CSC227
Formal Specification of Software(4)

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。

フォーマルモデルの基本的な要素

- VDM-SLの論理を説明したので、次に主要なデータ型および言語のオペレータを説明する。
- フォーマルモデルがタイプ定義と関数定義から形成されることを思い出してもらいたい。従ってデータの表現は、有用なフォーマルモデルを展開する上で非常に重要である。
- 型とオペレータの定義:全オペレータ、部分オペレータ
- 交通信号コントロールの核:
  - 引用タイプ(Quote types)
  - タイプの結合(Type union)
  - 混合タイプ(Composite types)
  - トークンタイプ(Token types)

タイプの定義

- タイプ定義には、次の要素が必要である。
  - タイプシンボル
  - (タイプの採る)値を記述する方法
  - 値を操作するオペレータ
  - 値の比較をするオペレータ

- それぞれのオペレータに対して、シグネチュアを与える。

  ＋:

  ／:

タイプを定義する。部分オペレータ

オペレータは

```
op : T1 * . . . * Tn  ->  R
```

任意のa1:T1,. . . an:Tnに対して、次の式が定義されるとき、全(total)オペレータといわれる。

```
op(a1,. . ., an)
```

次の式が未定義となるb1:T1,. . .,bn:Tnが存在するとき、

```
op(b1m,. . .,bn)
```

opは部分(partial)オペレータであるといわれる。

未定義となる値を部分オペレータに適用することを避ける！

基本タイプ

Type Symbol	Values	Example Values	Operators
nat	0 を含む自然数	0, 1, 2, …	+,-,*,/,…
nat1	nat　0を除く	1, 2, 3, …	+,-,*,/,…
int	整数	…,-1,0,1,2,…	+,-,*,/,…
real	実数	-23.334	+,-,*,/,…
char	文字	‘g’, ‘@’	=
bool	ブール値	true, false	and, or, …
token	構造のないトークン	*Not applicable*	=
quote	名前付き引用	<Red>, <Bio>	=

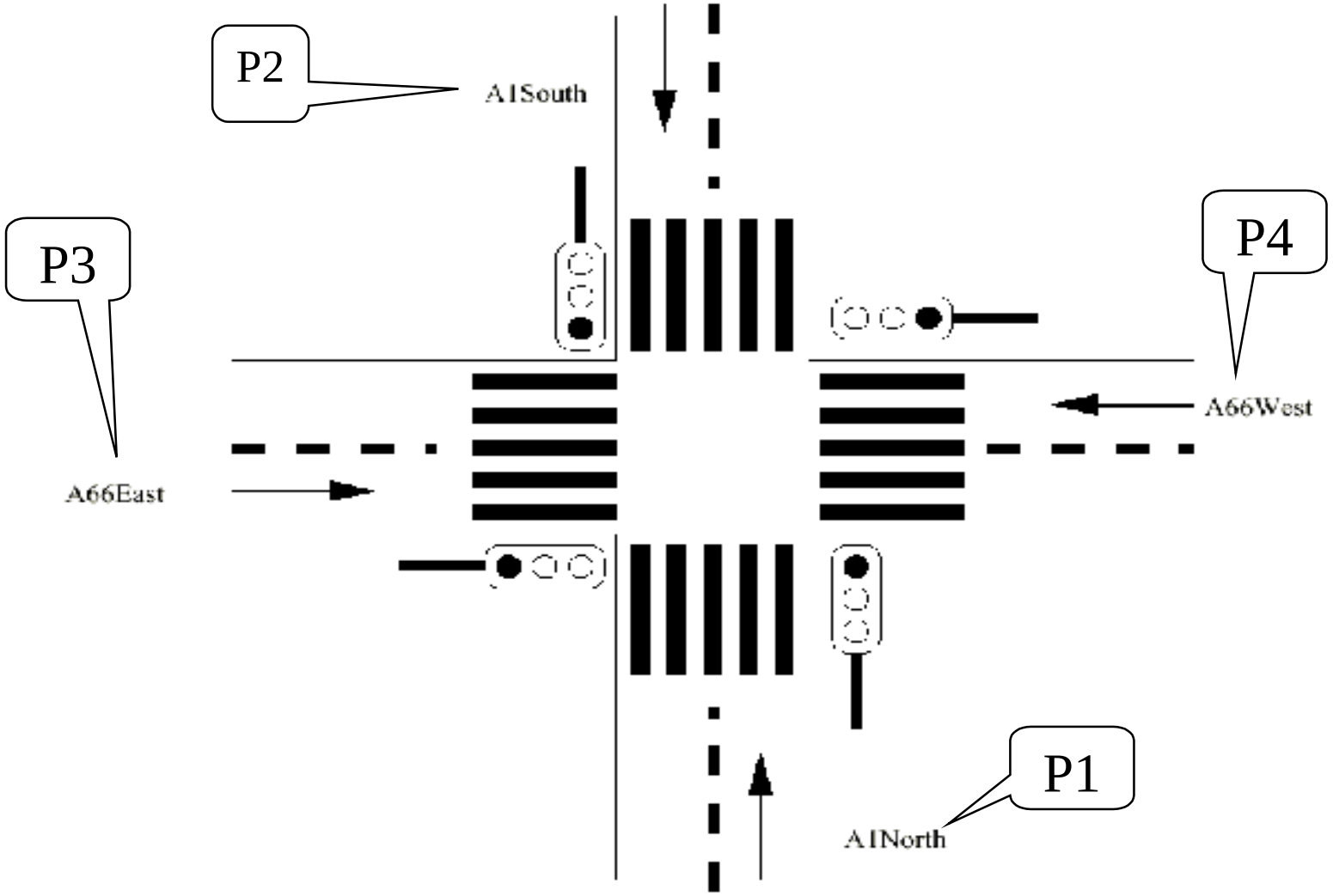
タイプ構成子

`|`	ユニオンタイプ(Union types)
`[_]`	オプショナルタイプ(Optional types)
`::`	レコードタイプ(Record types)
`set of _`	有限集合(Finite sets)
`seq of _`	有限シーケンス(Finite sequences)
`map _to _`	有限マッピング(Finite mappings)

- ユニオンタイプ　複数タイプを結合したもの（このタイプの値は、結合されたどれかのタイプであることを示している。）
- オプショナルタイプ　特別な値 nil をとることができる。

```
Light = <Red> | <Amber> | <Green>;
LightFail = Light | <Dark>
LightFail = [Light] = Light | nil
```

交通信号制御カーネル

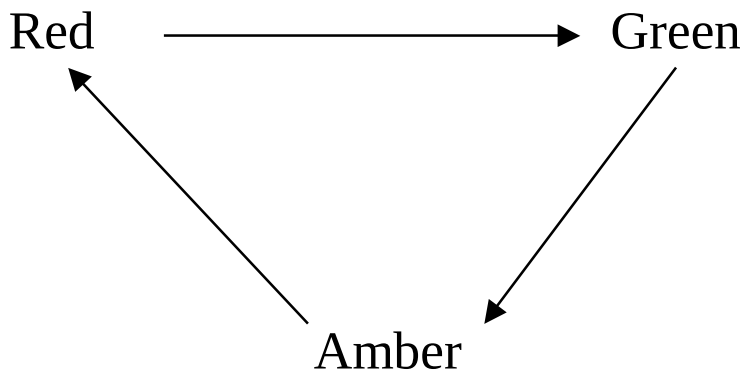


交通信号制御カーネル　　安全要求仕様

S1:2つの通り道が交わっている場合、それらの道のうち一方の信号機の表示が常に赤で無ければならない。

S2:信号機の表示が赤になり、対応するもう一方の信号機の表示が青になる間の時間は、少なくとも5秒以上なければならない。

S3:表示が黄色になりそれが赤になるまでの間は少なくとも5秒以上なければならない。



交通信号制御カーネル　　ユニオン／引用タイプ

```
Light = <Red>|<Green>|<Amber>
```

- タイプ<Red> は唯一の値<Red>(表現上は同じ)を持つタイプである。これは引用タイプと呼ばれる。従って、次のように書ける:

```
<Red> : Light
<Red> : <Red>
```

引用リテラルは、一致(=)か不一致(<>)でのみで比較することが出来る。:

```
x,y : Light
x = y
x <> y
```

交通信号制御カーネル

数値タイプ

```
Time = real
```

または、時間をミリ秒で表し、

```
Time = int
```

時間に関して不変式はあるか？

例えば `inv t == t >= 0`

交通信号制御カーネル　　トークンタイプ

```
Path = token
```

- 我々が特定のタイプの表現の詳細が必要でない場合、トークンタイプが使用される。
- トークンタイプを用いる一般的なガイドラインは、モデルの機能がそのタイプの値にアクセス（変更や値を使った計算など）しない時である。
- トークンタイプの値は=および<>によってのみ比較される。
- トークンを用いる場合
- トークンはモデルを抽象化するものである。
- 実際にインプリメントする場合には当然その中身を規定することになる。
- 値の区別だけを問題にする場合はトークンを使用できる(すべきである)。
- 数値を用いることが想定される場合でもその大小や順序等が問題にならない限り、トークンを使用できる。

交通信号制御カーネル　　トークンタイプ

定数はVDM-SLではvaluesと呼ばれる:

value definitions = `values', value definition , { `;' , value definition };

value definition = name , [`:' , type ] , `=' , expression;

```
values
p1 : Path = mk_token("A1North")
p2 : Path = mk_token("A1South")
p3 : Path = mk_token("A66East")
p4 : Path = mk_token("A1North")
```

- "**mk_token**"構成子および()の中の任意の値の使用により、tokenの値を表すことが出来ることに注意。
- 厳密に言えば、これはISOの標準VDM-SLの拡張であり、標準では、tokenタイプの値は調べられたり、構成されたり出来ないことになっている。

交通信号制御カーネル　　レコードタイプ

```
Conflict :: path1 : Path
            path2 : Path
```

他のタイプと異なり=は使わず::である。

構成子(Constructor)

```
record constructor = `mk_`, name, `(`, [expression list], `)`
mk_Conflict(mk_token("A1North"),mk_token("A66East"))
mk_Conflict(p1,p3)
```

- 選択子(Selectors)

```
field select = expression , `.`, identifier
```

所謂、ドットノーテーション　例えば con:Conflict に対し con.path1

- 不変式は?　任意の通り道はそれ自体とは交差(競合)しない。

```
inv mk_Conflict(p1,p2) == p1 <> p2
```

交通信号制御カーネル　　レコードタイプ

- タイプConflictの値を作成するのに、タイプ構成子を使用することが出来る。

```
mk_Conflict(mk_token(“A1North”),mk_token(“A66East”))
```

- すでに定義されたvalue定義を用いて、次のようにも書ける。

```
mk_Conflict(p1,p3)
```

交通信号制御カーネル　　レコードタイプ

```
Kernel :: lights    : Light
          conflicts : set of Conflict
```

Example values:

```
conflicts: set of Conflict =
             { mk_Conflict(p1,p3),
               mk_Conflict(p1,p4),
               mk_Conflict(p2,p3),
               mk_Conflict(p2,p4),
               mk_Conflict(p3,p1),
               mk_Conflict(p4,p1),
               mk_Conflict(p3,p2),
               mk_Conflict(p4,p2)}
```

交通信号制御カーネル　　レコードタイプ

- 制御カーネルの信号機コンポーネントの値の例:

```
lights : map Path to Light
   = { p1  |->  <Red>,
       p2  |->  <Red>,
       p3  |->  <Green>,
       p4  |->  <Green> }
```

交通信号制御カーネル　　レコードタイプ

制御カーネルの不変式:

```
Kernel :: lights    : map Path to Light
          conflicts : set of Conflict

inv mk_Kernel(ls,cs) ==
```

-- 交差している通り道のそれぞれに信号機がなければ成らない。

```
forall c in set cs &
    c.path1 in set dom ls and
    c.path2 in set dom ls
```

-- それぞれ交差している通り道のうちのひとつの信号機は、赤を表示していなければならない。

```
forall c in set cs &
    c.path1 in set dom ls and
    c.path2 in set dom ls and
       (ls(c.path1) = <Red> or ls(c.path2) = <Red>)
```

-- 交差は反射的(reflexive)である。

上記の不変式を強化せよ。

交通信号制御カーネル　　直接定義関数

```
f: T1 * . . . * Tn -> R
```
シグネチュア

```
f(a1,…,an) == 結果を示す式
pre 仮定の論理(ブール)式
```

信号を青に変化させる関数

```
ToGreen: Path * Kernel  ->  Kernel
ToGreen(p, mk_Kernel(lights,conflicts)) ==
mk_Kernel(ChgLight(lights,p,<Green>),conflicts)
```

他の色のための関数も類似したものである＝>*信号を黄色、赤に変える関数を作成せよ。*

何らかの事前条件があるか？

交通信号制御カーネル　　直接定義関数

```
ToGreen: Path * Kernel -> Kernel
ToGreen(p, mk_Kernel(lights,conflicts)) ==
    mk_Kernel( lights ++ { p |-> <Green> },
               conflicts)
```

マップの上書き

pre　-- 道pは信号機がある。 --(1)

　　-- 通り道pの信号機の現示は、赤である。 --(2)

　　-- pに交差している全ての通り道の信号機の現示は赤である。--(3)

```
pre
   p in set dom lights and --(1)
   lights(p) = <Red> and   --(2)
   forall con in set conflicts &
      (con.path1 = p => lights(con.path2) = <Red>) and --(3)
      (con.path2 = p => lights(con.path1) = <Red>)
```

正しいが冗長(！)

交通信号制御カーネル　　直接定義関数

```
ToRed: Path * Kernel -> Kernel
ToRed(p, mk_Kernel(lights,conflicts)) ==
   mk_Kernel( lights ++ { p |-> <Red> },
              conflicts)
pre p in set dom lights and
    lights(p) = <Amber>
```

交差道の状況を調べる必要は無い。

```
ToAmber: Path * Kernel -> Kernel
ToAmber(p, mk_Kernel(lights,conflicts)) ==
   mk_Kernel( lights ++ { p |-> <Amber> },
              conflicts)
pre p in set dom lights and
    lights(p) = <Green>
```

交通信号制御カーネル　　時間の足し算

```
Kernel :: lights    : map Path to Light
          conflicts : set of Conflict
          lastch    : map Path to Time
```

その道の信号の変化した最後の時間

```
inv mk_Kernel(ls,cs) ==
   dom ls = dom lc and
   forall c in set cs &
      mk_Conflict(c.path2, c.path1) in set cs and
      c.path1 in set dom ls and
      c.path2 in set dom ls and
      (ls(c.path1) = <Red> or ls(c.path2) = <Red>)
```

交通信号制御カーネル　　時間の足し算

```
ToGreen: Path * Kernel * Time -> Kernel
ToGreen(p, mk_Kernel(lights,conflicts,lastch),clock)==
   mk_Kernel( lights ++ { p |-> <Green> },
              conflicts,
              lastch ++ { p |-> clock})
pre p in set dom lights and
    lights(p) = <Red> and
    forall con in set conflicts &
      p = con.path1 => ( lights(con.path2) = <Red> and
                         (clock-lastch(con.path2)) >= 5)
```

forallなので、必ず自分がpath1になっているデータがある。

lights とlastchの定義域は不変式により一致していることに注意

交差している道が赤になってから自分が青になるのに5秒以上

交通信号制御カーネル　　時間の足し算

```
ToRed: Path * Kernel * Time -> Kernel
ToRed(p, mk_Kernel(lights,conflicts,lastch),clock) ==
   mk_Kernel( lights ++ { p |-> <Red> },
              conflicts,
           lastch ++ { p |-> clock})
pre p in set dom lights and
    lights(p) = <Amber> and
    clock-lastch(p) >= 5
```

自分が黄色になってから赤になるには5秒以上

交通信号制御カーネル　　時間の足し算

```
ToAmber: Path * Kernel * Time -> Kernel
ToAmber(p, mk_Kernel(lights,conflicts,lastch),clock)
  ==
   mk_Kernel( lights ++ { p |-> <Amber> },
              conflicts,
              lastch ++ { p |-> clock})
pre p in set dom lights and
    lights(p) = <Green>
```

交通信号制御カーネル　　要求仕様のレビュー

S1: 2つの通り道が交差する場合、交差する通り道のうちの1つの信号機が必ず赤でなければ成らない。

S2: 信号機が赤になる時と、交差している通り道の方向の信号機が青になる時間との間に、少なくとも5秒の遅れがなければならない。

S3: 信号機が黄色になるのと、その信号機が赤になるのとの間には少なくとも5秒の遅れがなければならない。

交通信号制御カーネル　　オプショナルタイプ

```
Light = <Red> | <Amber> | <Green>
```

- 電球のフィラメントの切断などを示す値を簡単に増やす方法：オプショナルタイプを用いる。

```
LightFail = [Light]
          =  <Red> | <Amber> | <Green> | nil
```

- 次の式は真である。

```
nil <> <Red>
nil : LightFail
```

まとめ

- タイプの定義:タイプのシンボル、値を表わす方法およびオペレータを与える。
- オペレータは全関数かもしれないし、部分関数かもしれない。部分関数の場合は、アプリケーションが定義域以外の値を取らないように回避する。
- 基本的なタイプ: 数値、文字、ブール値、token、引用タイプ
- タイプ構成子:ユニオン、レコード、optionals、有限集合、シーケンスおよびマッピング。