([2], pp.141)[*5].

```
1  $ spin -f '<>[] p'
2  never {    /* <>[] p */
3  T0_init:
4          if
5          :: ((p)) -> goto accept_S4
6          :: (1) -> goto T0_init
7          fi;
8  accept_S4:
9          if
10         :: ((p)) -> goto accept_S4
11         fi;
12 }
```

ここでは，永続性を展開している．p が真であるならば，自動的に生成されたラベル accept_S4 に飛び無限ループする．そうでないならば，評価を繰り返す Promela の式を生成している．

[*5] until 演算子については，強いもののみをサポートする

4 まとめ

CSP における概念的な枠組みは，自己充足したコンポーネントである [6]．また，時間論理は，(反応系の) システムの性質について，妥当性確認する機会を与える．また，検証することも可能であり，より信頼性の高いプログラムを作る上で有益である．また，有限状態検証の一つである Spin ツールは，いわゆる「状態空間爆発 (state space explosion) 問題」に対して，有効に機能する．実世界において，WEB や組込み系といった反応系の高い信頼性を保持するシステムを，多数のコンポーネントから構成しようとするとき，これらアプローチは工学としての基盤をなすと考えられる．

参考文献

[1] Zohar Manna, Amir Pnueli, THE TEMPORAL LOGIC OF REACTIVE AND CONCURRENT SYSTEMS, Springer-Verlag, 1991.

[2] http://spinroot.com/spin/Man/Manual.html

[3] C.A.R. Hoare, COMMUNICATING SEQUENTIAL PROCESS, http://www.usingcsp.com/, 2004 (邦訳は，吉田訳「ホーア CSP モデルの理論」， 丸善， 1992).

[4] 堀田 英一, プロセス代数の意味論, 情報処理学会誌, Vol.37, No.4, 1996.

[5] Gerard J. Holzmann, THE SPIN MODEL CHECKER, Addison Wesley, 2003. 古いが Promela 及び Spin を扱っている Holzmann 氏の別の著書に DESIGN AND VALIDATION OF COMPUTER PROTOCOLS [*6](邦訳：コンピュータプロトコルの設計法）がある．

[6] Steve Schneider, CONCURRENT AND REAL-TIME SYSTEMS - THE CSP APPROACH -, Wiley, 2000.

[7] Allen Emerson, "Temoral and Modal Logic", (in Handbook of Theoretical Computer science), Elseveir Science, 1990.

[8] Matthew B. Dwyer, et al, "Property Specification Patterns for Finite-State Verification", 実際のパターン集については，http://patterns.projects.cis.ksu.edu/にある．

[*6], 原書の電子版は http://spinroot.com/spin/Doc/popd.html

Promela 簡介

伊藤　昌夫 (nil@nil.co.jp)

2005 年 2 月 23 日

1　概要

これは，Promera についてのごく簡単な解説である．基本的には，[1] のページの構文に関する部分の拙い訳になっている．記述が古い部分に関しては，4.2.4 の仕様に合わせた．なお，参照個所が本資に含まれていない場合がある．

2　*Promela*

Proomela プログラムは プロセス，メッセージ チャンネル，および 変数から構成される． プロセスは大域的なオブジェクトである． メッセージチャンネルと変数は大域的ないしは局所的に宣言できる．プロセスには，振る舞い，チャネル，大域変数を記述し，プロセスを実行する環境を規定する．

2.1　語彙に関する仕様

5 つの字句クラスがある: 識別子，キーワード，定数，演算子，および文分離子． 空白，タブ，ニューライン，改行，およびコメントは字句を分離するために用いる． 1 つ以上の解釈が可能であるなら，字句を構成可能な最も長い文字列として解釈する．

2.2　コメント

コメントは，/*から始まって，*/で終わる．コメントは入れ子にしても良い．

2.3　識別子

識別子は，一文字／ピリオド／アンダースコアで始まり，ゼロ個以上の文字，数字，ピリオド，アンダースコアの何れかがにより連接されたものである．

2.4　キーワード

以下の識別子は，キーワードとして予約されている．
while はキーワードではないことに注意．

List 1 *Promela* のキーワード

active	assert	atomic	bit	
bool	break	byte	chan	
d_step	D_proctype	do	else	
empty	enabled	fi	full	
goto	hidden	if	init	
int	len	mtype	nempty	
never	nfull	od	of	
pc_value	printf	priority	proctype	
provided	run	short	show	skip
timeout	trace	typedef	unless	unsigned
xr	xs			

2.5 ラベル

ラベルは識別子であり，最後にコロン (:) を付加する． 全ての文はラベル化できる．*end*，*progress* ないしは *accept* で始まるラベルは特別な意味を持っている．

2.6 定数

定数は 10 進整数を表す数字列である． 浮動小数点はサポートしていない． 2 つの方法で定数のための英字名を定義できる． 最初の方法は C 言語風のマクロ定義である．

List 2 定数

```
#define NAME    5
```

もう一つは，キーワード *mtype* を利用する． シンボル定数参照．

2.7 式 (expression)

式を作成するために以下の演算子と関数を使用できる:

List 3 演算子と関数

+	−	*	/	%	>
>=	<	<=	==	!=	!
&	||	&&	|	~	>>
<<	^	++	−−		
len()	empty()	nempty()		nfull()	full()
run	eval()	enabled()		pc_value()	

ほとんどの演算子は 2 進数演算用である． 文脈によって，論理的な否定 (!) とマイナス (-) 演算子は単項と 2 進演算の両方であることができる． ++ と −− 演算子は，C 言語と同様に単項接尾演算子である．排他的論理和は ^ である．最下段の一つ前の行にある関数は，単項であり，メッセージチャンネルだけに適用される (メッセージチャネル参照)． *len*() は，チャンネルが保持するメッセージの数を返す． 他のチャンネル演算

子は，既定義の意味があり半順序簡約をより有効に機能するために導入されている．半順序簡約を参照のこと．これらの関数を他と組み合わせ，新たな式として用いるべきではない．例えば，$!empty(q)$ は構文誤りとなる．$nempty(q)$ を用いるべきである．

最終行の単項関数は特別なものである．最初の演算子は，プロセスのインスタンス化のために用いる（プロセスの項参照）．実行後，幾つかのインスタンスを生成する．2 番目（*eval*）については，チャネル演算子の項で議論する．また，他の 2 つについては，*never* 要求 の項で説明する．

最初の 4 行は，C 言語と同様に定義されている．即ち，ブーリアンの *false* は，0 の値であり，その他は非ゼロとなり，truth を表す．

2.7.1 条件式

List 4 条件式

```
( expr1 -> expr2 : expr3 )
```

*expr*1 が 0 であれば上の式は *expr*3 の値をとり，そうでなければ，*expr*2 の値をとる．−＞は必要であり，';' で置き換えることはできない（文（statement）の項参照）

2.8 宣言 (declaration)

プロセス，チャンネル，変数などは使用する前に宣言しなければならない．変数とチャネルはプロセス内部で局所的に宣言することもできるし，大域的に宣言することもできる．プロセスは，*proctype* 宣言中でのみ大域的に宣言することができる．局所的な宣言はプロセスの本体中のどこにでも記述することができる．

2.8.1 変数

変数宣言は，基本データ型である *bit*, *bool*, *byte*, *short*, *int* に引き続いて一つ以上の識別子が記述される．初期子を続いて記述することもできる．

List 5 変数宣言

```
byte name1, name2 = 4, name3
```

初期子が記述されている場合，定数でなくてはならない．デフォルトでは，変数は 0 に初期化される．これらの基本データ型のビット幅は，それぞれ 1, 1, 8, 16, 32 である．最後の 2 つは符号付であるが，前の 3 つは，符号なしである．ユーザ定義型の変数も同様である（構造を参照のこと）．

2.8.2 配列

変数配列は例えば次のように宣言する．

List 6 配列の宣言

```
int name[4]
```

配列は，初期子として単一の定数を持つ．それにより全ての配列要素を初期化する．配列は，C 言語と同様に 0 から始まる．従って，上記の場合の最大のインデックスは，3 である．固定長のみ宣言できる．

2.8.3 シンボリック定数

シンボリック定数は，以下のように宣言できる．

List 7 シンボリック定数の宣言

```
mtype = {OK, READY, ACK}
```

一般に，メッセージタイプの定義に用いられる．そして，この型の変数は，次のように宣言できる．

List 8 シンボリック定数の宣言

```
mtype Status = OK;
```

mtype の定義は一つだけが許される．それは大域的であり，最大 255 個のシンボリック定数を持つことができる．

#define マクロに対する *mtype* の優位性は，次の点である．*SPIN* は，シンボリック定数の型を理解し，シミュレーションの間，値ではなく，シンボル名を使用する．

2.8.4 メッセージチャンネル

メッセージチャンネルは例えば次のように宣言される．

List 9 メッセージチャネルの宣言

```
chan Transfer   =[2] of {mtype, bit, short, chan};
chan Device[3]  =[0] of {byte};
chan Channel;
```

ここで，Transfer はチャネルであり，最大 2 つのメッセージをチャネルに保存できる．メッセージの型は，中括弧内に示される．Device は，チャネルの配列であり，各チャネルは同期用になる．即ち，メッセージをチャネルは保管できないので，送受信は同期しなくてはならない．最後の Channel は初期化されていないチャネルであり，別の初期化済みチャネルを割り当てるという使い方のみである．

チャネルはメッセージの一部にもなり得るということに注意せよ．チャネル上での排他的受信 (*xr*) や排他的送信 (*xs*) は以下のように記述する．

List 10 排他的なチャネルへのアクセス

```
xr Transfer;
xs Channel;
```

動作している様々なプロセス中で，*xr* を宣言しているプロセスのみが，Transfer からメッセージを受け取ることができ，同様に *xs* を宣言しているプロセスのみが，Channel にメッセージを送信することができる．

もし，分析中に他のプロセスが Transfer からメッセージを受け取ったり，Channel にメッセージを送信した場合は，実行中エラーである．詳細については，半順序簡約の項参照．

2.8.5 構造

typedef 定義によって，ユーザ定義のデータ型を用いることができる．

ユーザ定義型は，変数宣言中でもちいることもできるし，型の定義が必要な場所で用いることができる：

List 11 ユーザ定義のデータ型

```
typedef Msg {
        byte a[3], b;
        chan p
}
```

List 12 ユーザ定義型の利用

```
Msg foo;
chan stream =[0] of {mtype,Msg}
```

構造要素は，C 言語と似た方法でアクセスできる．

List 13 構造要素へのアクセス

```
foo.a[1]
```

2.8.6 隠蔽変数

List 14 隠蔽変数

```
hidden int foo
```

隠蔽変数は，システム状態の一部ではない（実行をみよ），そして，値を割り当て可能であるが，その値は常に未定義でである．有効な変数の無効化が必要な時にこれは有益である．例えば，チャネルバッファの中の値を消し去り（flush）たい時である．分析中に状態ベクトルのサイズを増加させないので，分析におけるメモリ要求を下げる効果がある（メモリと時間要求を見よ）．

大域的な宣言のみが有効である．大域的であっても，構造要素は局所的とみなされる．*bit*, *bool*, *channel* 変数を隠蔽変数にすることはできない．

'_' は事前定義された隠蔽変数であり，それは全ての文脈中で割り当て可能である．その（暗黙的な）型は *int* であり，*bit*, *bool*, *byte*, *mtype*, *short*, *int* の値を割り当て可能である．

2.9 プロセス

次の形式で基本的なプロセスの宣言をすることができる．

List 15 プロセスの宣言

```
proctype pname( chan In, Out; byte id )
{ statements }
```

プロセスは，*run* 操作によって，インスタンス化できる．

List 16 プロセスのインスタンス化

```
run pname(Transfer, Device[0], 0)
```

最初に実引数を仮引数に割り当て，次にプロセスのボディ部を実行する．各プロセスインスタンスは，ユニークで，整数のインスタンス番号を持っている．これは *run* 演算子（それと pid ）によって生成されたも

のである（特殊な変数を見よ）．インスタンス化されたプロセスは，プロセスのボディ部が終了するまで，活性状態にある．

プロセスは配列を引数の型としてとることはできない．しかし，構造（型）は許される．

プロセス宣言の最も一般的なフォームは以下のものである．

List 17 一般的なプロセスの宣言

```
active [N] proctype pname(...) provided (E) priority M
```

active 修飾子は，システムの初期状態において，N 個の *proctype* のインスタンスを活性状態にする．もし，[N] が記述されない場合，一つのインスタンスだけが活性状態になる．*active* 修飾子によって活性化されたインスタンスの引数は 0 に初期化される．即ち，実引数は，run-文によってのみ渡される．

実行可能条件を式 E に記述することができる．この式に記述できるのは，一般的な副作用なしの表現であり，定数／大域変数／事前定義された変数であるタイムアウト／ pid を含むことができ，局所変数／引数／遠隔参照は記述できない．実行可能条件は初期状態において一度だけ評価される．

ランダム模擬の間（意味があるのは，この模擬のときのみである），プロセスインスタンスは優先度 M で実行される．優先度は，1 以上の定数である．このプロセスを優先度 1 のものと比べ，M 倍多くスケジューリングされることを意味する．実行優先度は，run-文においても同様に用いることができる．

List 18 優先度付きのプロセスの実行

```
run pname(...) priority M
```

run 文によってインスタンス化されたプロセスは，明示的にないしは暗示的に（デフォルトは 1 ）示された優先度を常に持つ．

分析中では，優先度は意味がないことに注意せよ．

2.9.1 決定的プロセス

List 19 決定的プロセスの宣言

```
D_proctype pname( chan In , Out; byte id )
{ statements }
```

上記の pname の全てのインスタンスは決定的である．それ以外の他の影響はない．もしインスタンスが実行中に決定的でないならば，検証時に誤りとなる．

決定性といのは，動的な性質であることに注意せよ．もし，pname が次の文をそのボディ部に持つとする：

List 20 チャネルへの書き出しと読み出し

```
if
  :: In?v -> ...
  :: Out!e -> ...
fi
```

このときにチャネルへの書き込みが同時である必要があるインスタンスが存在する場合，非決定的となる．

2.9.2 初期プロセス

List 21 初期プロセス

```
init { statements }
```

このプロセスは，一度だけインスタンス化される．変数や適切なプロセスをインスタンス化することによって，システムの初期状態を生成するためにしばしば用いる．

2.9.3 never 要求

List 22 never 要求

```
never { statements }
```

これは，一度だけインスタンス化される特別なタイプのプロセスである．これについては，時間要求で述べる．*never* 要求は，望ましくない或いは不正と考えられる振る舞いを検出するために用いる．

個々の文は，条件として解釈される（実行可能性，式を参照）．従って，副作用を持つべきではない（副作用については，構文誤りというより警告を出すことになる）．

副作用を持つ文には，割り当て，自動増加（auto-increment）／自動減少（auto-decrement）演算，通信，実行，表明がある．

never 要求とともに関数ないしは変数を用いることができる．

- *enabled(pid)*: pid によって示されるインスタンス番号を持っているプロセスが操作を実行可能であれば，このブーリアン関数は，true を返す．この関数は，同期型通信を持たないシステムでのみ用いることができる．
- *pc_value(pid)*: この関数は，pid によって示されるインスタンス番号を持ったプロセスが実行中の状態の数を返す（-d 実行時オプションによって状態数が与えられる）．
- *_np*: システム状態において，少なくとも一つの実行プロセスが，進行（progress）ラベルによって印付けられている制御フロー上にある時，false を返す．*never* 要求と *trace* 表明で用いられる．

2.9.4 trace 表明

List 23 trace 表明

```
trace { statements }
```

never 要求と同様に，*trace* 表明は新たな振る舞いを規定するわけではない．*trace* 表明は，メッセージチャネルの性質を表す．特に，プロセスがメッセージチャネルに働きかける操作列が妥当か否かについての形式的な文である．

2.9.5 特別な変数

List 24 特別な変数

```
_, np_, _pid, _last
```

- _ は書き込みのみの変数で，メッセージフィールドのうち気にする必要のない場所を表すのに用いる．
- *_pid* は，事前定義された局所変数である．プロセスインスタンスのインスタンス番号を保持している．

- *_last* は事前定義された大域変数である．現時点の実行列のなかで最後のステップを実行するプロセスインスタンスのインスタンス番号を保持している．初期値は 0 である．*never* 要求と *trace* 表明で用いられる．

2.9.6 遠隔参照

List 25 遠隔参照

```
procname[pid]@label
```

この式は，インスタンス番号 pid を持つプロセス procname が，label 文の直前にいるときに，*true* を返す．

2.10 文 (statement)

文として次を使用することができる．

List 26 文を構成するもの

assert	assignment	**atomic**	**break**
declaration	**d_step**	**else**	expression
goto	receive	selection	**skip**
repetition	send	**timeout**	**unless**

式（expression）は文と同様に使用できるということにも注意せよ．

文はセミコロン (;) 或いは，矢印 (− >) によって分離する．これらは同じ意味であるが，矢印は，後続の文との因果関係を示すのに時に用いられる．また，選択や繰返しにおいて，ガード条件とガードされる文を分離するのに用いられる．選択と繰返しを見よ．

構成要素として，文より小さなものはない場合，基本 (basic) と呼ばれる．例えば，代入は基本文であるが，選択はそうではない．

2.10.1 実行可能性

文の実行は，その実行可能性に依存している． 文は，実行可能 (enable) かブロックされている (blocked) かの何れかである．先の文のリストのうち，代入文 (assignment)，宣言文 (declaration)，表明文 (assert)，スキップ (skip)，goto 文，break 文は常に実行可能である．もし，ある文がブロックされたならば，実行可能になるまでその点で停止する．

2.10.2 表明 (assert)

List 27 表明文

```
assert( expression )
```

この式 expression が false(ないしは 0) を返す場合，プログラムを abort する．そうでない場合は，単に通過する．

2.10.3 核化 (atomic)

List 28 核化文

```
atomic { statements }
```

statements の個々の文においてそれぞれを実行しようと試みる．即ち，核化プロセスを実行中に分岐して他のプロセスを実行しないということである．核化文は，最初の文が実行可能であったときのみ実行可能になる．

局所的な計算を核的にすることは，検証システムの複雑性を減らすために重要である．核化文と *d_step* を見よ．

実行中，明示的な *goto* や，核化のスコープ内で文がブロックされた点でのみ，そのスコープの外に制御が移る．このとき，もし，続きの文が再び実行可能になるならば，その点から，再度実行する．

スコープ内で何が起きるかについての制約はない．但し，ネスト化された核化や *d_step* は許可されない．核化のスコープ内で，（ラベル化された）位置にジャンプすることは可能である．

run 文によってインスタンス化されたプロセスインスタンスのボディ部は，（活動中の）核化文のスコープの外側にあると考える．

2.10.4 break

繰返しを参照のこと．

2.10.5 goto

List 29 goto 文

```
goto label
```

goto と同じ手続き中にあるラベル label に制御を移す文である．

2.10.6 決定的ステップ

List 30 決定的ステップ

```
d_step { statements }
```

これは，核化と同様な効果を持つが，別の効果も持つ（核化文と *d_step* を見よ）．決定的ステップのスコープ中の文は完全に決定的である．しかし，*d_step* のスコープ外のラベルに制御は移動しない．また，外部からスコープ内へのジャンプも存在しない．遠隔参照 (remote reference) は許可されない．最初の文が実行可能な場合のみ，*d_step* は実行可能である．*d_step* は，*d_step* の直前はスコープ内とみなし，最終行の直後はスコープ外とみなす．従って，下記のコード (List. 31) で label の効果を出そうとする場合：

List. 32 にあるコードに変更しなくてはならない．

2.10.7 else

選択を参照のこと．

2.10.8 式 (expression)

自動増加／自動減少 (++ / −−) を使用しないどんな式も文として使用可能である．この場合，もし評価結果が非ゼロ値ならば，厳密な意味で実行可能である．実行可能な式は，状態に変更を与えることなく実行さ

List 31 効果のないラベル

```
        goto label;
        ...
label:
        d_step {
                ...
                do
                ...
                        break
                ...
                od
        }
```

List 32 効果のあるラベル

```
        goto label;
        ...
label: skip;
        d_step {
                ...
                do
                ...
                        break
                ...
                od; skip
        }
```

れる.

例えば, busy wait ループを書く代わりに, 次を使用することができる.

List 33 busy wait ループ

```
        while (a != b) skip
```

以下も同等である.

List 34 より簡単な busy wait ループの記述

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

2.10.9 チャネル操作

これらは (List. 35), 標準的なチャネルの操作である.

最初は, 送信文である. 二番目は受信文である. ここで, q はチャネルであり, var_1, $const$, var_2… は, チャネルのメッセージ型と合致しなくてはならない. 送信, 受信が可能となるためには, q は初期化されなければならない. 更に, チャネルがバッファを持つならば, 送信はバッファが一杯でないときのみ可能になる. 受信はバッファが空でないときのみ可能になる. バッファを持たないチャネルにおいて, 送受信は, 共に受送

List 35 チャネル操作

```
q!var1,const,var2,... 或いは  q!var1(const,var2,...)
q?var1,const,var2,... 或いは  q?var1(const,var2,...)
```

信が可能なとき（即ち同期している）時のみ可能である．受信文は，チャネルから最も古いメッセージを読む．送信文は，メッセージを（バッファを持っている限りにおいて）そこに置く．バッファを持たないチャネルへの操作は，同期を取ってのみ実行可能である．

受信リスト中の定数は，実行可能性に制約を与える．すなわち，最も古いメッセージ中の対応する値が同じである場合に可能であり，そうでない場合はブロックされる．

同様の制約をチャネル操作の実行可能性に加えるために，局所ないしは大域変数を用いることもできる．

List 36 チャネルからの値の取得

```
q?var1,eval(var2),var3
```

2 番目のメッセージが var_2 の値に等しくない場合，この受信はブロックされる．var_2 の値は変化しないことに注意せよ．

バッファリングされたチャンネルのための以下の操作がある．

- $q?[var_1, const, var_2]$: 受信操作が可能である時に，true を返す（即ち非ゼロ値）．– 問い合わせ
- $q? < var_1, const, var_2 >$:メッセージがバッファから取り除かれないということを除けば，通常の受信操作と同様である．
- $q!!var_1, const, var$:ソート化送信：数字順，辞書順で直ちにメッセージを送信する．
- $q??var_1, const, var$: ランダム受信：同一のメッセージが存在したら，最も古いメッセージを受信する．
- $q??[var_1, const, var_2]$: 先の問い合わせ操作と同様．
- $q?? < var_1, const, var_2 >$: メッセージがバッファから取り除かれないということを取り除けば，標準的なランダム受信と同様である．

バッファ付きチャネルは，Spin のコマンドラインスイッチ "-m" によって異なる振る舞いをする．スイッチが指定された場合，バッファが一杯であっても，送信アクションはブロックされない．その代わり，バッファが一杯の状態で送信されたメッセージは失われる．

2.10.10 選択

List 37 選択文

```
if
:: statements
...
:: statements
fi
```

オプション (::から始まる) の中の 1 つを選択して，実行する．もし，最初の文（ガード）が可能ならば，そのオプションが選択される．少なくとも，選択可能な分岐が一つもないとき，選択はブロックされる．一つ以上のオプションが選択可能ならば，ランダムにそのうちの一つが選択される．特別なカードである *else* が選

択と繰返し文で一度だけ用いられる．もし，他のガードがブロックされている時のみ，実行可能となる．送信や受信文がガードとして用いられているときは利用されない．

2.10.11 繰返し (repetition)

List 38 繰返し文

```
do
:: statements
...
:: statements
od
```

選択と同様であるが，*goto* や *break* によって，制御が明示的に外側に移らない限り，繰り返して実行されるという点が異なる．*break* は，実行中の最も内側の繰返し文を終了させる．繰返し以外では *break* は用いられない．

2.10.12 skip

影響を与えない．構文上の要求を満足するために主として用いられる．

2.10.13 timeout

timeout 文は，システム中の全ての他の文がブロックされているときにのみ実行可能になる．実行されても，影響を与えない．

2.10.14 unless

List 39 unless 文

```
{ statements-1 } unless { statements-2 }
```

statements-1 の文を実行し，各文が実行終了する前に，statements-2 の文の実行可能性が検査される．もし，実行可能であれば，statements-1 は *abort* し，制御は，statements-2 に移る．そうでなければ，statements-1 に制御が残る．もし，statements-1 が終了すれば，statements-2 は無視される．

unless において，*atomic* とは対照的に *d_step* は分割不能と考えられる．即ち，もし *d_step* のスコープ中に statements-1 の制御があるならば，statements-2 の実行可能性は，検査されない．

参考文献

[1] Gerth, R., Concise Promela Reference, http://spinroot.com/spin/Man/Quick.html, 1997

今後の SEA Forum の予定

ここ何年か SEA Forum 開催の間隔がまちまちになっていましたが，先ごろの幹事会で話し合った結果，この春以降は少なくとも毎月１回は開催しようということに決まりました．主な会場としては，東京・東銀座のＪＪＫ会館（全国情報サービス産業厚生年金会館）を考えています．また，それぞれの地方支部の企画にもとづいて，大阪・名古屋・福岡その他の都市でも順次開催します．

とりあえず現在予定されている東京での Forum の一覧を以下にまとめておきます．

期日	テーマ
４月３日（月）	ウォータフォール再考（再考？）— すでに案内済み
５月１７日（水）	アスペクト指向は実用的か？
５月３１日（水）	システム危機と仕様検証（東証のトラブルは防げたか？）
６月１６日（水）	オブジェクト指向の理想と現実 － 当日夜はＳＥＡ総会
７月上旬	ＧＰＬ Version 3
８月２１日（月）	要求工学（Ss2006 での討論を受けて）

これらの Forum の記録（簡単なまとめと発表に用いられたスライドその他の資料）は，順次 SEAMAIL に収録して会員に配布するつもりです（これで SEAMAIL も船便状態から脱出できるか？：－）

秋以降の東京 Forum および地方開催 Forum の企画は，これから幹事会で検討します．会員各位からのご希望や提案を sea@sea.or.jp までお寄せください．

SS2006 in 熊本での新しい試み

すでにみなさんのお手元に届いた Call for Papers SEA Forum でアナウンスされているように，今年のソフトウェア・シンポジウム（7/19-21）では，３日間の会期のうちの第２日目をいくつかのテーマによるワークショップ形式の討論にあてるという新しい試みを行います．これまでのように講演や論文発表を聴くだけのシンポジウムではなく，参加者全員が討論に参加するという形のイベントにしようという狙いです．

４月１７日に予定されている応募論文の審査が終わり，プログラムが決まったあと，参加者募集が始まりますが，参加申し込みにさいしては，希望するワークショップへのエントリーを，簡単なポジション・ステートメントつきで指定することが求められることになると思います．

発表論文は応募しなかったがシンポジウムには参加したいと考えておられる方々は，どのテーマのワークショップに参加されるか，あらかじめ考えておいてください．

Call for Papers にリストアップされていたワークショップのテーマは次の通りでした：

A. プロセス，メトリクス，マネジメント
B. 要求，テスティング，メインテナンス
C. 仕様記述技法
D. 教育
E. FLOSS，セキュリティ，ネットワーク
F. モデリング
G. 組込み

この他に８番目のテーマとして「要求工学」も考えられており，このワークショップの討論結果は８月２１日の Forum で公開される予定です．

それぞれのワークショップの具体的な討論の企画は参加者募集のさいに明らかになります．期待してお待ちください．

編集後記

☆

昨年の２月に開催されたデザインワークショップの報告をお届けします．

☆☆

このレポートは，ワークショップの世話人だった野中さんから，去年の夏にはいただいてあって，SS2005 特集の Vol.14, No.7 のあとすぐに発行する予定だったのですが，IWFST2005 その他の雑用に追われているうちに，いつのまにか冬も終わる季節になってしまいました．

☆☆☆

巻末 (94ページ) の「今後の SEA Forum の予定」に書いたように，これからは毎月定期的に Forum を開催し，その記録を SEAMAIL に載せて行くということが先日の幹事会で決まったので，これからはいままでのような船便状態からは脱出できるのではないかと考えています(甘いか？)．

☆☆☆☆

ソフトウェア技術者協会
〒160-0004 東京都新宿区四谷3-12 丸正ビル5F
Tel:03-3356-1077　Fax:03-3356-1072
E-mail:sea@sea.or.jp
URL:http://www.sea.jp/