怠け者のためのソフトウェアモデル検査入門

安部 達也

まえがき

本書は、システムの安全性や活性について検査せざるを得ない状況に陥ってしまったときに最小限のことを学ぶことで乗り切ろうとする、とにかく必要に迫られない限りは新しい概念を導入せずに、最短経路で目的を達成しようとする人たちに向けたソフトウェアモデル検査の入門書である。

目次

第 1 章　逐次プログラム 3
1.1　状態遷移系 . . . 3
1.2　モデル検査器 SPIN . . . 4
1.3　命題論理式 . . . 7
1.4　状態遷移系におけるアサーション . . . 8

第 2 章　並行プログラム 9
2.1　非決定性 . . . 9
2.2　パスの検査 . . . 10
2.3　線形時相論理式 . . . 11

第 3 章　典型的な検査 14
3.1　停止にかかわる検査 . . . 14
3.2　進行性検査 . . . 16
3.3　公平性 . . . 18

第 4 章　Büchi オートマトン 20
4.1　定義と諸性質 . . . 20
4.2　非空問題の決定可能性 . . . 22
4.3　線形時相論理式の変換 . . . 22

第 5 章　効率的な探索 25
5.1　抽象解釈 . . . 25
5.2　半順序簡約 . . . 28

第 6 章　組合せ検査 30
6.1　仕様 . . . 30
6.2　非干渉性 . . . 31
6.3　契約 . . . 32

参考文献 35

第 1 章

逐次プログラム

逐次プログラムを渡されて、「このシステムでほげほげという性質は常に成り立っているか？」と問われたという状況であると仮定する。

1.1 状態遷移系

渡されたプログラムは

```
1 int x;
2
3 int main() {
4   x = x + 1;
5
6   x = x + 1;
7
8   x = x + 1;
9
10  x = x + 1;
11
12  return 0;
13 }
```

といったものであった。システムと大袈裟に言っているが、この程度のプログラムで書かれるものであったとする。しかし、まあシステムと言っているので、このプログラムが表している状態遷移系もここで書いておく。

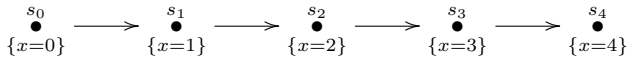
状態遷移系（state transition system）とは $\mathfrak{S} = (S, s_0, \to, V)$ という四つ組であり、

1. S は集合である（$s \in S$ を状態と呼ぶ）
2. $s_0 \in S$ である（初期状態と呼ぶ）
3. $\to \subseteq S \times S$、つまり、$\to$ は S 上の関係である（遷移関係と呼ぶ）
4. $V \colon S \to \mathfrak{P}(AP)$、つまり、$V$ は 状態集合 S から原子命題集合 AP の部分集合への関数である（付値と呼ぶ）

を満たすものをいう。

おっと、原子命題集合 AP なるものを定義していなかった。それは成り立っているかを確かめたい性質が何であるかに依るものであるが、ここでは $x > 0$ や $x = 0$ といったものを原子命題と呼ぶことにし、それらを集めたものを原子命題集合と呼ぶことにし、これは既に与えられているものと仮定する。

前述のプログラムが表している状態遷移系は



と図示される。上記の状態遷移系の定義に従って正確に記述すると、

$$
\begin{aligned}
S &= \{s_0, s_1, s_2, s_3, s_4\} \\
\to &= \{\langle s_0, s_1\rangle, \langle s_1, s_2\rangle, \langle s_2, s_3\rangle, \langle s_3, s_4\rangle\} \\
V &= \{s_0 \mapsto \{x = 0\}, s_1 \mapsto \{x = 1\}, s_2 \mapsto \{x = 2\}, s_3 \mapsto \{x = 3\}, s_4 \mapsto \{x = 4\}\}
\end{aligned}
$$

である。

1.2 モデル検査器 SPIN

このシステムにおいて x の値が常に 0 以上であるかを確かめたいとする。となれば、やはりここは

```
1 int x;
2
3 int main() {
4   printf("x: %d\n", x);
5
6   x = x + 1;
7   printf("x: %d\n", x);
8
9   x = x + 1;
10  printf("x: %d\n", x);
11
12  x = x + 1;
13  printf("x: %d\n", x);
14
15  x = x + 1;
16  printf("x: %d\n", x);
17
18  return 0;
19 }
```

プリントデバッグである。先頭行と `x = x + 1` がある行ごとに x の値を表示させればよい。0、1、2、3、4 が表示されるはずである。確かに x の値は常に 0 以上である。しかし、x の値が常に 0 以上であるかを目視

するのも面倒な話である。普段代わりに計算してくれている便利な機械がせっかく目の前にあるのだから、この確認も代行してもらいたいものである。

道具は何でもよいのであるが、ここではモデル検査器 SPIN [2] と呼ばれるものとそのモデリング言語 PROMELA[*1] を使うことにする。PROMELA は C に似た構文を持つ。実は、意味はあまり似ていないのだが今のところは気にしないことにする。前述の C プログラムを PROMELA で書くと

```
1 int x;
2
3 active proctype proc() {
4   printf("x: %d\n", x);
5
6   x = x + 1;
7   printf("x: %d\n", x);
8
9   x = x + 1;
10  printf("x: %d\n", x);
11
12  x = x + 1;
13  printf("x: %d\n", x);
14
15  x = x + 1;
16  printf("x: %d\n", x);
17 }
```

といったものになる。3 行目が異なるだけである。このまま `printf` を使い続けては PROMELA で書いた意義が無いので

```
1 int x;
2
3 active proctype proc() {
4   assert(x >= 0);
5
6   x = x + 1;
7   assert(x >= 0);
8
9   x = x + 1;
10  assert(x >= 0);
11
12  x = x + 1;
13  assert(x >= 0);
```

[*1] https://spinroot.com/spin/Man/promela.html

```
14
15   x = x + 1;
16   assert(x >= 0);
17 }
```

というように printf の代わりに assert を使う。ここで、

```
1 $ spin -a foo.pml; gcc pan.c; ./a.out
```

とすると

```
1 (Spin Version 6.5.2 -- 15 February 2024)
2         + Partial Order Reduction
3
4 Full statespace search for:
5         never claim             - (none specified)
6         assertion violations    +
7         acceptance   cycles     - (not selected)
8         invalid end states      +
9
10 State-vector 20 byte, depth reached 10, errors: 0
11        11 states, stored
12         0 states, matched
13        11 transitions (= stored+matched)
14         0 atomic steps
```

という出力を得る。assertion violations が + であり assert(x >= 0) が検査されていて、そのうえで errors: 0 であることを確認する。

念のため、assert(x > 0) に変更した上で改めて検査してみると

```
1 pan: wrote foo.pml.trail
2
3 (Spin Version 6.5.2 -- 15 February 2024)
4 Warning: Search not completed
5         + Partial Order Reduction
6
7 Full statespace search for:
8         never claim             - (none specified)
9         assertion violations    +
10        acceptance   cycles     - (not selected)
11        invalid end states      +
12
13 State-vector 20 byte, depth reached 0, errors: 1
```

```
14          1 states, stored
15          0 states, matched
16          1 transitions (= stored+matched)
17          0 atomic steps
```

errors: 1 と出力されており、確かになにかおかしいと言っていることが確認できる。どうせ 4 行目でアサーション違反をしているのであろうが pan: wrote foo.pml.trail と言ってくれているので、実際に違反した実行列（反例と呼ぶ）を確認しておこう。反例は

```
1 $ spin -t -p foo.pml
```

で確認できて

```
1 spin: foo.pml:4, Error: assertion violated
2 spin: text of failed assertion: assert((x>0))
3   1:    proc  0 (proc0:1) foo.pml:4 (state 1)   [assert((x>0))]
4 spin: trail ends after 1 steps
5 #processes: 1
6                 x = 0
7   1:    proc  0 (proc0:1) foo.pml:6 (state 2)
8 1 process created
```

なるほど、確かに 4 行目で assert((x>0)) に違反していると報告されている。

1.3 命題論理式

今のところ x >= 0 といった単一の原子命題の検査しかおこなっていないが、もっと複雑なことの検査もできる。命題論理式の集合を

- $\top$ は論理式
- 原子命題 p は論理式
- φ が論理式のとき $\neg\varphi$ は論理式である
- φ と ψ が論理式のとき $\varphi \wedge \psi$ は論理式である

を満たす最小の集合と定義する。世の中には Backus–Naur 記法（BNF）と呼ばれる便利な記法があり、これを使うと、

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi$$

と短く書けて大変便利である。選言 $\varphi \vee \psi$ は $\neg(\neg\varphi \wedge \neg\psi)$ で表され、また、本稿では含意を $\supset$ で書くことにし、$\varphi \supset \psi$ は $\neg(\varphi \wedge \neg\psi)$ であるものとする。

状態遷移系 $\mathfrak{S} = (S, s_0, \to, V)$ 中の状態 s が命題論理式 φ を満たすことを

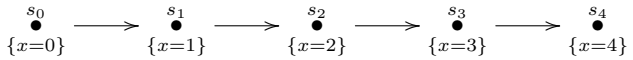
$$
\begin{aligned}
\mathfrak{S}, s &\vDash \top \\
\mathfrak{S}, s &\vDash p \Longleftrightarrow p \in V(s) \\
\mathfrak{S}, s &\vDash \neg\psi \Longleftrightarrow \mathfrak{S}, s \nvDash \psi \\
\mathfrak{S}, s &\vDash \varphi_0 \wedge \varphi_1 \Longleftrightarrow \mathfrak{S}, s \vDash \varphi_0 \text{ かつ } \mathfrak{S}, s \vDash \varphi_1
\end{aligned}
$$

と定義し $\mathfrak{S}, s \vDash \varphi$ と書く。初期状態 s_0 と遷移関係 $\rightarrow$ が $\mathfrak{S}, s \vDash \varphi$ であるかどうかにまったくかかわっていないが、この伏線は 2.3 節で回収する。

1.4 状態遷移系におけるアサーション

1.2 節のとおり `assert(x >= 0)` はプログラム中に書かれるものであった。これは 1.1 節で導入した状態遷移系においてはどのように現れてくるであろうか。

本章で扱っているプログラムが表す状態遷移系は前述のとおり



である。`assert(x >= 0)` はこの各状態において成り立っている原子命題から $x \geq 0$ がいえるかを検査せよと指示しているといえる。実は、このプログラム中の `assert(x >= 0)` と状態遷移系中の状態におけるアサーション検査が一対一に対応するのは逐次プログラムだけに見られるものであり、並行プログラムでは成立しない。

第 2 章

並行プログラム

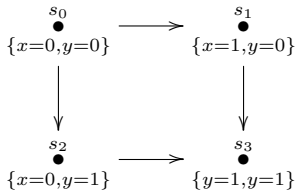
本章では並行性のあるシステムの検査をおこなうために並行プログラムを扱う。

2.1 非決定性

1.1 節で定義した状態遷移系における遷移関係 $\rightarrow$ は S 上の関係であって関数であるとは限らず、$\langle s_0, s_1\rangle, \langle s_0, s_2\rangle \in \rightarrow$（ただし $s_1 \neq s_2$）であり得る。こういった非決定性を持つ状態遷移系を生むプログラムの例として

```
1 int x, y;
2
3 active proctype proc0()
4 {
5   x = x + 1;
6   assert(x==y);
7 }
8
9 active proctype proc1()
10 {
11   y = y + 1;
12   assert(x==y);
13 }
```

を考える。プロセス `proc0` と `proc1` はお互いに待合せをしない、つまり、非同期に実行される。これの状態遷移系を図示すると



というように分岐の存在を確認できる。よって、逐次プログラムを検査するときと異なり、並行プログラムに

おいてはステイトメントの実行列（パスと呼ぶ）は複数存在し得て、この場合、二つのパスが存在する。

プログラム中に `assert(x==y)` は二つだけ存在するが、この状態遷移系においてこの `assert(x==y)` が確認されるかもしれない状態は s_1、s_2、s_3 というように三つある。状態遷移系のある状態において φ が成り立っているかを検査せよという指示が `assert(`φ`)` であるが、その状態をプログラム中で指定することは必ずしも容易でない。1.2 節でプリントデバッグに言及したが、逐次プログラムに有効なプリントデバッグが並行プログラムに対しては必ずしも有効でない原因の一つがこれである。

2.2 パスの検査

これまでは「この（特定の）状態において φ という性質が成り立っているか」ということだけを考えてきたが、「このシステムにおいてずっと φ が成り立っているか」ということを考えたいとする。2.1 節の言葉でいうと、「任意のパスにおいて φ がずっと成り立っているか」である。これにはシステムを監視するプロセスを別に立ち上げるとよい。以下のプログラムは

```
1 #define N 5
2 int x;
3
4 active proctype increment()
5 {
6   do
7   :: atomic { x < N -> x++ }
8   od
9 }
10
11 active proctype decrement()
12 {
13   do
14   :: atomic { x > 0 -> x-- }
15   od
16 }
17
18 active proctype monitor() {
19   do
20   :: assert(0 <= x && x <= N);
21   od
22 }
```

x が N 未満なら x をインクリメントするプロセス `increment` と x が 0 より大きければ x をデクリメントするプロセス `decrement` に、ずっと $0 \leq x \leq N$ であるかを監視し続けるプロセス `monitor` を追加したものである。PROMELA の構文で初出のものがあるがどういうものかは予想がつくと思うので説明を省略する。

このように任意のタイミングで `monitor` が `assert(0 <= x && x <= N)` を実行できるため、SPIN に網

羅的探索してもらうことで「ずっと $0 \leq x \leq N$ が成り立っているか」を検査できる。これで話が済めば楽なのであるがちょっと込み入った性質を調べたくなると困ったことになる。例えば、「$x < 0$ である間はずっと $y > 0$ であるか」といったような性質を調べたいときには、当然 `if` や `do` といった制御構文を用いた監視プロセスをつくることはできるものの、調べたい性質から監視プロセスへの翻訳という手間がかかる。これの原因は、調べたい性質が宣言的に与えられているにもかかわらず、監視プロセスは手続き的に書かれなければならないからである。この手間を失くすために、モデル検査器 SPIN に調べたい性質を直接与えることにする。

2.3 線形時相論理式

1.3 節で定義した命題論理式を

$$\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathrm{X}\,\varphi \mid \varphi\,\mathrm{U}\,\varphi$$

と拡張したものを線形時相論理（Linear Temporal Logic, LTL）式という。任意の i で $\langle s_i, s_{i+1}\rangle \in \to$ である状態の無限列 $\pi = s_0 s_1 \cdots$ をパスと呼ぶ。有限列は最後の状態を繰り返し続けること（stuttering という）で無限列と見なせる。3.1 章で停止性に関する検査をする場合は stuttering を都合よく無視したりする。π 中の i 番目の状態を $\pi(i)$ と書く。π の i 番目以降を $\pi \restriction i$ と書く。$(\pi \restriction i)(0)$ は s_i である。状態遷移系 $\mathfrak{S} = (S, \to, V)$ 中のパス π が線形時間論理式 φ を満たすことを

$$\begin{aligned}
\mathfrak{S}, \pi &\vDash \top \\
\mathfrak{S}, \pi &\vDash p \iff p \in V(\pi(0)) \\
\mathfrak{S}, \pi &\vDash \neg\psi \iff \mathfrak{S}, \pi \nvDash \psi \\
\mathfrak{S}, \pi &\vDash \varphi_0 \wedge \varphi_1 \iff \mathfrak{S}, \pi \vDash \varphi_0 \text{ かつ } \mathfrak{S}, \pi \vDash \varphi_1 \\
\mathfrak{S}, \pi &\vDash \mathrm{X}\,\psi \iff \mathfrak{S}, \pi \restriction 1 \vDash \psi \\
\mathfrak{S}, \pi &\vDash \varphi_0\,\mathrm{U}\,\varphi_1 \iff \text{ある } i \text{ が存在して } \mathfrak{S}, \pi \restriction i \vDash \varphi_1 \text{ かつ任意の } j < i \text{ で } \mathfrak{S}, \pi \restriction j \vDash \varphi_0
\end{aligned}$$

で定義する。$\Diamond\psi$ と $\Box\psi$ を $\top\,\mathrm{U}\,\psi$ と $\neg\Diamond\neg\psi$ にそれぞれ定義することで

$$\begin{aligned}
\mathfrak{S}, \pi &\vDash \Diamond\psi \iff \text{ある } i \text{ が存在して } \mathfrak{S}, \pi \restriction i \vDash \psi \\
\mathfrak{S}, \pi &\vDash \Box\psi \iff \text{任意の } i \text{ で } \mathfrak{S}, \pi \restriction i \vDash \psi
\end{aligned}$$

である。1.3 節では特定の状態が論理式を満たすかどうかといったことしか考えることができず、以下のように、パスを満たすかどうかといったことを考えることができなかった。

命題 2.1. φ を命題論理式とする。このとき、任意の π, π' に対して $\pi(0) = \pi'(0)$ ならば $\mathfrak{S}, \pi \vDash \varphi$ と $\mathfrak{S}, \pi' \vDash \varphi$ は同値である。

証明. $\mathfrak{S}, \pi \vDash \varphi$ として一般性を失わない。φ の構成に関する帰納法で示す。φ の形で場合分けをする。

φ が $\top$ のとき、ただちに従う。

φ が p のとき、$\pi(0) = \pi'(0)$ よりただちに従う。

φ が $\neg\psi$ のとき、$\mathfrak{S}, \pi \vDash \neg\psi$ より $\mathfrak{S}, \pi \nvDash \neg\psi$ である。帰納法の仮定より $\mathfrak{S}, \pi' \nvDash \neg\psi$ である。よって、$\mathfrak{S}, \pi \vDash \neg\psi$ である。

φ が $\varphi_0 \wedge \varphi_1$ のとき、$\mathfrak{S}, \pi \vDash \varphi_0 \wedge \varphi_1$ より $\mathfrak{S}, \pi \vDash \varphi_0$ かつ $\mathfrak{S}, \pi \vDash \varphi_1$ である。帰納法の仮定より $\mathfrak{S}, \pi' \vDash \varphi_0$

かつ $\mathfrak{S}, \pi' \vDash \varphi_1$ である。よって、$\mathfrak{S}, \pi' \vDash \varphi_0 \wedge \varphi_1$ である。 □

その一方、線形時相論理式に対してはパスがそれを満たすかを考えることができるようになっている。監視するプロセスの代わりに

```
#define N 5
int x;

active proctype increment()
{
  do
  :: atomic { x < N -> x++ }
  od
}

active proctype decrement()
{
  do
  :: atomic { x > 0 -> x-- }
  od
}

ltl { [](0 <= x && x <= N) }
```

というように SPIN に線形時相論理式を与えることでも検査できる。`ltl{`φ`}` の検査をするにあたっては 4.2 節で紹介する acceptance cycle checking を足さないといけないため、網羅的探索のコマンドに

```
$ spin -a foo.pml; gcc pan.c; ./a.out -a
```

というように `-a` というオプションをつける必要がある。

線形時相論理式により安全性や活性と呼ばれる性質を記述できる。安全であるとは、危険な状況をあらかじめ決めておいた場合にその状況に至らない（到達可能でない）ことをいう。危険な状況を φ であるとすると、これに至らないとは $\neg\Diamond\varphi$ である。もちろんこれと同値である、ずっと安全であるを意味する $\Box\neg\varphi$ でもよい。反対に活性とは、こうなってほしいという状況をあらかじめ決めておいた場合にその状況にいずれ至ることをいう。こうなってほしいという状況を φ であるとすると、これにいずれ至るとは $\Diamond\varphi$ である。線形時相論理式 $\varphi\,\mathrm{U}\,\psi$ を用いることで、より込み入った性質も検査できるようになった。

ちなみに、線形時相論理式の表現力にもある種の限界がある。$V(\pi)$ をパス π 中の状態 s_i に V をそれぞれ適用して得られる原子命題集合の列とする（トレースと呼ぶ）。任意のパス π で $\mathfrak{S}, \pi \vDash \varphi$ であることを $\mathfrak{S}, s_0 \vDash \varphi$ と書くことにする。以下のように、線形時相論理式はトレースの集合が一致している状態遷移系を区別できない。

命題 2.2. φ を線形時相論理式とする。$\mathfrak{S}_0, \mathfrak{S}_1$ のトレース集合が一致しているとき $\mathfrak{S}_0 \vDash \varphi$ と $\mathfrak{S}_1 \vDash \varphi$ は同値である。

証明. $\mathfrak{S}_0$ と $\mathfrak{S}_1$ の付値をそれぞれ V_0 と V_1 と書くことにする。

$\mathfrak{S}_0 \vDash \varphi$ として一般性を失わない。π_1 を $\mathfrak{S}_1$ 上にとる。$\mathfrak{S}_1 \vDash \varphi$ の真偽は $V_1(\pi_1)$ で決まる。

仮定より、$V_1(\pi_1)$ と一致する π_0 が $\mathfrak{S}_0$ 上に存在する。$\mathfrak{S}_0 \vDash \varphi$ より $\mathfrak{S}_0, \pi_0 \vDash \varphi$ である。$V_0(\pi_0)$ と $V_1(\pi_1)$ は一致しているので $\mathfrak{S}_1, \pi_1 \vDash \varphi$ である。π_1 は任意にとったので $\mathfrak{S}_1 \vDash \varphi$ である。 □

これらを区別したければそういった論理も存在するし、また、それと線形時相論理の両方を包含する、より表現力の強い論理も存在する。

第 3 章

典型的な検査

検査したい性質をユーザが与えるものだけでなく、知りたい性質としてよく挙げられるものの検査について紹介する。

3.1 停止にかかわる検査

SPIN はプロセスが遷移ができなくなるとそれを異常と判断して報告する。止まってしまってもよい（SPIN に見逃してほしい）箇所には end ラベルをつける。以下は最大公約数を求めるプログラムであり

```
1 repeat:
2   assert(0 < x && 0 < y);
3   printf("x␣=␣%d,␣y␣=␣%d\n", x, y);
4 end: if
5       :: (x > y) -> x = x - y
6       :: (x < y) -> y = y - x
7   fi;
8   goto repeat
```

4 行目で $x = y$ であって条件を満たすことができないことを利用してプログラムを停止させている。

これまでに停止するプログラムの例を挙げてきたが、SPIN がエラーを出力していないのは SPIN がプログラム中に end ラベルが無いときはプロセスの最終行に end ラベルがあるものとすると仮定しているからである。

以下はいわゆるデッドロックするプログラム

```
1 mtype = {LOCKED, UNLOCKED};
2 mtype mutex0 = UNLOCKED;
3 mtype mutex1 = UNLOCKED;
4
5 inline lock(m){
6   atomic { m == UNLOCKED -> m = LOCKED }
7 }
8
```

```
inline unlock(m){
  m = UNLOCKED
}

active proctype proc0()
{
repeat:
  lock(mutex0);
  lock(mutex1);

  unlock(mutex0);
  unlock(mutex1);
  goto repeat
}

active proctype proc1()
{
repeat:
  lock(mutex1);
  lock(mutex0);

  unlock(mutex1);
  unlock(mutex0);
  goto repeat
}
```

である。実際検査すると `pan:1: invalid end state (at depth 1)` という出力が得られる。その一方、

```
active proctype proc1()
{
repeat:
  lock(mutex0);
  lock(mutex1);

  unlock(mutex0);
  unlock(mutex1);
  goto repeat
}
```

というように `proc1` のロックをとる順番を変更すると invalid な end 状態に至って止まってしまった、ということは起きなくなる。

3.2 進行性検査

起こってほしいことが本当に起こる（進行性）の検査を紹介する。起こってほしいことの行に progress ラベルをふる。このラベルをふった行を通らないパスが存在するとき SPIN は異常と判断する。

```
1 mtype = {LOCKED, UNLOCKED};
2 mtype mutex0 = UNLOCKED;
3 mtype mutex1 = UNLOCKED;
4
5 inline lock(m){
6   atomic { m == UNLOCKED -> m = LOCKED }
7 }
8
9 inline unlock(m){
10  m = UNLOCKED
11 }
12
13 active proctype proc0()
14 {
15 repeat:
16   lock(mutex0);
17   lock(mutex1);
18
19 progress:
20   skip;
21
22   unlock(mutex0);
23   unlock(mutex1);
24   goto repeat
25 }
26
27 active proctype proc1()
28 {
29 repeat:
30   lock(mutex0);
31   lock(mutex1);
32
33   unlock(mutex1);
34   unlock(mutex0);
```

```
35   goto repeat
36 }
```

といったプログラムを

```
1 $ spin -a foo.pml; gcc -DNP pan.c; ./a.out -l
```

で検査する。進行性の検査には（non-progress を意味する NP をオンにする）-DNP と（liveness を意味する）-l オプションが必要である。

```
1 pan:1: non-progress cycle (at depth 2)
```

という出力を得る。これは non-progress、つまり progress ラベルを通らない、cycle、つまりある状態からその状態へのサイクルが存在することを意味し、すなわち、いくら待っても progress ラベルを通ることがないパスが存在するということである。

具体的には proc1 が延々とロックをとり続けていて proc0 に譲っていない。そこで、手番（turn）を考慮する。つまり、

```
1 bit turn0 = 0;
2 bit turn1 = 0;
3
4 inline lock(m,t){
5   atomic { m == UNLOCKED && t == _pid -> m = LOCKED }
6 }
7
8 inline unlock(m,t){
9   m = UNLOCKED; t = 1 - _pid
10 }
11
12 active proctype proc0()
13 {
14 repeat:
15   lock(mutex0, turn0);
16   lock(mutex1, turn1);
17
18 progress:
19   skip;
20
21   unlock(mutex0, turn0);
22   unlock(mutex1, turn1);
23   goto repeat
24 }
25
```

```
26 active proctype proc1()
27 {
28 repeat:
29   lock(mutex0, turn0);
30   lock(mutex1, turn1);
31
32   unlock(mutex0, turn0);
33   unlock(mutex1, turn1);
34   goto repeat
35 }
```

とすれば、一度ロックをとったら次は譲るので進行性が保証される。

3.3 公平性

前節で手番を導入することで譲るを実現したが、そういったある種の公平性を与えたうえで検査するということもできる。例として

```
1 active proctype proc0()
2 {
3   do
4   :: 1;
5 chosen:
6       printf("0-0\n");
7 progress:
8       printf("0-1\n");
9   od;
10 }
11
12 active proctype proc1()
13 {
14   do
15   :: printf("1-0\n");
16      printf("1-1\n");
17   od;
18 }
```

を考える。PROMELA の実行モデルがインターリービングであり、proc1 が隙間なく実行し続けていると proc0 が実行する間隙を与えてもらえず progress ラベルがふってある行が実行されない。

SPIN に -f というオプションがあり、これをつけると、ずっと実行可能であればいずれ実行される、という仮定（弱公平性という）を加えられる。

```
$ spin -a foo.pml; gcc -DNP pan.c; ./a.out -l -f
```

この仮定の下で進行性が保証される。

弱公平性の仮定だけは SPIN のオプションで置いたうえで検査できるが、それ以外の公平性については線形時相論理式を与えて検査をおこなう。以下のプログラム

```
active proctype proc0()
{
  do
  :: lock(mutex);
chosen:
     printf("0-0\n");
here:
     printf("0-1\n");
     unlock(mutex);
  od;
}

active proctype proc1()
{
  do
  :: lock(mutex);
     printf("1\n");
     unlock(mutex);
  od;
}
```

を考える。このとき □◇proc0@here、つまり、here が無限回実行される、は成り立たない。というのも、proc1 が実行し続けるパスが存在するからである。

無条件公平性 □◇proc0@chosen、つまり、chosen が無限回実行される、という仮定を置く。このとき □◇proc0@here は成り立つ。というのも、chosen が実行されるその直後に here も実行されるからである。

弱公平性 ¬◇□(mutex == UNLOCKED ∧ ¬proc0@chosen)、つまり、chosen がずっと実行可能であるならば chosen がいずれは実行される、という仮定を置く。このとき □◇proc0@here は成り立たない。というのも、ずっと mutex == UNLOCKED はたびたび実行不可能になるからである。

その一方、強公平性 ¬(□◇mutex == UNLOCKED ∧ ◇□¬proc0@chosen)、つまり、chosen が無限回実行可能であるならば chosen がいずれは実行される、という仮定を置く。このとき □◇proc0@here が成り立つ。というのも、というのも、その無限回 chosen が実行されるその直後に here も実行されるからである。

第 4 章

Büchi オートマトン

状態遷移系において線形時相論理式を満たすかどうかがどのように検査されているかの理論を紹介する。

4.1 定義と諸性質

有限オートマトン $\mathfrak{A} = (\Sigma, S, I, \delta, F)$ とは、

1. Σ は有限集合（$a \in \Sigma$ をアルファベットと呼ぶ）
2. S は有限集合（$s \in S$ を状態と呼ぶ）
3. $I \subseteq S$ は初期状態集合
4. δ は $S \times \Sigma$ から $\mathfrak{P}(S)$ への関数
5. $F \subseteq S$ は終了状態集合

をいう。アルファベットの有限列全体の集合を Σ^* と書く。$w = a_0 a_1 \cdots a_{n-1} \in \Sigma^*$ が $\mathfrak{A}$ に受理されるとはある $s_0, s_1, \ldots, s_n$ が存在して

- $s_0 \in I$
- 任意の $1 \leq i < n$ について $s_i \in \delta(s_{i-1}, a_{i-1})$
- $s_{n-1} \in F$

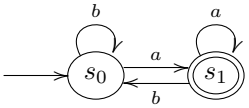
をいう。$\mathfrak{A}$ が受理されるもの全体を $L(\mathfrak{A})$ と書く。

Büchi オートマトンとは有限オートマトンをアルファベットの無限列を受理できるように拡張したものをいう。アルファベットの可算無限列全体の集合を Σ^ω と書く。$w = a_0 a_1 \cdots \in \Sigma^\omega$ が $\mathfrak{A}$ に受理されるとはある $s_0, s_1, \ldots$ が存在して

- $s_0 \in I$
- 任意の $1 \leq i$ について $s_i \in \delta(s_{i-1}, a_{i-1})$
- $\{\, s \mid \#(\{\, i \mid s_i = s \,\}) = \omega \,\} \cap F \neq \varnothing$

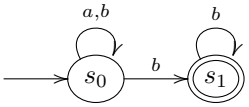
をいう。ただし、最後の条件中の $\#$ は集合の濃度をとる関数であり、ω は最小の極大順序数である。この三つ目の条件は終了状態が無限回現れるということを表している。

例 4.1. $\mathfrak{A} = (\{a,b\}, \{s_0,s_1\}, \{s_0\}, \{\langle s_0,a\rangle \mapsto \{s_1\}, \langle s_0,b\rangle \mapsto \{s_0\}, \langle s_1,a\rangle \mapsto \{s_1\}, \langle s_1,b\rangle \mapsto \{s_0\}\}, \{s_1\})$



とすると $L(\mathfrak{A})$ は a を無限回通るもの全体、つまり、$\{\, c_0c_1\cdots \mid \#(\{\, i \mid c_i = a\,\}) = \omega\,\}$ である。

例 4.2. $\mathfrak{A} = (\{a,b\}, \{s_0,s_1\}, \{s_0\}, \{\langle s_0,a\rangle \mapsto \{s_0\}, \langle s_0,b\rangle \mapsto \{s_0,s_1\}, \langle s_1,b\rangle \mapsto \{s_1\}\}, \{s_1\})$



とすると $L(\mathfrak{A})$ は a を有限回通るもの全体、つまり、$\{\, c_0c_1\cdots \mid \#(\{\, i \mid c_i = a\,\}) < \omega\,\}$ である。

二つの Büchi オートマトンについてそれらが受理するものの合併を受理する Büchi オートマトンをつくれることは以下のように容易にわかる。

命題 4.3. Büchi オートマトン $\mathfrak{A}_i = (\Sigma, S_i, I_i, \delta_i, F_i)$ $i = 0,1$ に対して $L(\mathfrak{A}) = L(\mathfrak{A}_0) \cup L(\mathfrak{A}_1)$ となる Büchi オートマトン $\mathfrak{A}$ が存在する。

証明. $\mathfrak{A}$ を $(\Sigma, (S_0 \times \{0\}) \cup (S_1 \times \{1\}), (I_0 \times \{0\}) \cup (I_1 \times \{1\}), \delta, (F_0 \times \{0\}) \cup (F_1 \times \{1\}))$、ただし

$$\delta(s,a) = \begin{cases} \delta_0(s_0,a) \times \{0\} & \text{if } s = \langle s_0, 0\rangle \\ \delta_1(s_1,a) \times \{1\} & \text{if } s = \langle s_1, 1\rangle \end{cases}$$

にとれば十分である。 □

その一方、共通部分については一考を要する。まず、有限オートマトンについて以下が成り立つ。

命題 4.4. 有限オートマトン $\mathfrak{A}_i = (\Sigma, S_i, I_i, \delta_i, F_i)$ $i = 0,1$ に対して $L(\mathfrak{A}) = L(\mathfrak{A}_0) \cap L(\mathfrak{A}_1)$ となる有限オートマトン $\mathfrak{A}$ が存在する。

証明. $\mathfrak{A}$ を $(\Sigma, S_0 \times S_1, I_0 \times I_1, \delta_0 \times \delta_1, F_0 \times F_1)$ にとれば十分である。 □

受理するものの共通部分を受理する Büchi オートマトンをつくるにあたっても基本的なアイデアは同じである。しかし、Büchi オートマトンの受理条件は有限オートマトンの受理条件と異なり、安直に終了状態集合の直積をとるとそれによって得られる受理条件は強すぎるものとなる。というのも、無限回通る状態の集合が一致していないといけないという要請になってしまうからである。そこで、この部分に微調整をおこなって以下を得る。

命題 4.5. Büchi オートマトン $\mathfrak{A}_i = (\Sigma, S_i, I_i, \delta_i, F_i)$ $i = 0,1$ に対して $L(\mathfrak{A}) = L(\mathfrak{A}_0) \cap L(\mathfrak{A}_1)$ となる Büchi オートマトン $\mathfrak{A}$ が存在する。

証明. まず、状態集合について $S_0 \times S_1 \times \{0,1\}$ というように、単なる $S_0 \times S_1$ でなく $\{0,1\}$ と直積をとることで $S_0 \times S_1$ のコピーをつくる。これは遷移によって状態のモードを 0 というモードと 1 というモードとに切り替えることがあるということだと考えてほしい。

初期状態集合を $I_0 \times I_1 \times \{0\}$ とする。モード 0 から開始するというつもりである。

遷移関数 $\delta(\langle s_0, s_1, i\rangle, a)$ を $\delta_0(s_0, a) \times \delta_1(s_1, a) \times \{\mathrm{mode}(s_0, s_1, i)\}$ とする。ただし、

$$\mathrm{mode}(s_0, s_1, i) = \begin{cases} 0 & \text{if } (i = 0 \text{ かつ } s_0 \notin F_0) \text{ または } (i = 1 \text{ かつ } s_1 \in F_1) \\ 1 & \text{if } (i = 0 \text{ かつ } s_0 \in F_0) \text{ または } (i = 1 \text{ かつ } s_1 \notin F_1) \end{cases}$$

である。モードが 0 であるとして、モードを 1 に切り替えるのは状態が F_0 に入ったときであり、また、そうでなければモードを 0 に維持しておく、ということである。逆に、モードが 1 であった場合も同様である。

終了状態集合を $F_0 \times S_1 \times \{0\}$ とする。$\mathfrak{A}_0$ の受理条件を満たすようになっていることはすぐにわかるだろう。その一方、以下のとおり $\mathfrak{A}_1$ の受理条件も満たしている。δ の定義により F_0 に入った瞬間にモードは 1 に変わり、 F_1 に入らないとモード 0 に戻ることはなく、よって、$F_0 \times S_1 \times \{0\}$ を無限回通らなければならないという条件は F_1 も無限回通らなければならないということをも要請している。 □

4.2 非空問題の決定可能性

Büchi オートマトンは有限状態であるもののその上には無限遷移が存在するので、受理または非受理の判定が決定可能であるかはただちにはわからない。しかし、アルファベット集合の有限性より一つの状態からの分岐は有限であることに注目するとわりと容易に非空問題については決定可能であることがわかる。

定理 4.6. $L(\mathfrak{A}) \neq \varnothing$ かどうかは決定可能である。

証明. まず、受理条件は終了状態を無限回通ることなので一回は通る必要がある。初期状態から終了状態への同じ状態を二度以上通らないという制約の下での到達可能性は前述の有限分岐性より有限回で判定できる。

受理条件は終了状態を無限回通ることなので、次はその終了状態からまた自身に戻ってくることができるかを確認する（acceptance cycle checking)。これもさきほどと同様に有限回で判定できる。 □

4.3 線形時相論理式の変換

U だけを持つ線形時相論理式 φ に対して、この φ だけを受理する Büchi オートマトンを $\mathfrak{A}_\varphi$ を構築する。

まず、受理状態集合 F を $F_0, \ldots, F_{k-1}$ と複数に増やした上で受理条件を、$w = a_0 a_1 \cdots \in \Sigma^\omega$ が $\mathfrak{A}$ に対してある $s_0, s_1, \ldots$ が存在して

- $s_0 \in I$
- 任意の $1 \leq i$ について $s_i \in \delta(s_{i-1}, a_{i-1})$
- 任意の $0 \leq j < k$ に対して $\{\, s \mid \#(\{\, i \mid s_i = s \,\}) = \omega \,\} \cap F_j \neq \varnothing$

に拡張したものを一般化 Büchi オートマトンという。以下が成り立つ。

定理 4.7. 一般化 Büchi オートマトン $\mathfrak{A}$ に対して $L(\mathfrak{A}) = L(\mathfrak{A}')$ となる Büchi オートマトン $\mathfrak{A}'$ が存在する。

証明. 命題 4.5 でおこなった Büchi オートマトンの構築を終了状態集合が二つから任意有限個に増えたという設定の下でおこなえば十分である。 □

次に、$cl(\varphi)$ を以下で定義する。

- $\top \in cl(\varphi)$

- $\varphi \in cl(\varphi)$
- $\psi \in cl(\varphi)$ ならば $\neg\psi \in cl(\varphi)$
- $\neg\psi \in cl(\varphi)$ ならば $\psi \in cl(\varphi)$
- $\varphi_0 \wedge \varphi_1 \in cl(\varphi)$ ならば $\varphi_0 \in cl(\varphi)$ かつ $\varphi_1 \in cl(\varphi)$
- $\mathrm{X}\,\psi \in cl(\varphi)$ ならば $\psi \in cl(\varphi)$
- $\varphi_0 \,\mathrm{U}\, \varphi_1 \in cl(\varphi)$ ならば $\varphi_0 \in cl(\varphi)$ かつ $\varphi_1 \in cl(\varphi)$

$s \subseteq cl(\varphi)$ が $cl(\varphi)$-極大無矛盾であるとは

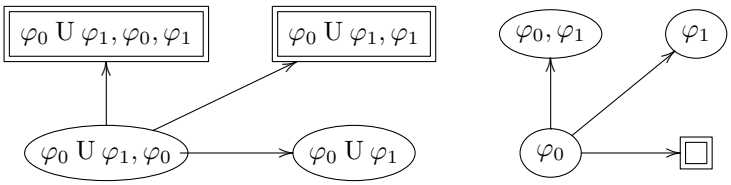
- $\psi \in cl(\varphi)$ について、$\psi \in s$ と $\neg\psi \notin s$ が同値
- $\varphi_0 \wedge \varphi_1 \in cl(\varphi)$ について、$\varphi_0 \wedge \varphi_1 \in s$ と $\varphi_0 \in s$ かつ $\varphi_1 \in s$ が同値

をいう。

最後に、$\mathfrak{A}_\varphi = (\mathfrak{P}(AP(\varphi)), \{\, s \mid s \text{ は } cl(\varphi)\text{-極大無矛盾} \,\}, \{\, s \mid \varphi \in s \,\}, \delta, \{\, F_{\varphi_0 \mathrm{U} \varphi_1} \mid \varphi_0 \,\mathrm{U}\, \varphi_1 \in cl(\varphi) \,\})$ とする。ただし、

- $AP(\varphi)$ は φ に現れる原子命題全体という有限集合である。
- $s' \in \delta(s, a)$ を以下のすべてを満たすものと定義する：
  - $a = s \cap AP$
  - $\mathrm{X}\,\psi \in cl(\varphi)$ について、$\mathrm{X}\,\psi \in s$ と $\psi \in s'$ が同値
  - $\varphi_0 \,\mathrm{U}\, \varphi_1 \in cl(\varphi)$ について
    - $\varphi_0 \,\mathrm{U}\, \varphi_1 \in s$
    - $\varphi_1 \in s$ または $[\varphi_0 \in s$ かつ $\varphi_0 \,\mathrm{U}\, \varphi_1 \in s']$

    が同値。以下の図中で二重に囲まれたものからはすべてのものに矢印が出ているものとする。



- $F_{\varphi_0 \mathrm{U} \varphi_1}$ を $\{\, s \mid \varphi_1 \in s$ または $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in s \,\}$ とする。

考え方を説明する。まず、状態の定義に $cl(\varphi)$-極大無矛盾性を利用することで X や U を持たない論理式たちの整合性は保証されている。その一方、一般の論理式たちとの整合性がとれているとは限らない。そこで、遷移をこのように定義することで X や U を持つ論理式との整合性もとろうとしている。しかし、U を持つ論理式については遷移の定義の工夫だけではまだ足りていない。そこで、終了状態集合族をうまく定義することで全体としてうまくいくようにしている、と考えるとよい。

$\varphi_0 \,\mathrm{U}\, \varphi_1$ について、これが成り立つものであるかはいずれ $\varphi_1 \in s$ となるかに依る。このことが $F_{\varphi_0 \mathrm{U} \varphi_1}$ の定義中の $\varphi_1 \in s$ に対応している。$\neg(\varphi_0 \,\mathrm{U}\, \varphi_1)$ については、まず、遷移の定義より以下が成り立つことに注意する。

命題 4.8. $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in cl(\varphi)$ について、

1. $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in s$

2. $\neg\varphi_1 \in s$ かつ $[\neg\varphi_0 \in s$ または $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in s']$

は同値である。

証明. $cl(\varphi)$-極大無矛盾性より $\varphi_0 \,\mathrm{U}\, \varphi_1 \notin s$ である。δ の定義より $\varphi_1 \notin s$ かつ $[\varphi_0 \notin s$ または $\varphi_0 \,\mathrm{U}\, \varphi_1 \notin s']$ である。$cl(\varphi)$-極大無矛盾性より $\neg\varphi_1 \in s$ かつ $[\neg\varphi_0 \in s$ または $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in s']$ である。逆も同様である。 □

よって、一般化 Büchi オートマトンの受理条件が終了状態集合族を無限回通らなければならないというものであることを考慮すると、$\neg(\varphi_0 \,\mathrm{U}\, \varphi_1)$ についての終了状態集合の条件は $\neg(\varphi_0 \,\mathrm{U}\, \varphi_1) \in s$ ということになる。

定理 4.9. $\#(S) < \omega$ であり、各 $s \in S$ に対して $\#(\{\, s' \mid \langle s, s'\rangle \in \to \,\}) < \omega$ である $\mathfrak{S} = (S, s_0, \to, V)$ について $\mathfrak{S} \vDash \varphi$ かどうかは決定可能である。

証明. $\mathfrak{A}_{\mathfrak{S}}$ を $(\mathfrak{P}(AP(\varphi)), S, \{s_0\}, \delta, S)$ と定義する。ただし、

$$\delta(s,a) = \begin{cases} \{\, s' \mid \langle s, s'\rangle \in \to \,\} & \text{if } V(s) = a \\ \varnothing & \text{otherwise} \end{cases}$$

である。

構成の定義より $\mathfrak{S} \vDash \varphi$ と $L(\mathfrak{A}_{\mathfrak{S}}) \subseteq L(\mathfrak{A}_{\varphi})$ は同値である。$L(\mathfrak{A}_{\varphi}) \cap L(\mathfrak{A}_{\neg\varphi}) = \varnothing$ なので、$L(\mathfrak{A}_{\mathfrak{S}}) \subseteq L(\mathfrak{A}_{\varphi})$ ならば $L(\mathfrak{A}_{\mathfrak{S}}) \cap L(\mathfrak{A}_{\neg\varphi}) = \varnothing$ である。逆に、$\mathfrak{S} \nvDash \varphi$、つまり、$\mathfrak{S} \vDash \neg\varphi$ とすると、前述の $\mathfrak{S} \vDash \varphi$ と $L(\mathfrak{A}_{\mathfrak{S}}) \subseteq L(\mathfrak{A}_{\varphi})$ の同値性より、$L(\mathfrak{A}_{\mathfrak{S}}) \subseteq L(\mathfrak{A}_{\neg\varphi})$ なので、当然 $L(\mathfrak{A}_{\mathfrak{S}}) \cap L(\mathfrak{A}_{\neg\varphi}) \neq \varnothing$ である。

よって、$\mathfrak{S} \vDash \varphi$ と $L(\mathfrak{A}_{\mathfrak{S}}) \cap L(\mathfrak{A}_{\neg\varphi}) = \varnothing$ は同値である。定理 4.7 より $\mathfrak{S} \vDash \varphi$ は決定可能である。 □

第 5 章

効率的な探索

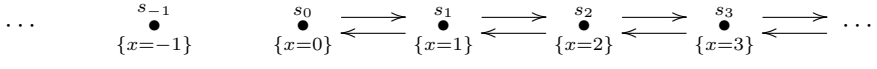
モデル検査をおこなっていくと探索する状態数の増加に苦しめられることが多い。本章では探索する状態数を削減する方法を紹介する。

5.1 抽象解釈

モデル検査器 SPIN は状態の有限性を前提とした検査をおこなうので、安直に考えると無限の状態を持ってしまうプログラムを検査することはできない。例えば、

```
1 int x;
2
3 active proctype increment()
4 {
5   do
6   :: atomic { x++ }
7   od
8 }
9
10 active proctype decrement()
11 {
12   do
13   :: atomic { x > 0 -> x-- }
14   od
15 }
16
17 ltl { [](0 <= x) }
```

といったプログラム、つまり、状態遷移系が



$$
\begin{aligned}
\mathfrak{S}_C &= (S_C, \to_C, V_C) \\
S_C &= \{\, s_i \mid i \in \mathbb{Z} \,\} \\
\to_C &= \{\, \langle s_i, s_{i+1} \rangle \mid 0 \leq i \,\} \cup \{\, \langle s_{i+1}, s_i \rangle \mid 0 \leq i \,\} \\
V_C &= \{\, s_i \mapsto \{x = i\} \mid i \in \mathbb{Z} \,\}
\end{aligned}
$$

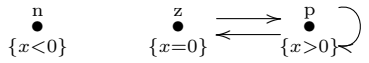
であるものを検査すると

```
1 error: max search depth too small
2
3 (Spin Version 6.5.2 -- 15 February 2024)
4         + Partial Order Reduction
5
6 Full statespace search for:
7         never claim             - (none specified)
8         assertion violations    +
9         acceptance   cycles     - (not selected)
10        invalid end states      +
11
12 State-vector 36 byte, depth reached 9999, errors: 0
13     10000 states, stored
14     20000 states, matched
15     30000 transitions (= stored+matched)
16         0 atomic steps
```

といった結果が出力される。`errors: 0` で一見うまくいっているように見えるが、1 行目の `error` を見落としてはならない。探索の深さ上限（デフォルトは 10000）に到達して探索が終了しておらず、その終了していない探索中でエラーが見つかっていないだけである。

このように SPIN は無限の状態を持つプログラムの検査をおこなうことができないが、1.1 章で紹介した状態遷移系の定義を読み返すと、そもそも状態集合には集合であることしか要求されておらず、どういった集合をとらなければならないかについては何も要求されていないことに気づく。また、付値の値域中の原子命題についても既に与えられているものと仮定されているだけであり、ここで $x = 0$ といったものをとらなければならない理屈はない。

そこで、厳密に x の値を見るのでなく x の正負・零非零だけを考慮する状態遷移系を考えてみる。つまり、



$$
\begin{aligned}
\mathfrak{S}_A &= (S_A, \to_A, V_A) \\
S_A &= \{\mathrm{n}, \mathrm{z}, \mathrm{p}\} \\
\to_A &= \{\langle \mathrm{z}, \mathrm{p} \rangle, \langle \mathrm{p}, \mathrm{z} \rangle, \langle \mathrm{p}, \mathrm{p} \rangle\} \\
V_A &= \{\mathrm{z} \mapsto \{x = 0\}, \mathrm{p} \mapsto \{x > 0\}, \mathrm{n} \mapsto \{x < 0\}\}
\end{aligned}
$$

を考えてみる。この状態線形系の状態数はたったの 3 つである。そのかわりに遷移はややこしくなっており、具体的には自己ループ、つまり、自分自身への遷移が存在する。また、$x > 0$ というように x の値が隠蔽されているため、実際には例えば $x = 2$ であれば遷移は z に遷移するはずはないのであるが z に遷移する可能性があるかもしれないというものになっている。このように $\mathfrak{S}_A$ は $\mathfrak{S}_C$ のある種の抽象化になっている。

$\mathfrak{S}_A$ は状態数がたった 3 つしかなく SPIN で検査できるので、$\mathfrak{S}_A$ を検査することで $\mathfrak{S}_C$ に対してある種の検査をしたことにできれば話がうまい。そこで、$\mathfrak{S}_C$ から $\mathfrak{S}_A$ へどういった抽象化がおこなわれているかを見ていく。$\mathrm{sgn}\colon \mathbb{Z} \to \{\mathrm{n}, \mathrm{z}, \mathrm{p}\}$ を符号判定をする関数とした上で

$$\mathfrak{P}(\mathbb{Z}) \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathfrak{P}(\{\mathrm{n}, \mathrm{z}, \mathrm{p}\})$$

$$\alpha(C) = \{\, \mathrm{sgn}(n) \mid n \in C \,\}$$
$$\gamma(A) = \{\, n \in \mathbb{Z} \mid \mathrm{sgn}(n) \in A \,\}$$

と定義すると、α と γ は単調、つまり、任意の $C, C' \subseteq \mathbb{Z}$ に対して $C \subseteq C'$ ならば $\alpha(C) \subseteq \alpha(C')$ であるし、任意の $A, A' \subseteq \{\mathrm{n}, \mathrm{z}, \mathrm{p}\}$ に対して $A \subseteq A'$ ならば $\alpha(A) \subseteq \alpha(A')$ である。また、$C \subseteq (\gamma \circ \alpha)(C)$ であり、$(\alpha \circ \gamma)(A) = A$ である。

この事実について、さらに一般化して、半順序集合 $(P, \leq_P), (Q, \leq_Q)$ に対して単調な $\alpha\colon (P, \leq_P) \to (Q, \leq_Q)$ と $\gamma\colon (Q, \leq_Q) \to (P, \leq_P)$ が存在し $p \leq_P (\gamma \circ \alpha)(p), (\alpha \circ \gamma)(q) \leq_Q q$ であるとする。このとき以下が成り立つ。

命題 5.1. $\alpha(p) \leq_Q q$ と $p \leq_P \gamma(q)$ は同値である。

状態遷移系の話に戻る。$\beta\colon S_C \to S_A$ を

$$\beta(s_i) = \begin{cases} \mathrm{n} & \text{if } i < 0 \\ \mathrm{z} & \text{if } i = 0 \\ \mathrm{p} & \text{if } i > 0 \end{cases}$$

で定義すると、以下のように、この β は遷移を保存する。

命題 5.2. $s_C \to_C s'_C$ ならば $\beta(s_C) \to_A \beta(s'_C)$ である。

また、β を $\beta(X) = \{\, \beta(x) \mid x \in X \,\}$ というように集合上に拡張すると以下が成り立つ。

命題 5.3. 任意の $X \subseteq S_C$ に対して $\beta(X) = \alpha(X)$ である。

集合 X を状態集合 S の部分集合とすると X から 1 ステップ以下で到達可能な状態 $f(X)$ は

$$f(X) = X \cup \{\, s' \in S \mid \text{ある } s \in X \text{ があって } s \to s' \,\}$$

である。f は単調である。X から i ステップ以下で到達可能な状態は $f^i(X) = (\overbrace{f \circ \cdots \circ f}^{i})(X)$ である。 X から到達可能な状態は $\bigcup\{\, f^i(X) \mid 1 \leq i \,\}$ である。

補題 5.4. 任意の $X \subseteq S_C$ に対して $\beta(f_C(X)) \subseteq f_A(\beta(X))$ である。

証明. 命題 5.2 より

$$\begin{aligned}\beta(f_C(X)) &= \beta(X) \cup \{\, \beta(s'_C) \in S_A \mid \text{ある } s_C \in X \text{ があって } s_C \to_C s'_C \,\} \\ &\subseteq \beta(X) \cup \{\, s'_A \in S_A \mid \text{ある } s_A \in \beta(X) \text{ があって } s_A \to_A s'_A \,\} \\ &= f_A(\beta(X))\end{aligned}$$

である。 □

定理 5.5. $\bigcup\{\, f_C^i(X) \mid 1 \le i \,\} \subseteq \gamma(\bigcup\{\, f_A^i(\beta(X)) \mid 1 \le i \,\})$ である。

証明. 補題 5.4 に命題 5.3 を通して命題 5.1 を適用することで $f(X) \subseteq \gamma(f_A(\beta(X)))$ を得る。ここで X に $f_C(X)$ を入れることで、f_A と γ の単調性より、任意の i について $f_C^i(X) \subseteq \gamma(f_A^i(\beta(X)))$ であることがわかる。γ は単調なので任意の $i \ge 1$ に対して $\gamma(\{\, f_A^i(\beta(X)) \mid 1 \le i \,\}) \subseteq \gamma(\bigcup\{\, f_A^i(\beta(X)) \mid 1 \le i \,\})$ である。ゆえに $\bigcup\{\, f_C^i(X) \mid 1 \le i \,\} \subseteq \gamma(\bigcup\{\, f_A^i(\beta(X)) \mid 1 \le i \,\})$ である。 □

定理 5.5 は、$\mathfrak{S}_C$ における到達可能性（安全性）を知りたいならば $\mathfrak{S}_A$ における到達可能性を調べてその結果を γ に通せばよい、ということを示している。

5.2 半順序簡約

対称性を利用した探索する遷移の削減方法を紹介する。そのために、今までよりも状態遷移系の遷移をもっと細かく見られるようにしておきたい。

本節では、状態遷移系とは $\mathfrak{S} = (S, s_0, T, V)$ という四つ組であり、

1. S は有限集合である
2. $s_0 \in S$ である
3. T は有限集合であり、各 $a \in T$ は S から S への部分関数である
4. $V \colon S \to \mathfrak{P}(AP)$ である

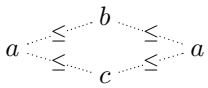
を満たすものをいうことにする。

遷移 $a \in T$ が $s \in S$ で実行可能であるとは $s \in \mathrm{dom}(a)$ であることをいう。実行列 $s_0 a_0 s_1 a_1 \cdots s_{n-1} a_{n-1} s_n$ とは任意の $i \ge 0$ で $a_i(s_i) = s_{i+1}$ をいう。$a_0 a_1 \cdots a_{n-1}$ をこの実行列の遷移列という。$w = a_0 \cdots a_{n-1}$ と書くことにしたとき $s_0 a_0 s_1 a_1 \cdots a_{n-1} s_n$ を $s_0 \overset{w}{\Rightarrow} s_n$ と書く。

対称的で非反射的な関係 $ID \subseteq T \times T$ が独立関係であるとは、$s \in S$ で実行可能である $\langle a, b \rangle \in ID$ に対して a が $b(s)$ で実行可能であり、かつ、$(a \circ b)(s) = (b \circ a)(s)$ であることをいう。

遷移列 $w = a_0 a_1 \cdots$ に対して $\langle a, b \rangle \in ID$ で $ab \equiv_{ID} ba$ を含む最小の同値関係を考える。w の同値類を $[w]$ と書く（Mazurkiewicz 列という）。

例 5.6. $T = \{a, b, c\}$、$ID = \{\langle b, c \rangle, \langle c, b \rangle\}$ とすると $w = abca$ について $[w] = \{abca, acba\}$ である。



命題 5.7. $s_0 \overset{w_0}{\Rightarrow} s_n$ かつ $s_0 \overset{w_1}{\Rightarrow} s'_n$ かつ $[w_0] = [w_1]$ であるとき $s_n = s'_n$ である。

証明. ID の定義、特に、$(a \circ b)(s) = (b \circ a)(s)$ から従う。 □

s_0 で実行可能な遷移の集合 T が s_0-永続的であるとは 任意の $0 \leq i < n$ で $a_i \notin T$ である実行列 $s_0a_0s_1a_1 \cdots a_{n-1}s_{n-1}a_n$ について、任意の $b \in T$ に対して $\langle a_{n-1}, b\rangle \in ID$ であることをいう。s_0 で実行可能な遷移集合全体 T は、$a_0 \notin T$ である a_0 が存在しないため、自明に s_0-永続的である。

s_0 を初期状態とする状態遷移系 $\mathfrak{S}$ に対して遷移関係を s_0-永続的な T_R に制限したものを $\mathfrak{S}_R$ と書くことにする。このとき以下が成り立つ。

補題 5.8. 状態 s_n を $\mathfrak{S}$ において任意の $a \in T$ に対して $s_n \notin \mathrm{dom}(a)$ であるとする。このとき $\mathfrak{S}$ の実行列 $s_0a_0s_1a_1 \cdots s_{n-1}a_{n-1}s_n$ に対してある i が存在して $a_i \in T_R$ である。

証明. いずれも属していないとすると s_0-永続的の定義より任意の $b \in T_R$ に 対して $\langle a_{n-1}, b\rangle \in ID$ でなければならない。独立関係の定義よりある s' が存在して $s_{n-1}a_{n-1}s_nbs'$ という実行列が存在しなければならないが、これは前提に反する。 □

定理 5.9. 状態 s_n が $\mathfrak{S}$ において任意の $a \in T$ に対して $s_n \notin \mathrm{dom}(a)$ であり、$\mathfrak{S}$ の実行列 $s_0a_0s_1a_1 \cdots s_{n-1}a_{n-1}s_n$ が存在するとき、s_n は $\mathfrak{S}_R$ においても s_0 から到達可能である。

証明. 実行列の長さに関する帰納法で示す。実行列の長さが 0 のときは明らかなので実行列の長さ n を $n > 0$ とする。補題 5.8 よりある i が存在して $a_i \in T_R$ である。そういった i のうち最小のものを j とする。任意の $0 \leq k < j$ について $a_k \notin T_R$ であることと $a_j \in T_R$ より a_k と a_j は独立なので、$s_0a_js'_1a_0 \cdots s_j$ というように a_j を先頭にもっていった実行列を $\mathfrak{S}$ においてつくれる。よって、長さ $n-1$ の $\mathfrak{S}$ の実行列 $s'_1a_0 \cdots s_ja_js_{j+1}a_{j+1} \cdots s_{n-1}a_{n-1}s_n$ に対して存在するので s'_1 から s_n へは $\mathfrak{S}_R$ において到達可能である。よって、s_0 から s_n に $\mathfrak{S}_R$ においても到達可能である。 □

到達可能性についてだけであれば s_0-永続的な遷移に限っても問題ないことを示した。実際は、示したい性質によっては独立関係をもっと細かくしたものに修正する必要がある。また、s_0-永続的な遷移の構成についての説明も省略した。最後に、今回は永続的（persisitency）だけをとりあげたが、ample、stubborn、sleep といったそれぞれ性質の異なるものも知られている。

第 6 章

組合せ検査

複数のプログラムを組み合わせると考慮すべき状態数が爆発的に増えてしまうことを説明する。また、その回避方法について説明する。

6.1 仕様

式 E を

$$E ::= c \mid x \mid E + E \mid E - E \mid E * E \mid \dots$$

といったように定数 c と変数 x と算術演算で定義されているものとする。また、逐次プログラム prg を

$$prg ::= \texttt{skip} \mid x := E \mid prg; prg$$

で定義する。`skip` は何もしないプログラム、$x := E$ は x に E を代入するプログラム、$prg_0; prg_1$ は prg_0 をした後に prg_1 をするプログラムをそれぞれ意図している。

状態をプログラム prg とメモリ M の組とする。ただし、メモリとは変数から値への関数である。また、付値 V は M から自然に定まるため、以下ではいちいち言及しない。

プログラムを実行すると状態遷移系が得られる。前述したプログラムの意図を反映した状態遷移が以下である。

$$\frac{}{\langle x := E, M\rangle \to \langle \texttt{skip}, M[x \mapsto [\![E]\!]_M]\rangle}$$

$$\frac{\langle prg_0, M\rangle \to \langle \texttt{skip}, M'\rangle}{\langle prg_0; prg_1, M\rangle \to \langle prg_1, M'\rangle} \qquad \frac{\langle prg_0, M\rangle \to \langle prg_0', M'\rangle}{\langle prg_0; prg_1, M\rangle \to \langle prg_0'; prg_1, M'\rangle}$$

代入文 $x := E$ を実行すると、M 中の x が E を M で解釈した値 $[\![E]\!]_M$ で更新される。$[\![E]\!]_M$ の定義は

$$[\![c]\!]_M = c \qquad [\![x]\!]_M = M(x) \qquad [\![E + E']\!]_M = [\![E]\!]_M + [\![E']\!]_M \quad \dots$$

である。メモリの更新を関数の更新 $M[x \mapsto n](v')$、つまり、

$$M[x \mapsto c](x') = \begin{cases} c & \text{if } x' = x \\ M(x') & \text{otherwise.} \end{cases}$$

で表す。

原子命題として式 E 間の等式 $E_0 = E_1$ や不等式 $E_0 \leq E_1$ をとる。逐次プログラムの仕様を三つ組 $\varphi\,\{prg\}\,\psi$ で書くことにする。これを仕様と呼ぶのは、φ が成り立つ下で prg を実行したら ψ が成り立つ、を表すものと意図しているからである。このとき、φ と ψ をそれぞれ s の事前条件、事後条件という。

例 6.1. $x = 0\,\{x \coloneqq x + 1\}\,x = 1$ である。

例 6.2. $x = 0\,\{x \coloneqq x + 1; x \coloneqq x + 1\}\,x = 2$ である。

6.2 非干渉性

本章では、並行プログラムをいわゆるフォークジョイン方式でなく、最初から N 個のスレッドが立ち上がっているもの、つまり、

$$s_0 \parallel \cdots \parallel s_i \parallel \cdots \parallel s_{N-1}$$

とする。

前節で定義した逐次プログラムの仕様を単に組み合わせたものは並行プログラムの仕様とは見なせない。例えば、$x = 0\,\{x \coloneqq x + 1\}\,x = 1$ であるし $x = 0\,\{x \coloneqq x + 2\}\,x = 2$ であるが、これら二つを単に組み合わせた

$$x = 0 \wedge x = 0\,\{x \coloneqq x + 1 \parallel x \coloneqq x + 2\}\,x = 1 \wedge x = 2$$

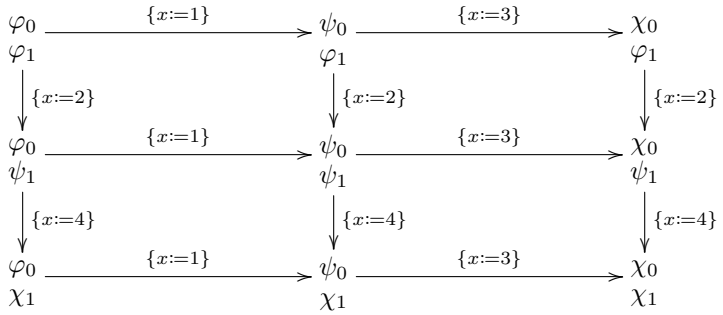
を仕様と見なすとおかしなことを言っているものとなる。

この理由は

- 事前条件 $x = 0$ に代入文 $x \coloneqq x + 2$ が干渉している
- 事後条件 $x = 1$ に代入文 $x \coloneqq x + 2$ が干渉している
- 事前条件 $x = 0$ に代入文 $x \coloneqq x + 1$ が干渉している
- 事後条件 $x = 2$ に代入文 $x \coloneqq x + 1$ が干渉している

つまり、代入文による状態の変更が条件成立の可否を変更するからである。

論理式 φ_0 が事前条件 φ_1 つき代入文 $x \coloneqq E$ に干渉されないというのを $\varphi_0 \wedge \varphi_1\,\{x \coloneqq E\}\,\varphi_0$ と定義する。任意のスレッドの事前事後条件がそれとは別のスレッドのすべての事前条件つき代入文から干渉されないとき、逐次プログラムどうしの仕様は組み合わせられる。それは以下の図からほぼ明らかである。



例 6.3. $x = 0 \vee x = 2\,\{x \coloneqq x + 1\}\,x = 1 \vee x = 3$ と $x = 0 \vee x = 1\,\{x \coloneqq x + 2\}\,x = 2 \vee x = 3$ は組み合わせることができ、$x = 0\,\{x \coloneqq x + 1 \parallel x \coloneqq x + 2\}\,x = 3$ となる。

例 6.4. $x = 0 \vee x = 1 \{x \coloneqq x + 1\} x = 1 \vee x = 2$ と $x = 0 \vee x = 1 \{x \coloneqq x + 1\} x = 1 \vee x = 2$ は組み合わせることができない。

例 6.5. $a_0 = 0 \wedge (a_1 = 0 \supset x = 0) \wedge (a_1 = 1 \supset x = 1) \{(x, a_0) \coloneqq (x + 1, 1)\} a_0 = 1 \wedge (a_1 = 0 \supset x = 1) \wedge (a_1 = 1 \supset x = 2)$ と $a_1 = 0 \wedge (a_0 = 0 \supset x = 0) \wedge (a_0 = 1 \supset x = 1) \{(x, a_1) \coloneqq (x + 1, 1)\} a_1 = 1 \wedge (a_0 = 0 \supset x = 1) \wedge (a_0 = 1 \supset x = 2)$ は組み合わせることができ、$x = 0 \wedge a_0 = 0 \wedge a_1 = 0 \{(x, a_0) \coloneqq (x + 1, 1) \parallel (x, a_1) \coloneqq (x + 1, 1)\} x = 2 \wedge a_0 = 1 \wedge a_1 = 1$ となる。ただし、$(x, a_0) \coloneqq (x + 1, 1)$ は $x \coloneqq x + 1$ と $a_0 \coloneqq 1$ の `atomic` な代入を意味する。

6.3 契約

非干渉性は仕様の組合せを可能にする十分条件であるものの、組み合わせる部品がすべてそろわないと確認を開始できないことと、スレッドの持つ代入文の増加にともなって確かめなくはならないことが爆発的に増加するという二つの欠点を持つ。これらの欠点を補う一つの方法が仕様の拡張である。

次変数 $\underline{x}$ を導入する。式 $\mathcal{E}$ を

$$\mathcal{E} ::= c \mid x \mid \underline{x} \mid \mathcal{E} + \mathcal{E} \mid \mathcal{E} - \mathcal{E} \mid \mathcal{E} * \mathcal{E} \mid \dots$$

で定義し、原子命題として式 $\mathcal{E}$ 間の等式 $\mathcal{E} = \mathcal{E}'$ や不等式 $\mathcal{E} \leq \mathcal{E}'$ をとれるようにする。このような拡張を施した論理式を R や G で書く。

次変数 $\underline{x}$ を持つ論理式は二状態、つまり、今のメモリ M に加えて次状態のメモリ $\underline{M}$ 込みで

$$\begin{aligned}
\mathfrak{S}, M, \underline{M} &\vDash \top \\
\mathfrak{S}, M, \underline{M} &\vDash \mathcal{E} = \mathcal{E}' \iff [\![\mathcal{E}]\!]_{M,\underline{M}} = [\![\mathcal{E}']\!]_{M,\underline{M}} \\
\mathfrak{S}, M, \underline{M} &\vDash \mathcal{E} \leq \mathcal{E}' \iff [\![\mathcal{E}]\!]_{M,\underline{M}} \leq [\![\mathcal{E}']\!]_{M,\underline{M}} \\
\mathfrak{S}, M, \underline{M} &\vDash \neg \psi \iff \mathfrak{S}, M, \underline{M} \nvDash \psi \\
\mathfrak{S}, M, \underline{M} &\vDash \varphi_0 \wedge \varphi_1 \iff \mathfrak{S}, M, \underline{M} \vDash \varphi_0 \text{ かつ } \mathfrak{S}, M, \underline{M} \vDash \varphi_1
\end{aligned}$$

と解釈する。ただし、$[\![\mathcal{E}]\!]_{M,\underline{M}}$ は

$$[\![c]\!]_{M,\underline{M}} = c \qquad [\![x]\!]_{M,\underline{M}} = M(x) \qquad [\![\underline{x}]\!]_{M,\underline{M}} = \underline{M}(x) \qquad [\![\mathcal{E} + \mathcal{E}']\!]_{M,\underline{M}} = [\![\mathcal{E}]\!]_{M,\underline{M}} + [\![\mathcal{E}']\!]_{M,\underline{M}} \quad \dots$$

である。

例 6.6. $\mathfrak{S}, M, \underline{M} \vDash x \leq \underline{x}$ は、メモリ M がメモリ $\underline{M}$ に更新されるにあたり変数 x の値が単調非減少であることを表している。

仕様を五つ組 $\varphi, R \{prg\} G, \psi$ に拡張する。事前条件 φ と依存条件 R 下でプログラム prg を実行したら保証条件 G と事後条件 ψ が成り立つ、を表している。各スレッドは R の下で φ と ψ が安定していること、つまり、φ かつ R ならば $\underline{\varphi}$ と ψ かつ R ならば $\underline{\psi}$ をあらかじめ確かめておく。ただし、$\underline{\varphi}$ は φ 中の変数 x をすべて $\underline{x}$ に置き換えたものである。また、prg を実行している間ずっと G が成り立ち続けていることもあらかじめ確かめておく。

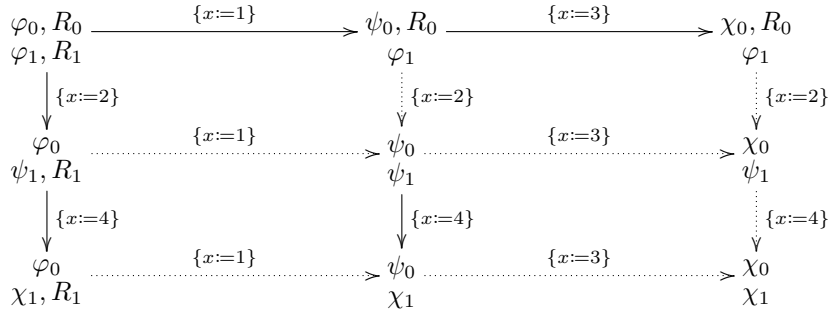
例 6.7. $x = 0$ は $x \leq \underline{x}$ の下で安定していない。

例 6.8. $0 \leq x$ は $x \leq \underline{x}$ の下で安定している。

例 6.9. $x := x + 1$ は $x \leq \underline{x} \wedge y = \underline{y}$ を保証している。

あるスレッドの依存条件 R とは他のスレッドたちがずっと保証してくれる条件である。他のスレッドたちがずっと保証してくれている条件なので依存することができる。あるスレッドの保証条件 G とは他のスレッドたちに保証する条件である。依存保証条件を契約ともいう。

仕様 $\varphi_0, R_0\,\{prg_0\}\,G_0, \psi_0$ と $\varphi_1, R_1\,\{prg_1\}\,G_1, \psi_1$ は、G_0 ならば R_1 と G_1 ならば R_0 が成り立っているとき、組み合わせられる。というのも、



で説明すると、スレッド 1 の $x := 2$ による遷移はスレッド 0 からすれば R_0 を保証しているものなので、この下で φ が安定していることはスレッド 1 が現れる前から既にスレッド 0 が自身で確かめておいたものにすぎないからである。このように、組み合わせる部品がすべてそろわないと確認を開始できないという欠点は克服されている。

また、非干渉性を確かめるには図中の矢印すべての合流性を調べなければならなかったのに対し、契約においては各スレッドがあらかじめ確かめておくべき図の実線だけから破線との合流性は自動的に得られる。このため、スレッドの持つ代入文の増加にともなって確かめなくはならないことが爆発的に増加するという欠点も克服されている。

例 6.10. $x = 0 \vee x = 2, (x = 0 \wedge \underline{x} = 2) \vee (x = 1 \wedge \underline{x} = 3)\,\{x := x + 1\}\,(x = 0 \wedge \underline{x} = 1) \vee (x = 2 \wedge \underline{x} = 3), x = 1 \vee x = 3$ と $x = 0 \vee x = 1, (x = 0 \wedge \underline{x} = 1) \vee (x = 2 \wedge \underline{x} = 3)\,\{x := x + 2\}\,(x = 0 \wedge \underline{x} = 2) \vee (x = 1 \wedge \underline{x} = 3), x = 2 \vee x = 3$ は組み合わせることができ、$x = 0\,\{x := x + 1 \parallel x := x + 2\}\,x = 3$ となる。

例 6.11. $x = 0 \vee x = 1, (x = 0 \wedge \underline{x} = 1) \vee (x = 1 \wedge \underline{x} = 2)\,\{x := x + 1\}\,(x = 0 \wedge \underline{x} = 1) \vee (x = 1 \wedge \underline{x} = 2), x = 1 \vee x = 2$ は正しい仕様ではない。なぜなら、$x = 0 \vee x = 1$ が $(x = 0 \wedge \underline{x} = 1) \vee (x = 1 \wedge \underline{x} = 2)$ の下で安定していないからである。

例 6.12. $a_0 = 0 \wedge (a_1 = 0 \supset x = 0) \wedge (a_1 = 1 \supset x = 1), ((a_1 = 0 \wedge x = 1) \vee (a_0 = 1 \wedge \underline{x} = 2)) \wedge \underline{a_1} = 1 \wedge a_0 = \underline{a_0}\,\{(x, a_0) := (x + 1, 1)\}\,((a_0 = 0 \wedge x = 1) \vee (a_1 = 1 \wedge \underline{x} = 2)) \wedge \underline{a_0} = 1 \wedge a_1 = \underline{a_1}, a_0 = 1 \wedge (a_1 = 0 \supset x = 1) \wedge (a_1 = 1 \supset x = 2)$ と $a_1 = 0 \wedge (a_0 = 0 \supset x = 0) \wedge (a_0 = 1 \supset x = 1), ((a_0 = 0 \wedge x = 1) \vee (a_1 = 1 \wedge \underline{x} = 2)) \wedge \underline{a_0} = 1 \wedge a_1 = \underline{a_1}\,\{(x, a_1) := (x + 1, 1)\}\,((a_1 = 0 \wedge x = 1) \vee (a_0 = 1 \wedge \underline{x} = 2)) \wedge \underline{a_1} = 1 \wedge a_0 = \underline{a_0}, a_1 = 1 \wedge (a_0 = 0 \supset x = 1) \wedge (a_0 = 1 \supset x = 2)$ は組み合わせることができ、$x = 0 \wedge a_0 = 0 \wedge a_1 = 0\,\{(x, a_0) := (x + 1, 1) \parallel (x, a_1) := (x + 1, 1)\}\,x = 2 \wedge a_0 = 1 \wedge a_1 = 1$ となる。

あとがき

本書はあくまで本トピックの概観を眺めるためのものである。詳細をきちんとおさえるには結局のところ [1] や [2] を読むことになる。もう少し日本語の本を読んでからにしたいというのであれば、良書はたくさんあり、例えば、[3] を勧める。

参考文献

[1] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. Model Checking. MIT Press, 2nd edition, 2018.

[2] Gerard J. Holzmann. The SPIN Model Checker. Addison-Wesley, 2003.

[3] 萩谷昌己, 吉岡信和, 青木利晃. SPIN による設計モデル検証. 日本評論社, 2008.