CSC227
Formal Specification of Software(3)

John Fitzgerald
Centre for Software Reliability
University of New Castle

Translated by
Takahiko Ogino
Railway Technical Research Institute

Based upon the book
Modelling Systems: Practical Tools and Techniques in Software Development
(ISBN 0 521 623480 Cambridge University Press 1998)
by John Fitzgerald and Peter Gorm Larsen (IFAD)

凡例

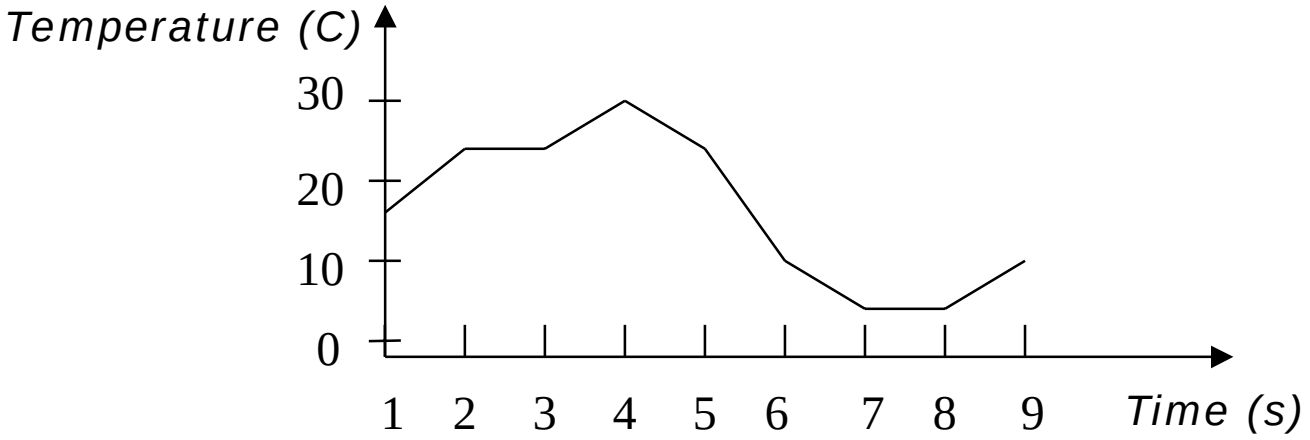
- 本翻訳は、John Fitzgerald博士（Centre for Software Reliability、University of New Castle）の講義用OHPを翻訳したものである。
- 翻訳にあたっては、なるべく忠実な訳をしたつもりである。ただし原稿が、授業の教材であることを勘案し、出来るだけ意味的な注釈を付けたり、空白になっておりクラスで埋められるのであろうと思われる部分は、出来るだけ講義のベースとなっている本、Modelling Systemsからの引用によって中身を埋めた。
- そのような部分をオリジナルと区別するため、フォントの色を変化させることで出来るだけ区別している。
- また、同じ鉄道総研の寺田夏樹君には、日本語と内容のプルーフリーディングをお願いしたことを記載し感謝の意を表します。
- 原OHPの修正も含めて全ての日本語版内容に関する責任は、荻野にあります。
- 誤訳も含めて、内容に関するご指摘をお待ちしています。

論理

不変式を宣言したり、事前条件、事後条件を記述する機能や、フォーマルモデルについて推論する機能は、モデル言語がよって立っている論理に依存する。

- 古典的論理の命題、および述語
- 連結記号 (Connectives)
- 限定記号／限定作用素（Quantifiers）
- 未定義(Undefinedness)の処理：
  LPF（Logic of Partial Function)部分関数の論理

温度モニターの例



モニターは最近の5つの温度の測定値を記録する。

25	*10*	*5*	*5*	*10*

温度モニターの例

下記に記述する状態はモニターによって検知される:

- **上昇**:最後の温度の読み取り値は、最初のものより大きい
- **上限越え**:サンプルの中に400℃を超える読み取り値がある。
- **連続した上限越え**:全ての読み取り値が400℃を超えている。
- **安全**:表示がサンプルの中間値までに400℃を超過しない場合、リアクターは安全である。もし、表示がサンプルの中間値までに400℃を超過した場合でも、もしサンプルの終端値での読み取りが400℃未満の場合は、リアクターはなお安全である
- **警報**:リアクターが安全で無い場合には、またその時のみ、警報が発せられなければならない。

モニターのフォーマルモデル:

```
Monitor :: temps : seq of int
           alarm : bool
    inv m == len m.temps = 5
```

述語　　命題

- 述語(predicate)は簡単に言えば論理式(logical expression)である
- 最も単純な種類の論理的な述語は命題(proposition)である。
- 命題は特定の値(複数可)に関する論理的な主張(logical assertion)である。通常は、値を比較するためにブール演算子を含んでいる。
  例:　　3<27　　5=9

- 命題は通常、真か偽である(しかし、VDMでは未定義の値(undifined values)を処理しなければならない－部分関数の論理(後述)を参照)。
- 命題の有用性は、非常に制限されている:
  変数という概念が無いため、いちいち評価する値を明示的に指定する必要があり、非常に使いづらい。命題を条件として使用するのは現実的でない(有りうる全ての値の組み合わせを明示する必要があるから)。

述語　　一般的な述語

述語は特定の値に特有でないが、可能な値の範囲のうちの1つを表わすことができる変数(複数)を含んでいる論理式である。例えば、

```
x < 27
```

```
(x**2)+x-6=0
```

述語の真理値は、変数が取る値に依存する。

述語　　モニターの例での述語

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

モニターmを考える。mは、ひとつのシーケンスであるので、インデックスを付けることができる。

- m中の最初の読み取り値: `m.temps(1)`
- m中の最後の読み取り値: `m.temps(5)`
- mの中の最初の読み取り値が厳密に最後の読み取り値未満であることを述べる述語: `m.temps(1) < m.temps(5)`

述語の真理値は、mの値に依存する。

述語　　上昇の条件

- サンプルの最新の読み取り値は、最初の値より大きい

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

- 上昇の条件をブール関数で表す。

```
Rising: Monitor -> bool

Rising(m) == m.temps(1) < m.temps(5)
```

- 任意のモニターmに対して、式Rising(m)はサンプルの最後の読み取り値が最初のものより大きいときのみ、またその時だけ、真となる。即ち、

```
Rising( mk_Monitor([233,45,677,650,900], false) )
-> true
```

基本的な論理演算

- 我々は、論理連結記号を使用して、単純な論理式からのより複雑な論理式を構築する:

`not`	否定(negation)
`and`	合接(conjunction)
`or`	離接(disjunction)
=>	含意(implication (if … then …))
<=>	両含意／同等(biimplication (if and only if))

基本的論理演算　　否定(Negation)

否定は、ある論理式の逆が真であるということを表明する操作である。

モニターmonの温度は、上昇していない:

```
not Rising(mon)
```

否定の真理値表:

A	not A
true	false
false	true

基本的論理演算　　離接(Disjunction)

離接(論理和)は、必ずしも排他的(exclusive)で無い論理的な選択／和を表す:

制限オーバー: サンプルの中に400℃を超える値がある。

```
OverLimit: Monitor -> bool
OverLimit(m) ==
      temp(1) > 400 or
      temp(2) > 400 or
      temp(3) > 400 or
      temp(4) > 400 or
      temp(5) > 400
```

A	B	A **or** B
true	true	true
true	false	true
false	true	true
false	false	false

基本的論理演算　　合接(Conjunction)

合接(論理積)は、述べられている要素の全てが真であることを表現するもの。

連続的に制限オーバー: *サンプル内の全ての値が400℃を超えている。*

```
COverLimit: Monitor -> bool
COverLimit(m) ==
     m.temps(1) > 400 and
     m.temps(2) > 400 and
     m.temps(3) > 400 and
     m.temps(4) > 400 and
     m.temps(5) > 400
```

A	B	A **and** B
true	true	true
true	false	false
false	true	false
false	false	false

基本的論理演算　　含意(Implication)

含意は、我々がある条件(“if…then”)のもとでのみ真実となる事実を示す:

安全:表示がサンプルの中間値までに400℃を超過しない場合、リアクターは安全である。もし、表示がサンプルの中間値までに400℃を超過した場合でも、もしサンプルの終端値での読み取りが400℃未満の場合は、リアクターはなお安全である

```
Safe: Monitor -> bool
Safe(m) ==
      temp(3) > 400 =>
            temp(5) < 400
```

A	B	A => B
true	true	true
true	false	false
false	true	true
false	false	true

基本的論理演算　両含意(Biimplication)

両含意は、同等(“必要十分条件”)を表わす。

警報:リアクターが安全でない場合、またその場合のみ警報が鳴らされる。
これは、不変式を用いて記録される。

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5 and
     not Safe(temps) <=> alarm
```

A	B	A <=> B
true	true	true
true	false	false
false	true	false
false	false	true

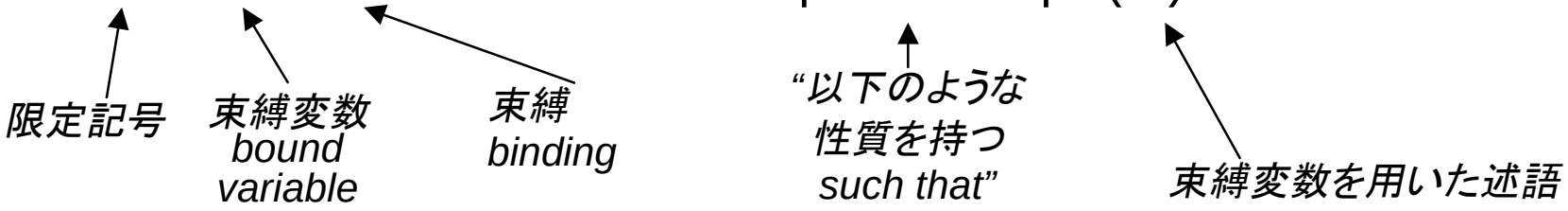
限定記号（Quantifiers)

- 値の大きな集りに対しては、個々の値をいちいち取り扱うより、変数を使用するのが通常の考え方である。

  inds m.temps　　サンプルのインデックス(1－5)を表す。

制限値越えの条件は、次のように簡便に表される。

```
exists i in set inds m.temps & temps(i) > 400
```



- 連続的に制限値越えをする条件は、“forall”を用いて表せる。

限定記号

- シンタックス: forall(全称作用素) exists(存在作用素)
  - `forall` *binding & predicate*
  - `exists` *binding & predicate*

- 2つの束縛(binding)がある:
  - **Type Binding**, e.g.
  - `x:nat`
  - `n: seq of char`

  タイプ束縛は、束縛変数の値のレンジ(恐らく無限集合)にタイプの値を束縛する。

  - **Set Binding**, e.g.
  - `i in set inds m`
  - `x in set {1,…,20}`

  セット束縛は有限集合上の値に束縛変数を束縛させる。(*VDMの集合が有限である制約による*)

限定記号

- いくつかの変数を纏めて、単一の限定記号で束縛を表しても良い。

```
forall x,y in set {1,…,5} &
        not m.temp(x) = m.temp(y)
```

- この述語はm.tempの次の値に対して真となるであろうか？

[320、220、105、119、150]

答え: 否

例えばx=3, y=3。x <> y となることは保証されない。従って

```
forall x,y in set {1,...,5} &
    not (x = y) => not (m.temp(x) = m.temp(y))
```

とすべきである。

限定記号　　練習問題

- サンプル中の全ての読み取り値は、400未満および50を越えるものである。(50は含まず)

```
forall i in set inds temp & temp(i)<400 and temp(i)>50
```

- サンプル中の各々の読み取り値は、その前の読み取り値より、高々10大きい。

```
forall i in set inds temp¥{1} &
    temp(i-1)>temp(i) and temp(i-1)+10>=temp(i)
```

- 400以上であるサンプルの中に、2つの異なった読み取り値がある。

```
exists i,j in set inds temp &
                i<>j and temp(i)>400 and temp(j)>400
```

限定記号　　順序

- 我々が次の特性を形式化しなければならないと仮定する:
  読み取り値のシーケンスに“single minimum”がある、つまり、その読み取り値が他のいかなる読み取り値より厳密に小さい。

```
exists min in set {1,...,5} &
        forall i in set {1,...,5} &
                i <> min => temp(i) > temp(min)
```

  – ある値**min**が集合中に存在して、その値に対して集合の全てiに対して以下の式が真である。(与えられた値に対して全て)

- 限定記号の順序が異なったとするとどうなるか？

```
forall i in set {1,...,5} &
        exists min in set {1,...,5} &
                i <> min => temp(i) > temp(min)
```

  – 集合の全ての**i**の値に対して、ある値**min**が集合中に存在して以下の式が真である。(与えられた値に対して、存在する)
  — **i=min**と選ぶと、前提が偽であるため、この式は真となる。ゆえに、恒に真な述語である。

まとめ

- 命題
- 述語は、自由変数を含む
- 述語は、連結記号を用いて連結させる事が出来る。
- 自由変数は、限定記号を用いて値の集まりの上で値を取らせる事が出来る。
- 限定記号はミックスできる。

LPF: 未定義との共存

- センサーは故障し、正しい温度の代わりに値「エラー」を生成することがあると仮定する。
- この場合、我々は値「エラー」を次のような形で比較することができない
  エラー < 400
- VDMの論理は演算子が例えば0で何かを割ったときのように未定義になることを扱うことが出来る。（現実的な問題に対処するため）
- 真理値表は未定義の値を対処するために拡張され次のようになる。

A	not A
true	false
false	true
*	*

＊は未定義の値を示す。

LPFにおける離接（Disjunction)

A	B	A **or** B
true	true	true
true	false	true
true	*	true
false	true	true
false	false	false
false	*	*
*	true	true
*	false	*
*	*	*

離接の中の単一の項が真の場合、全体の離接は、他の項が真、偽、未定義にかかわらず真である。

LPFの合接（Conjunction)

A	B	A **and** B
true	true	true
true	false	false
true	*	*
false	true	false
false	false	false
false	*	false
*	true	*
*	false	false
*	*	*

もし合接の中のひとつの項が偽である場合、全体の合接は、その他の項が、真、偽、未定義にかかわらず偽となる。

含意（Implication)と同等（Biimplication）

A	B	A => B
true	true	true
true	false	false
true	*	*
false	true	true
false	false	true
false	*	true
*	true	true
*	false	*
*	*	*

A	B	A <=> B
true	true	true
true	false	false
true	*	*
false	true	false
false	false	true
false	*	*
*	true	*
*	false	*
*	*	*

練習問題(ホームページからの引用)

変数に次の値が割当てられた場合、論理式を評価しなさい。:

```
a:=0, b:=34, c:= 90,  d:=1, e:=true, f:=false
1. not (e and f)
2. (e or f) => (not e and not f)
3. (e and f) <=> (not e and not f)
4. a < b and a = b
5. a*b > c or a+c>d or true
6. (a=0 or b < 23) <=> (c=4 and d=5 and d=44)
7. (a+b > 2 => c+d > 3) => a=12
8.  a+b > 2 => (c+d > 3 => a=12)
```

```
1.not (true and false)
     = not false
     = true
2.(true or false) => (true => false)
     = true => false
     = false
3.(true and false) <=> (not true and not false)
     = false <=> (false and true)
     = false <=> false
     = true
4.0 < 34 and 0 = 34
     = true and false
     = false
5.0*34 > 90 or 0+90 > 1 or true
     = 0>90 or 90 > 1 or true
     = true
6.(0=0 or 34>23) <=> (90=4 and 1=5 and 1=44)
     = true <=> false
     = false
7.(0+34 > 2 => 90+1>3) => 0=12
     = (true => true) => false
     = true => false
     = false
8.0+34>2 => (90+1>3 => 0=12)
     = true => (true => false)
     = true => false
     = false
```

フォーマティング

- フォーマットは、論理的には何の意味も足さないが、論理式が人間に対して分りやすくなる。
- 次の式に評価の順を示すように括弧をつけよ。p, q, r, s と t は任意の論理式を表すとする。

1. `p and not not not not q <=> r`
2. `p and q or r <=> s and t`
3. `p => q <=> r => s <=> p => q or s`
4. `(p and q => r) <=> p and q or r => c and d`

- 括弧のない表現を書くことは大変悪いスタイルである:それはモデルのリーダビリティを悪くする。括弧を多用して、モデルが自分自身や想定している相手に分かりやすくなるように、ページ上の表現を工夫しなさい。

フォーマッティングの答え

```
1.(p and (not (not (not (not q))))) <=> r

2.((p and q) or r) <=> (s and t)

3.((p => q) <=> (r => s)) <=> (p => (q or s))

4.((p and q) => r) <=> (((p and q) or r) => (c and
  d))
```

練習問題

- 次の文章を論理式に変換しなさい:

1. 集合 {7,55,133,200} の全ての数は、5 より大きい。
2. 50未満の自然数がある。
3. 掛け合わせると60になる2つの自然数がある。
4. 数24は偶数である。
5. 任意の2つの連続する自然数に対して、それらの一つは偶数である。
6. その直後の数が偶数である偶数は無い。
7. 任意の自然数の対にたいして、それらの商に等しい有理数が存在する。(これは本当ではない:割るほうの数はゼロではいけない。この制約を持つ式をかけ。)

解答

1. `forall x in set {7,55,133,200} & x > 5`
2. `exists x:nat & x < 50`
3. `exists x,y:nat & x*y = 60`
4. `exists factor:nat & 2*factor = 24`
5. `forall x:nat & (exists factor & 2*factor = x)`
   `or (exists factor & 2*factor = x+1)`
6. `not exists x:nat &`
   `(exists factor & 2*factor = x) and`
   `(exists factor & 2*factor = x+1)`
7. `forall x,y:nat & exists q:rat & q = x/y`

   ゼロで割ってはならないことを書く場合

   `forall x:nat, y:nat1 & exists q:rat & q = x/y`

練習問題

- 次の式を論理積、論理和で表せ:

```
1.forall x in set {1,...,5} & 2*x < 10
2.exists y in set {1,...,5} & y*y = 9
3.exists x,y in set {1,...,3} & x*y = y*y
```

- 次の式のうちどれが true で、どれが false か？

```
1.forall i in set {1,...,10} & i < 50
2.forall i : nat1 & i < 10000
3.exists n:nat & n < 0
4.exists i:int & i > -3 and i+20 < 5
5.forall i:nat & (exists j:nat & i*j=10) =>
      i in set {2,5}
6.forall i,j in set {3,4,5} & i*j <> 12
7.not exists i,j in set {3,4,5} & i*j = 9
8.exists x,y:nat & not exists r:nat & x*r=y
```

解答

- 展開:
  1. 2*1 < 10 and 2*2 < 10 and ... and 2*5 < 10
  2. 1*1 = 9 or 2*2 = 9 or ... 5*5 = 9
  3. 1*1 = 1*1 or 1*2 = 2*2 or 1*3 = 3*3 or
     2*1 = 1*1 or 2*2 = 2*2 or 2*3 = 3*3 or
     3*1 = 1*1 or 3*2 = 2*2 or 3*3 = 3*3

- 計算:
  1. true
  2. false
  3. false
  4. false
  5. false, e.g. i=1, j=10
  6. false, e.g. i=3, j=4
  7. false, e.g. i=j=3
  8. true, e.g. x=2,y=3

恋愛関係

- 次の命題を個人の **Name** の集まりの上に定義されるVDM-SLのブール関数に変換し、テストせよ。

  1. Somebody is loved by everybody

     SomebodyIsLovedByEverybody: ? -> ?

  2. Nobody loves everybody

     NobodyLovesEverybody: ? -> ?

  3. If you love somebody, you love some of those he/she loves, too

     TransitiveLove: ? -> ?

ヒント: (1) **Name** を **token** タイプとして定義する;

(2) タイプ **Lovers** を名前から各々が愛する人(人々)の名前へのマッピングとする。

(3) テストの為にタイプ**Lovers** の値を定義する。さらに代案として、名前のペアの集合としてモデルされる名前の間の関係が採用できるか否か考えることも出来る。

恋愛関係（解答例）

```
types

Name = token;

People = set of Name;

Lovers = map Name to People
inv m == not ({} in set rng m);
-- pがsの全ての人を愛しているとき p |-> S がマップに含まれる
-- 誰でも少なくとも一人は愛している。

Result ::   whom: [People]     --オプショナルタイプ
            flag: bool;        --結果のブール値
```

恋愛関係（解答例）

```
functions

Loves: Name * Name * Lovers -> bool
Loves(p,q,L) ==                    -- p が q を愛する
  p in set dom L and q in set L(p);

SILBE : Lovers * People -> Result
-- SomebodyIsLovedByEverybody の頭文字を採った関数名
-- 全ての人に愛されている人たちの集合を返し、ブール関数を真とする。
SILBE(L,C) ==
    let a = dinter {L(x)| x in set dom L}
            in let b = (C = dom L) and (a <> {})
--全ての人をチェックし、結果が空で無い時、bは真となる。
    in
      mk_Result(if b then a else nil, b);
```

Letの説明

```
let pattern = defining-expression
   in
        use-expression
let a = dinter {L(x)| x in set dom L}
            in ( let b = (C = dom L) and (a <> {})
            in  mk_Result(if b then a else nil, b) );
```

この構文は次のように評価される:

- *defining-expression* を評価し、パターン(この場合は、a)に対してその結果を対応させる。
- *use-expression* 中のパターンの中の識別子(この場合は単にa)を、上のステップで得た値で置き換えて*use-expression* を評価する 。

恋愛関係（解答例）

```
SILBE1: Lovers * People -> bool
-- 関数Lovesを用いた代案
SILBE1(L,C) ==
   exists  i in set C  & forall j in set C &
  Loves(j,i,L);
-- テストデータの例
-- これに対して SILBE(CC1,C) = mk_Result(nil, false)
--              SILBE(CC2,C) = mk_Result({C3}, true)
C1: Name = mk_token(“jim“);
C2: Name = ... ;
…
C: People = {C1, C2, C3, C4, C5, C6};
CC1: Lovers    = { C1 |-> {C2, C3}, C4 |-> {C5,C6}};
CC2: Lovers
  = {C1 |-> {C2, C3}, C2 |-> {C1,C3,C5}, C3 |-> {C3},
     C4 |-> {C3,C6}, C5 |-> {C3,C6}, C6 |-> {C2,C3}};
```

恋愛関係（解答）

```
NLE: Lovers * People -> bool
-- Nobody Loves Everybody
NLE(L,C) ==
forall i in set C &
   exists j in set C &
        not Loves(i,j,L);

not (exists i in set C & ( forall j in set C & Loves(i,j,L)))
== forall i in set C & not( forall j in set C & Loves(i,j,L))
== forall i in set C & exists j in set C & not Loves(i,j,L)

Not(forall x in set A & P(x)) |- exists x in set A & not P(x)
Not(exists x in set A & P(x)) |- forall x in set A & not P(x)

-- Tests: for (CC1,C) - true, for (CC4,C) false.
CC4: Lovers = { x |-> C | x in set C};
```

恋愛関係（解答）

```
TransLove: Lovers  -> bool
-- Transitive Love
TransLove(L) ==
  let R = dunion rng L in      -- 愛されている人々の集合
   let D = dom L in            -- 愛している人の集合
   forall i in set D &
    forall j in set R & -- i によって潜在的に愛されている
     Loves(i,j,L) and (j in set D) =>
  -- iが jを愛しており、且つ jが誰かを愛しているならば
       exists k in set L(j) & Loves(i,k,L);
  -- jの愛している人の中にiが愛している人kが存在する

CC2: Lovers
   = {C1 |-> {C2, C3}, C2 |-> {C1,C3,C5}, C3 |->{C3},
     C4 |-> {C3,C6}, C5 |-> {C3,C6}, C6 |-> {C2,C3}};
```