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 11 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竦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、初心者には予期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。

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: report.tex,v 1.1 2006/01/18 04:55:34 vdmtool
 *
 * 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] 荒木啓二郎; 張漢明; 荻野隆彦; 佐原伸; 染谷誠 訳ジョン・フィッツジェラルド; ピーター・ゴーム・ラーセン 著. ソフトウェア開発のモデル化技法. 岩波書店, 2003.

[8] 山崎利治. Dw2005 のための問題, Feb. 2005.

図 1　交信図