Interactive Systems and Agapia Programming

Gheorghe Stefanescu

Faculty of Mathematics and Computer Science
University of Bucharest

Contents:

- *Generalities*
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions

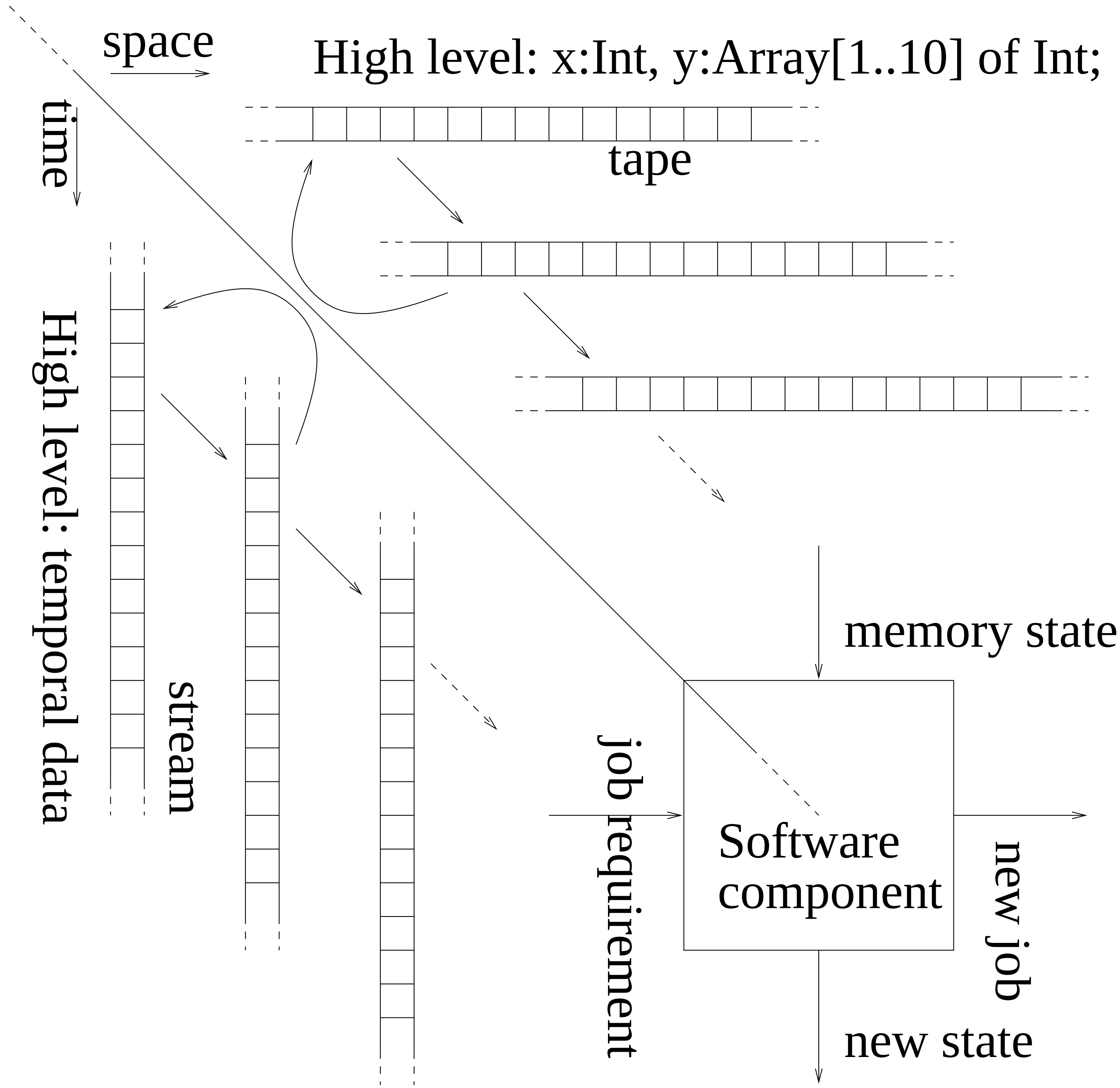
History

History

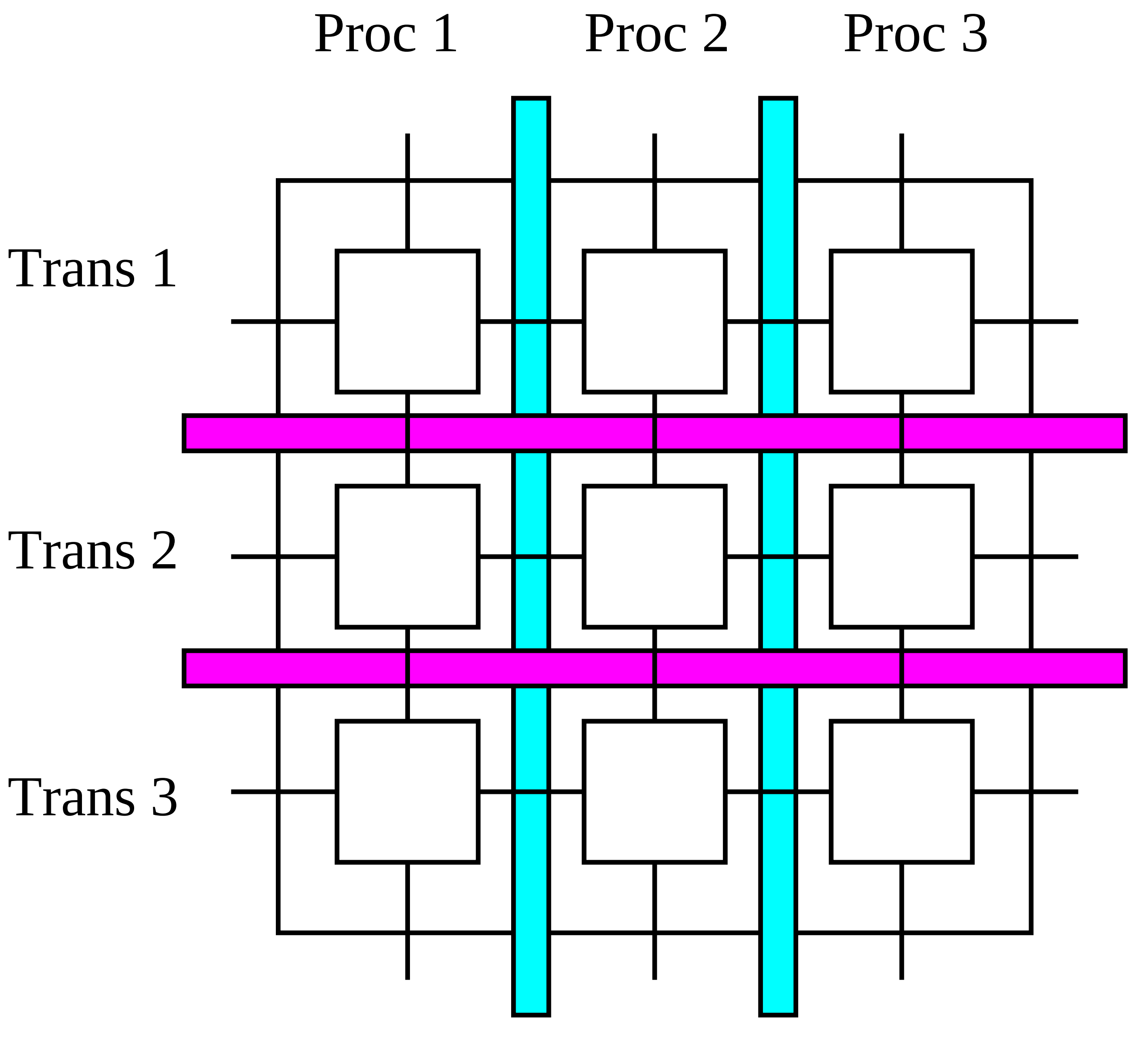
- *space-time duality "thesis"*
  - Stefanescu, *Network algebra*, Springer 2000
- *finite interactive systems*
  - Stefanescu, Marktoberdorf Summer School 2001
- *rv-systems* (interactive systems with registers and voices)
  - Stefanescu, NUS, Singapore, summer 2004
- *structured rv-systems*
  - Stefanescu, Dragoi, fall 2006

ST-Dual picture

ST-Dual picture



Processes and transactions



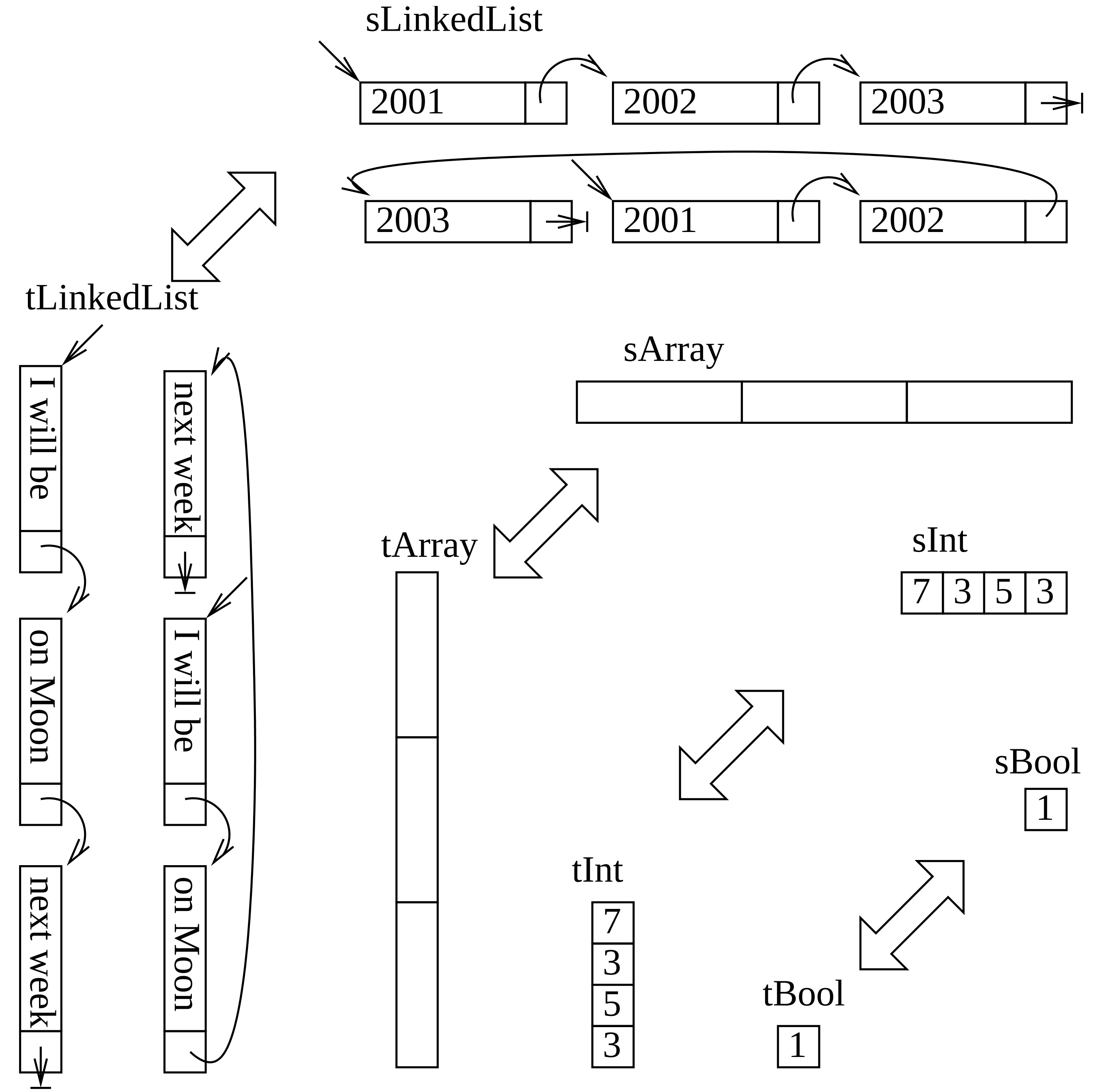
High level temporal structures

data with usual (*spatial*) representation:

sBool, sInt, sArray, sLinkedList, etc.

and their *time dual* (i.e., data with *temporal representation*):

tBool, tInt, tArray, tLinkedList, etc.



..High level temporal structures

Three allocations of a temporal linked list on a stream: The 1st starts at time $t = 10$ and is

time :	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
	..	I		w	i	l	l		b	e		o	n		M	o	o	n		n	e	x	t		w	e	e	k	..
	..	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	$\perp$	..

The 2nd allocation starts at time $t = 19$ and is

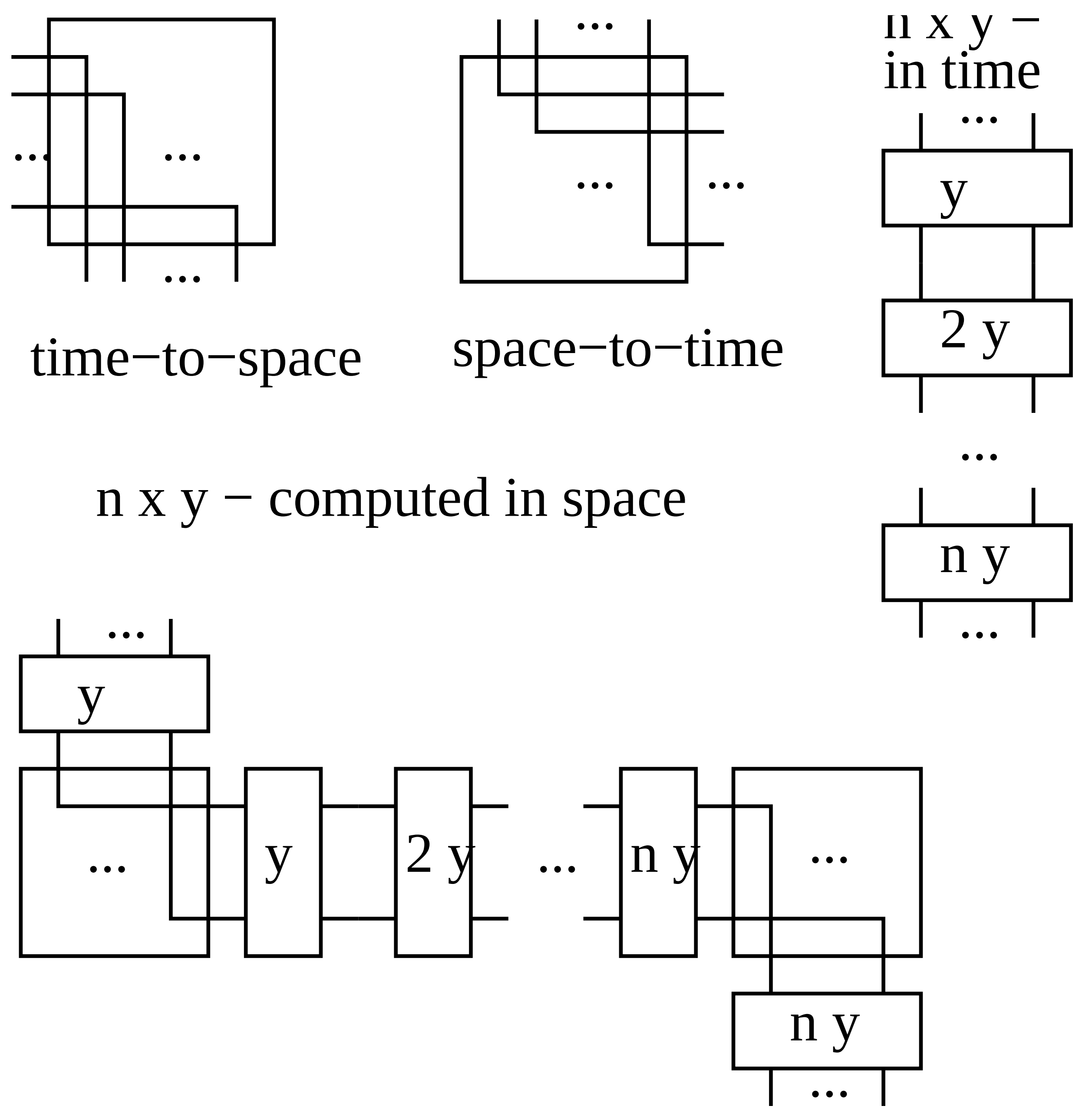
time :	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
	..	n	e	x	t		w	e	e	k	I		w	i	l	l		b	e		o	n		M	o	o	n		..
	..	11	12	13	14	15	16	17	18	$\perp$	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	10	..

The 3rd allocation starts at time $t = 21$ and is

time :	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
	..							b	e	e	e	e	I	i	k	l	l	M	o	o	o	n	n	n	t	w	w	x	..
	..	34	16	27	26	32	35	17	12	36	20	23	10	24	$\perp$	25	11	28	30	29	31	13	14	18	15	22	19	33	..

Space-time converters; computation in space

space-to-time
and
time-to-space
converters
may be used to
change
computation in time
into a
computation in space
paradigm



Contents:

Srv-programs for perfect numbers

A specification for perfect numbers:

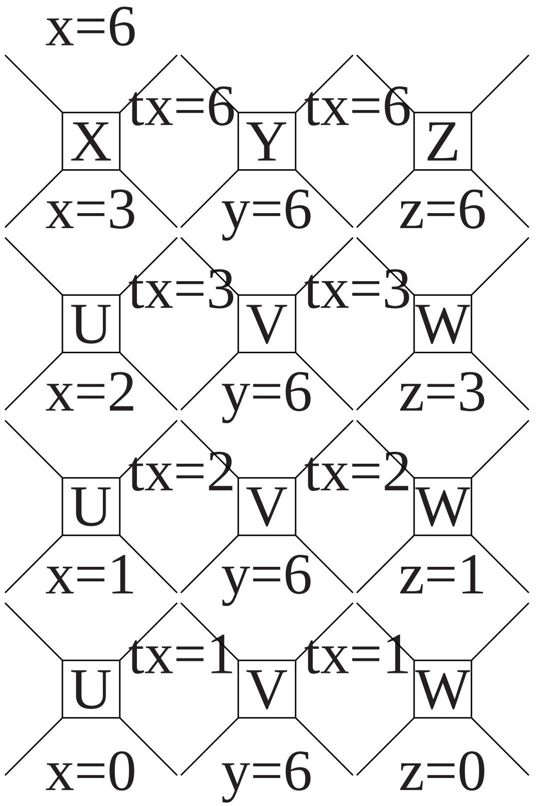
3 components C_x, C_y, C_z where:

- C_x: read n from north and write $n \frown \lfloor n/2 \rfloor \frown (\lfloor n/2 \rfloor - 1) \frown \ldots \frown 2 \frown 1$ on east;
- C_y: read $n \frown \lfloor n/2 \rfloor \frown (\lfloor n/2 \rfloor - 1) \frown \ldots \frown 2 \frown 1$ from west and write $n \frown \phi(\lfloor n/2 \rfloor) \frown \ldots \frown \phi(2) \frown \phi(1)$ on east [$\phi(k) =$ "if k divides n then k else 0"];
- C_z: read $n \frown \phi(\lfloor n/2 \rfloor) \frown \ldots \frown \phi(2) \frown \phi(1)$ from west and subtract from the first the other numbers.

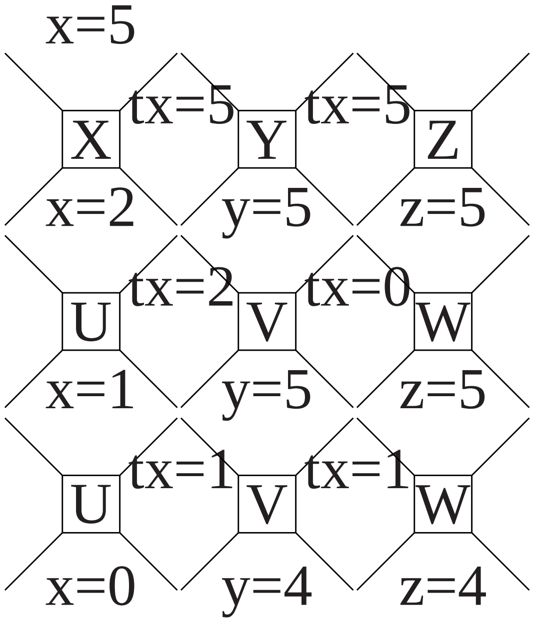
These components are composed *horizontally*. The global input-output specification: *if the input number in C_x is n, then the output number in C_z is 0 iff n is perfect*.

..Srv-programs for perfect numbers

Two scenarios for perfect numbers:



(a)



(b)

Types are denoted as $\langle west|north\rangle \rightarrow \langle east|south\rangle$

Our (s)rv-scenarios are similar with the tiles of Bruni-Gadducci-Montanari, et.al.

..Srv-programs for perfect numbers

The 1st AGAPIA program Perfect1 (construction by rows):

```
(X # Y # Z) % while_t(x>0){U # V # W}
```

Its type is **Perfect1** : $\langle nil | sn; nil; nil\rangle \to \langle nil | sn; sn; sn\rangle$.

Modules:

```
X:: module{listen nil;}{read x:sn;}
        {tx:tn; tx=x; x=x/2;}{speak tx;}{write x;}
Y:: module{listen tx:tn;}{read nil;}
        {y:sn; y=tx;}{speak tx;}{write y;}
Z:: module{listen tx:tn;}{read nil;}
        {z:sn; z=tx;}{speak nil;}{write z;}
U:: module{listen nil;}{read x:sn;}
        {tx:tn; tx=x; x=x-1;}{speak tx;}{write x;}
V:: module{listen tx:tn;}{read y:sn;}
        {if(y%tx != 0) tx=0;}{speak tx;}{write y;}
W:: module{listen tx:tn;}{read z:sn}
        {z=z-tx;}{speak nil;}{write z;}
```

..Srv-programs for perfect numbers

The 2nd AGAPIA program Perfect2 (construction by columns):

```
(X % while_t(x>0){U} % U1)
# (Y % while_t(tx>-1){V} % V1)
# (Z % while_t(tx>-1){W} % W1)
```

Its type is **Perfect2** : $\langle nil|sn;nil;nil\rangle \rightarrow \langle nil|nil;nil;sn\rangle$.

New modules:

```
U1:: module{listen nil;}{read x:sn;}
         {tx:tn; tx=-1;}{speak tx;}{write nil;}
V1:: module{listen tx:tn;}{read y:sn;}
         {null;}{speak tx;}{write nil;}
W1:: module{listen tx:tn;}{read z:sn}
         {null;}{speak nil;}{write z;}
```

Contents:

- Generalities
- A glimpse on AGAPIA programming
- *Finite interactive systems* $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions

Grids (or planar words)

A *grid* (or *planar word*) is

- a rectangular *two-dimensional area*
- filled in with *letters* from a given alphabet

Example:

```
aabbabb
abbcdbb
bbabbca
ccccaaa
```

(not used here:

```
aabb...
..bc..b
bbabbca
..c...a
```

)

A grid p has a *north* (resp. *south, west, east*) border denoted as

$$n(p) \quad \text{(resp.} \quad s(p), w(p), e(p))$$

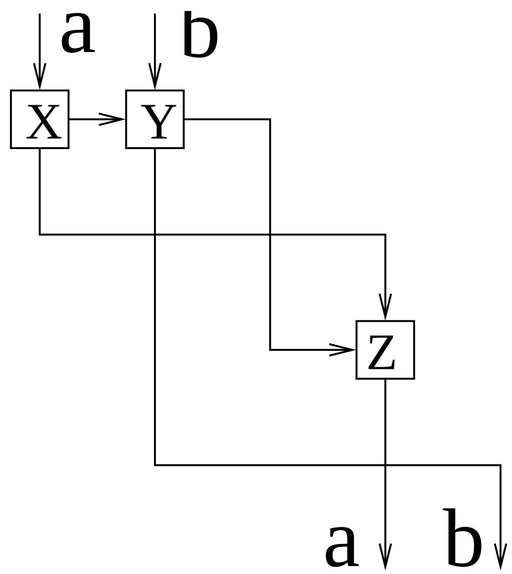
Notice: The requirement to have a rectangular area may be weakened, e.g., one may require to have a connected area, not a rectangular one.

..Grids (or planar words)

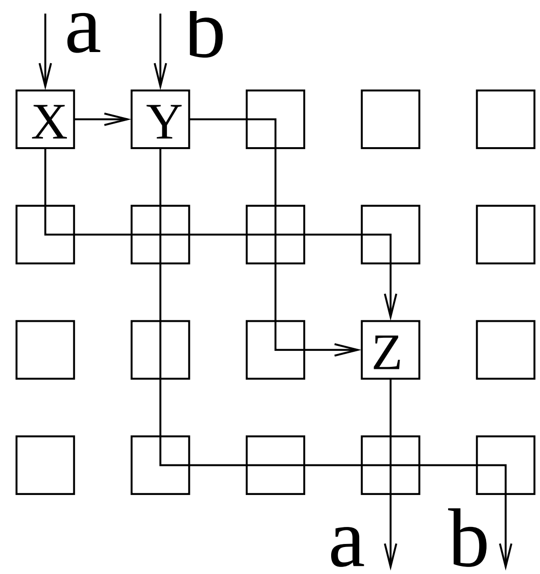
Causality in a grid/scenario:

$$\begin{tikzcd} a \arrow[r] \arrow[d] & b \arrow[r] \arrow[d] & c \arrow[r] \arrow[d] & d \arrow[d] \\ e \arrow[r] & f \arrow[r] & g \arrow[r] & h \end{tikzcd}$$

Action vs. inter-action:



(a)



(b)

- a two-ways interaction [in (a)]
- ... and its grid/scenario representation [in (b)]

The flattening operator

The *flattening operator*

$$\flat \;:\; \mathrm{LangGrids}(V) \quad\to\quad \mathrm{LangWords}(V)$$

maps sets of *grids* to *sets of strings* representing their topological sorting. Example:

—start with $\begin{smallmatrix}\mathtt{abcd}\\ \mathtt{efgh}\end{smallmatrix}$; there is one minimal element `a`; after its deletion we get $\begin{smallmatrix}\mathtt{\ bcd}\\ \mathtt{efgh}\end{smallmatrix}$;

—the minimal elements are `b` and `e`; suppose we choose `b`; what remains is $\begin{smallmatrix}\mathtt{\ \ cd}\\ \mathtt{efgh}\end{smallmatrix}$; and so on;

—finally a usual word, say `abecfgdh`, is obtained.

Actually, $\flat\left(\begin{smallmatrix}\mathtt{abcd}\\ \mathtt{efgh}\end{smallmatrix}\right) =$

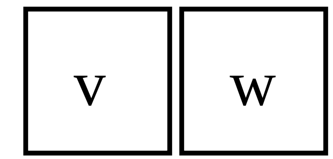
$\{$`abcdefgh`, `abcedfgh`, `abcefdgh`, `abcefgdh`, `abecdfgh`, `abecfdgh`, `abecfgdh`, `abefcdgh`, `abefcgdh`, `aebcdfgh`, `aebcfdgh`, `aebcfgdh`, `aebfcdgh`, `aebfcgdh`$\}$

Composition and identities on grids

- *horizontal composition* $v \triangleright w$

—it is defined only if $e(v) = w(w)$

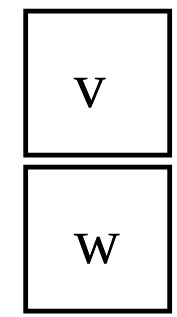
—$v \triangleright w$ is the word obtained putting v on the left of w



- *vertical composition* $v \cdot w$

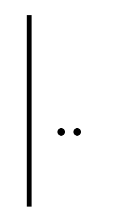
—it is defined only if $s(v) = n(w)$

—$v \cdot w$ is the word obtained putting v on top of w



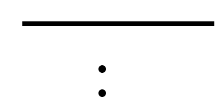
- *vertical identity* ε_k:

—with $w(\varepsilon_k) = e(\varepsilon_k) = 0$ and $n(\varepsilon_k) = s(\varepsilon_k) = k$



- *horizontal identity* λ_k:

—with $w(\lambda_k) = e(\lambda_k) = k$ and $n(\lambda_k) = s(\lambda_k) = 0$



Two-dimensional regular expressions

Signature: two sets of regular algebra operators, sharing the additive part

$$+,\ 0,\ \cdot,\ ^\star,\ \shortmid,\ \triangleright,\ ^\dagger,\ -$$

—$(+,0,\cdot,^\star,\shortmid)$ - a Kleene signature for the vertical dimension
—$(+,0,\triangleright,^\dagger,-)$ - a Kleene signature for the horizontal dimension

Two-dimensional regular expressions (denoted $\mathsf{2RegExp}(V)$):

$$E ::= a(\in V) \mid 0 \mid E+E \mid E\cap E \mid E\cdot E \mid E^\star \mid \shortmid \mid E\triangleright E \mid E^\dagger \mid -$$

Theorem:

> $\mathsf{2RegExp}(V)$ *enriched with letter-to-letter* homomorphisms *are* equivalent *with* FIS's *(finite interactive systems).*

From expressions to sets of grids

Interpretation (from *expressions* to *sets of grids*)

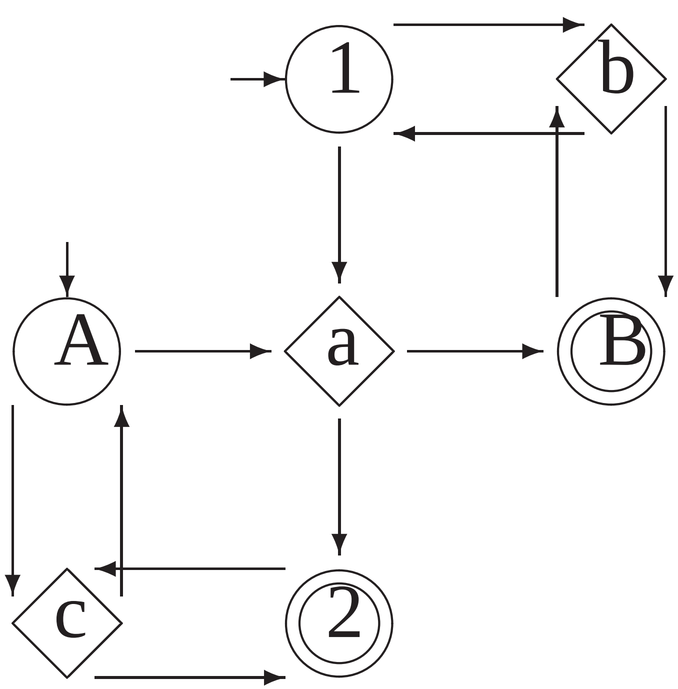
$$|\ \ | : \mathsf{2RegExp}(V) \rightarrow LangGrids(V)$$

- $|a| = \{a\}$; $|0| = \emptyset$; $|E+F| = |E| \cup |F|$
- $|\shortmid| = \{\varepsilon_0, \ldots, \varepsilon_k, \ldots\}$
- $|E \cdot F| = \{v \cdot w : v \in |E| \ \& \ w \in |F|\}$
- $|E^\star| = \{v_1 \cdot \ldots \cdot v_k : k \in \mathbb{N} \ \& \ v_1, \ldots, v_k \in |E|\} \cup |\shortmid|$
- $|-| = \{\lambda_0, \ldots, \lambda_k, \ldots\}$
- $|E \triangleright F| = \{v \triangleright w : v \in |E| \ \& \ w \in |F|\}$
- $|E^\dagger| = \{v_1 \triangleright \ldots \triangleright v_k : k \in \mathbb{N} \ \& \ v_1, \ldots, v_k \in |E|\} \cup |-|$

Finite interactive systems

Example (finite interactive system): A FIS S and its

graphical representation:



“cross” representation:

- $\begin{array}{|c|c|c|}\hline & 1 & \\ \hline A & a & B \\ \hline & 2 & \\ \hline\end{array}$, $\begin{array}{|c|c|c|}\hline & 2 & \\ \hline A & c & A \\ \hline & 2 & \\ \hline\end{array}$, and $\begin{array}{|c|c|c|}\hline & 1 & \\ \hline B & b & B \\ \hline & 1 & \\ \hline\end{array}$
- $A, 1$ are initial
- $B, 2$ final

or *textual representation*

- `a:<A|1>-><B|2>`, etc.

Scenarios, accepting criteria

Example: A successful scenario for recognizing a grid:

- Given a grid $w =$
```
abb
cab
cca
```
, start with initial states/classes on north/west borders;

- parse the grid: $w_0 =$
```
  1 1 1
Aa b b

Ac a b

Ac c a
```
; $w_1 =$
```
  1 1 1
AaBb b
  2
Ac a b

Ac c a
```
; $w_2 =$
```
  1 1 1
AaBbBb
  2 1
Ac a b

Ac c a
```
;

$w_3 =$
```
  1 1 1
AaBbBb
  2 1
AcAa b
  2
Ac c a
```
; ...; $w_9 =$
```
  1 1 1
AaBbBbB
  2 1 1
AcAaBbB
  2 2 1
AcAcAaB
  2 2 2
```

- The grid is recognized if, after parsing, only final states/classes are on south/east borbers

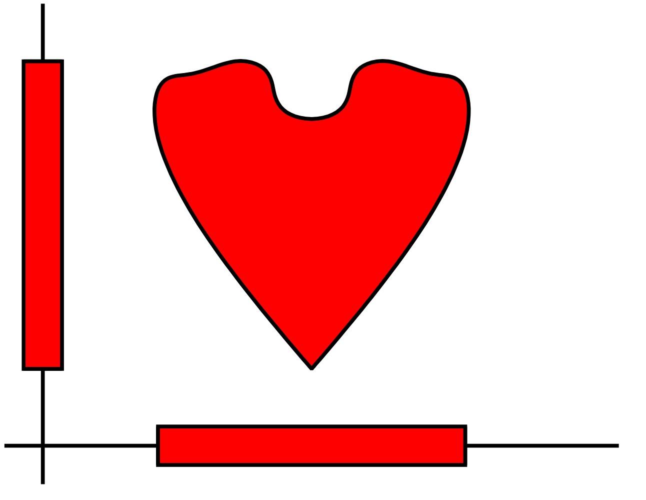
$L(S) = \{$ a's on the diagonal, top-right half of b's, and bottom-left half of c's$\}$.

State projection and class projection

Familiar NFA's (*nondeterministic finite automata*) are obtained *neglecting one dimension*

- *state projection nfa* $\mathsf{state}(S)$
  —obtained neglecting the class transforming part
- *class projection nfa* $\mathsf{class}(S)$
  —obtained neglecting the state transforming part

projections
may lose
information



Projections, example

For the above this FIS, the grid language may be obtained from the languages of its projection nfa as follows:

$$L(S) = L(\mathsf{state}(S))^{\dagger} \cap L(\mathsf{class}(S))^{\star}$$

Actually,

$$L(S) = (b^{\star} \cdot a \cdot c^{\star})^{\dagger} \cap (c^{\dagger} \triangleright a \triangleright b^{\dagger})^{\star}$$

Fact:

1. *Such a decomposition holds for all* FIS*'s with distinct labels on their transitions.*

2. *By enriching regular expressions with homomorphisms, one gets a representation theorem for all* FIS*'s.*

FIS **vs. 2-dimensional languages**

Theorem:

The following are equivalent for a 2-dimensional language L (called recognizable two-dimensional language*; their class is denoted by REC):*

1. *L is recognized by a* on-line tessellation automaton*;*
2. *L is defined by a* tile systems *(i.e., local lattice languages closed to letter-to-letter homomorphisms);*
3. *L is defined by an* existential monadic second order formula*; etc.*

See: Giammarresi-Restivo (1997), or Lindgren-Moore-Nordahl (1998);
a useful web-page is B.Borchert's page at

`http://math.uni-heidelberg.de/logic/bb/2dpapers.html`

Notice: 2-dimensional languages are also known as "picture" languages.

..FIS vs. 2-dimensional languages

Theorem:

> *A set of grids is recognizable by a finite interactive system iff it is recognizable by a tiling system.*

This shows that the class of FIS recognizable grid languages coincides with REC, so we may *inherit many results known for 2-dimensional languages*. Two important ones are:

Corollaries:

1. *Context-sensitive word languages coincide with the projection on the 1st row of the* FIS *recognizable grid languages.*
2. *The emptiness problem for* FIS*'s is undecidable.*

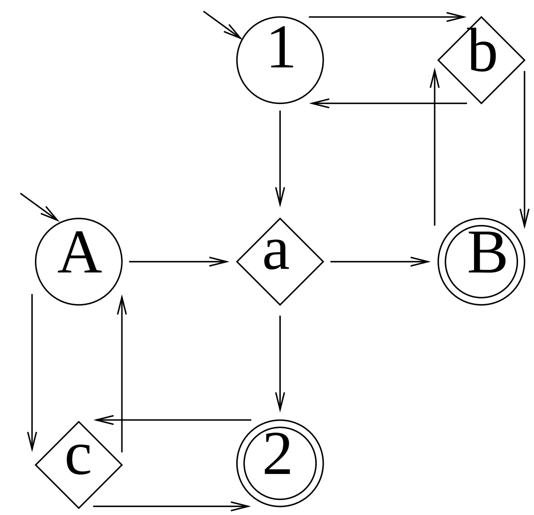
Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- *Rv-programs* $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions

Finite interactive systems

Finite interactive systems:

- *states*: 1,2 [1-initial; 2-final]
- *classes*: A, B [A-initial; B-final]
- *transitions*: a, b, c



Parsing procedure (to recognize grids):

A parssing for
```
abb
cab
cca
```
:

```
 1 1 1    1 1 1    1 1 1    1 1 1   ...  1 1 1
Aa b b   AaBb b   AaBbBb   AaBbBb       AaBbBbB
          2        2 1      2 1          2 1 1
Ac a b   Ac a b   Ac a b   AcAa b       AcAaBbB
                            2            2 2 1
Ac c a   Ac c a   Ac c a   Ac c a       AcAcAaB
                                         2 2 2
```

RV-programs

RV-systems:

- An *rv-system* (*interactive system with registers and voices*) is a FIS enriched with:
  - *registers* associated to its *states* and *voices* associated to its *classes*;
  - appropriate *spatio-temporal transformations for actions*.

  We study rv-systems specified by *rv-programs* (see below)

- A *computation* is described by a scenario like in a FIS, but with concrete data around each action.

..RV-programs

An rv-program (for perfect numbers):

in: A,1; out: D,2

X::

(A,1)	x : sInt

	tx : tInt;
	tx = x;
	x = x/2;
	goto [B,3];

U::

(A,3)	x : sInt

	tx : tInt;
	tx = x;
	x = x − 1;
	if (x > 0) goto [B,3]
	else goto [B,2];

Y::

(B,1)	y : sInt
tx : tInt	y = tx; goto [C,2];

V::

(B,2)	y : sInt
tx : tInt	if(y%tx != 0) tx = 0; goto [C,2];

Z::

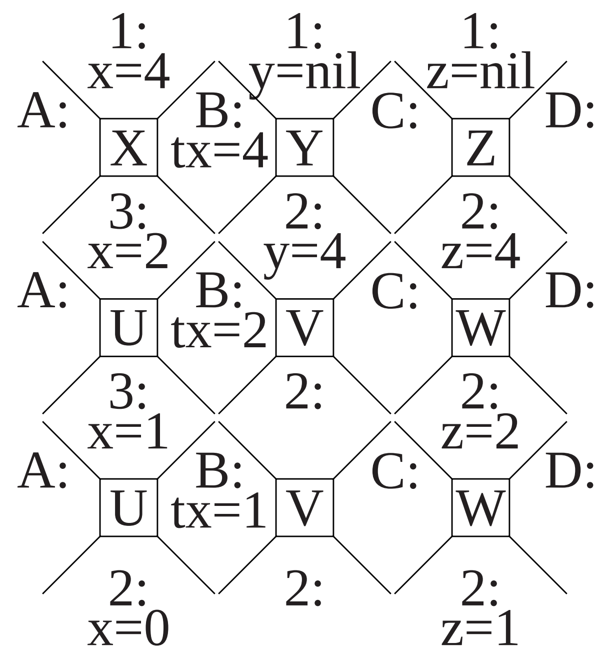
(C,1)	z : sInt
tx : tInt	z = tx; goto [D,2];

W::

(C,2)	z : sInt
tx : tInt	z = z − tx; goto [D,2];

..RV-programs

Scenario:



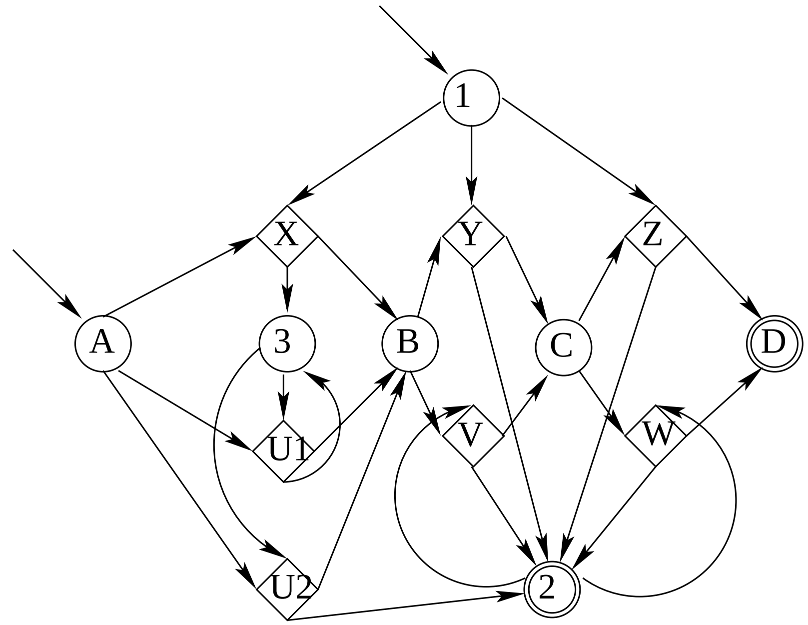
Operational semantics:

- defined in terms of scenarious

Relational semantics:

- input-output relation generated by all possible scenarious

Associating a FIS:



or, equivalently,

$$
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \hline A & X & B \\ \hline \multicolumn{1}{c|}{} & 3 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},\quad
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \hline B & Y & C \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},\quad
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \hline C & Z & D \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},
$$

$$
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 3 & \multicolumn{1}{c}{} \\ \hline A & U1 & B \\ \hline \multicolumn{1}{c|}{} & 3 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},\quad
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 3 & \multicolumn{1}{c}{} \\ \hline A & U2 & B \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},\quad
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \hline B & V & C \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array},\quad
\begin{array}{|c|c|c|}\cline{2-2} \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \hline C & W & D \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2}\end{array}
$$

..RV-programs

... and a grid language:

```
X  Y  Z
U1 V  W
.  .  .
U1 V  W
U2 V  W
```

These grids may be decomposed:

- by *rows*
- by *columns*, then each column by rows

leading to various equivalent structured programming variants for this program.

Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- *Structured rv-programs* $\leftarrow$ *[while programs]*
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions

Structured rv-programs

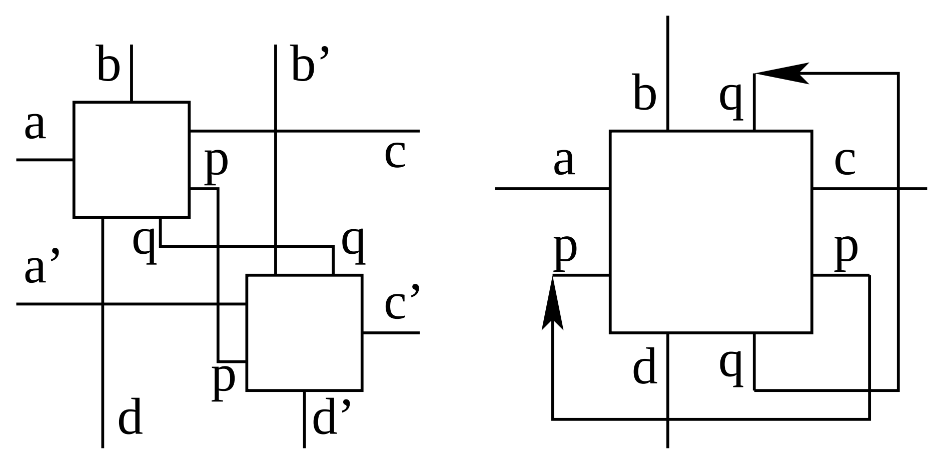
Syntax:

$$X ::= module\{listen\ t\_vars;\}\{read\ s\_vars;\}$$
$$\{code;\}\{speak\ t\_vars;\}\{write\ s\_vars;\}$$

$$P ::= X \mid if(C)then\{P\}else\{P\} \mid P\%P \mid P\#P \mid P\$P$$
$$\mid while\_t(C)\{P\} \mid while\_s(C)\{P\} \mid while\_st(C)\{P\}$$

More general operators: Composition and iterated composition statements are instances of a unique, more general, but less "structured" form (only the *tv/sv parts* of the connecting interfaces are *to be matched*):

- $P1\ comp\{tv\}\{sv\}\ P2$
- $while\{tv\}\{sv\}\{C\}\{P\}$



AGAPIA

Basic characteristics of AGAPIA

- *space-time invariant*
- *high-level temporal data* structures
- *computation extends* both in *time* and *space*
- a *structural, compositional model*
- simple *operational semantics* (using *scenarios*)
- simple *relational semantics*

AGAPIA v0.1: Syntax

Syntax of AGAPIA v0.1:

Interfaces

$SST ::= nil \mid sn \mid sb$
$\quad \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$
$ST ::= (SST)$
$\quad \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^*$
$STT ::= nil \mid tn \mid tb$
$\quad \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^*$
$TT ::= (STT)$
$\quad \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^*$

Expressions

$V ::= x : ST \mid x : TT$
$\quad \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k]$
$E ::= n \mid V \mid E + E \mid E * E \mid E - E \mid E/E$
$B ::= b \mid V \mid B\&\&B \mid B||B \mid !B \mid E < E$

Programs

$W ::= null \mid new\ x : SST \mid new\ x : STT$
$\quad \mid x := E \mid if(B)\{W\}else\{W\}$
$\quad \mid W; W \mid while(B)\{W\}$
$M ::= module\{listen\ x : STT\}\{read\ x : SST\}$
$\quad \{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}$
$P ::= null \mid M \mid if(B)\{P\}else\{P\}$
$\quad \mid P\%P \mid P\#P \mid P\$P$
$\quad \mid while\_t(B)\{P\} \mid while\_s(B)\{P\}$
$\quad \mid while\_st(B)\{P\}$

Example: Termination detection

Example: A program for distributed termination detection

```
P= I1# for_s(tid=0;tid<tm;tid++){I2}#
   $ while_st(!(token.col==white && token.pos==0)){
      for_s(tid=0;tid<tm;tid++){R}}
```

where:

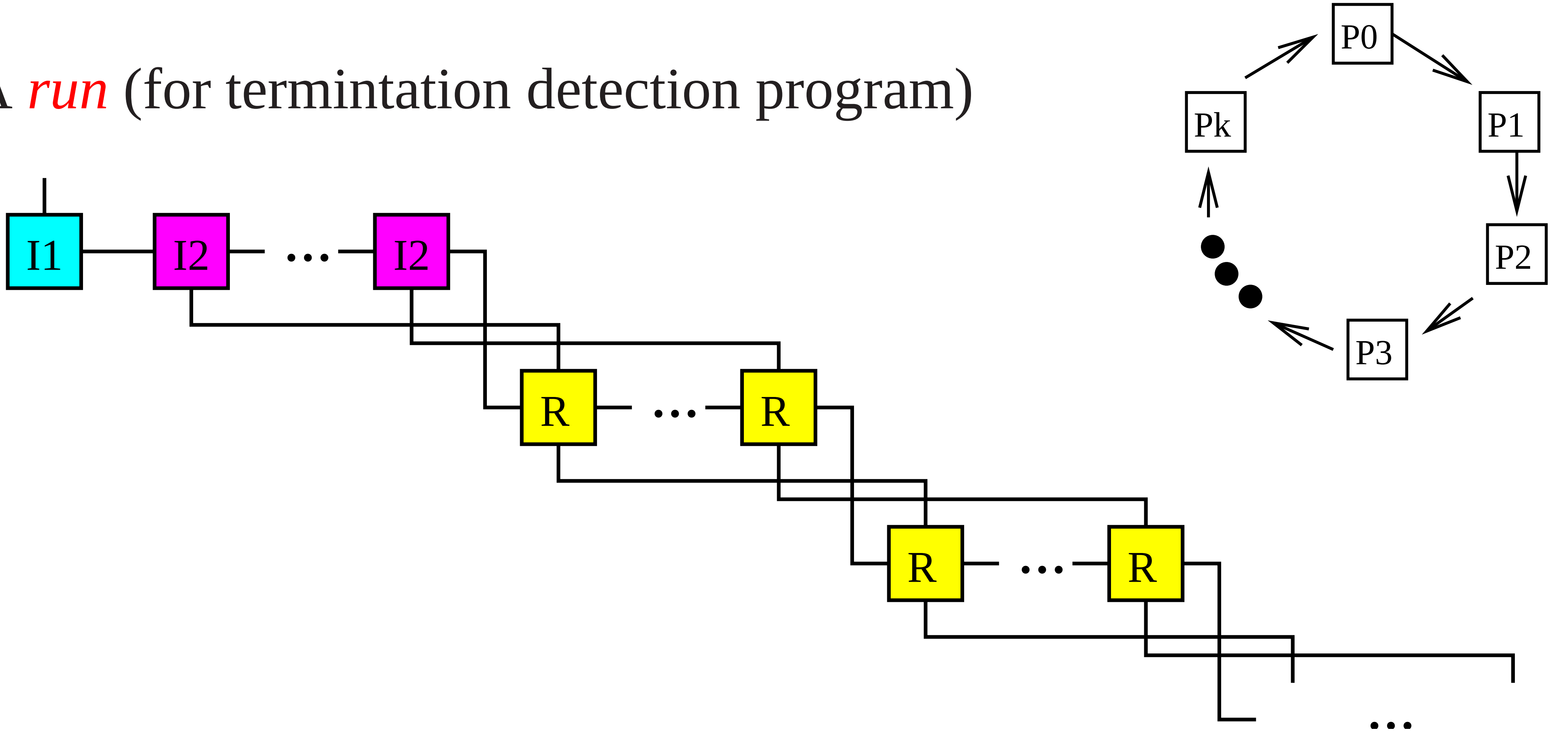
```
I1= module{listen nil}{read m}{
    tm=m; token.col=black; token.pos=0;
    }{speak tm,tid,msg[ ],token(col,pos)}{write nil}

I2= module{listen tm,tid,msg[ ],token(col,pos)}
    {read nil}{
    id=tid; c=white; active=true; msg[id]=null;
    }{speak tm,tid,msg[ ],token(col,pos)}
    {write id,c,active}
```

..Example: Termination detection

```
R= module{listen tm,tid,msg[ ],token(col,pos)}
   {read id,c,active}{
   if(msg[id]!=emptyset){ //take my jobs
      msg[id]=emptyset;
      active=true;}
   if(active){ //execute code, send jobs, update color
      delay(random_time);
      r=random(tm-1);
      for(i=0;i<r;i++){ k=random(tm-1);
        if(k!=id){msg[k]=msg[k]∪{id}};
        if(k<id){c=black};}
      active=random(true,false);}
   if(!active && token.pos==id){ //termination
      if(id==0)token.col=white;
      if(id!=0 && c==black){token.col=black;c=white};
      token.pos=token.pos+1[mod tm];}
   }{speak tm,tid,msg[ ],token(col,pos)}
   {write id,c,active}
```

..Example: Termination detection

A *run* (for termintation detection program)




```
I1# for_s(tid=0;tid<tm;tid++){I2}#
$ while_st(!(token.col==white && token.pos==0)){
   for_s(tid=0;tid<tm;tid++){R}}
```

AGAPIA v0.1: Syntax

Syntax of AGAPIA v0.1:

Interface types

We use two special separators “,” and “;”

On spatial interfaces:

- “,” separates the types used in *a process*
- “;” separates the types used in *different processes*

On temporal interfaces:

- “,” separates the types used within *a transaction*
- “;” separates the types used in *different transactions*

Interface types

Simple spatial types are defined by:

$$SST ::= nil \mid sn \mid sb \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$$

(“,” - associative with “nil” neutral element; “$\cup$” - associative)

Example:

$$((((sn)^*)^*, sb, (sn, sb, sn)^*)^*, (sb \cup sn))$$

represents the folowing data structure (for *a process*)

```
x:  struc1[], where
    struc1 = ( a:  Int[][],
               b:  Bool,
               c:  struc2[], where
                   struc2 = (p:Int, q:Bool, r:Int)
             ),
y:  Bool or Int
```

Simple temporal types — similar

Interface types

Spatial types are defined by::

$$ST ::= nil \mid (SST) \mid (ST \cup ST) \mid (ST;ST) \mid (ST;)^*$$

(“;” - associative with “nil” neutral element; “$\cup$” - associative)

Example:

$$((sn)^*)^*;nil;sb;((sn)^*;)^*$$

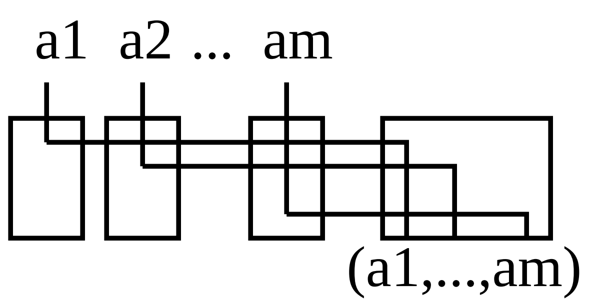
represents *a collection of processes* (A,B,C,D), where

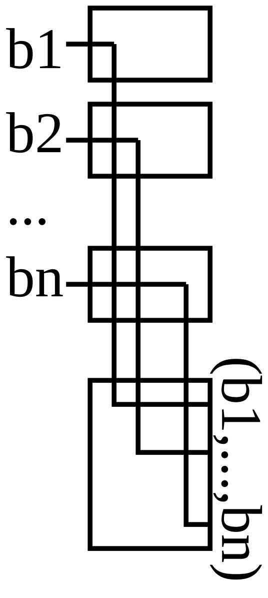
- A is *a process* using an array of arrays of integers
- B is *a process* with no starting spatial data
- C is *a process* using a boolean variable
- D is *an array of processes*, each process using an array of integers

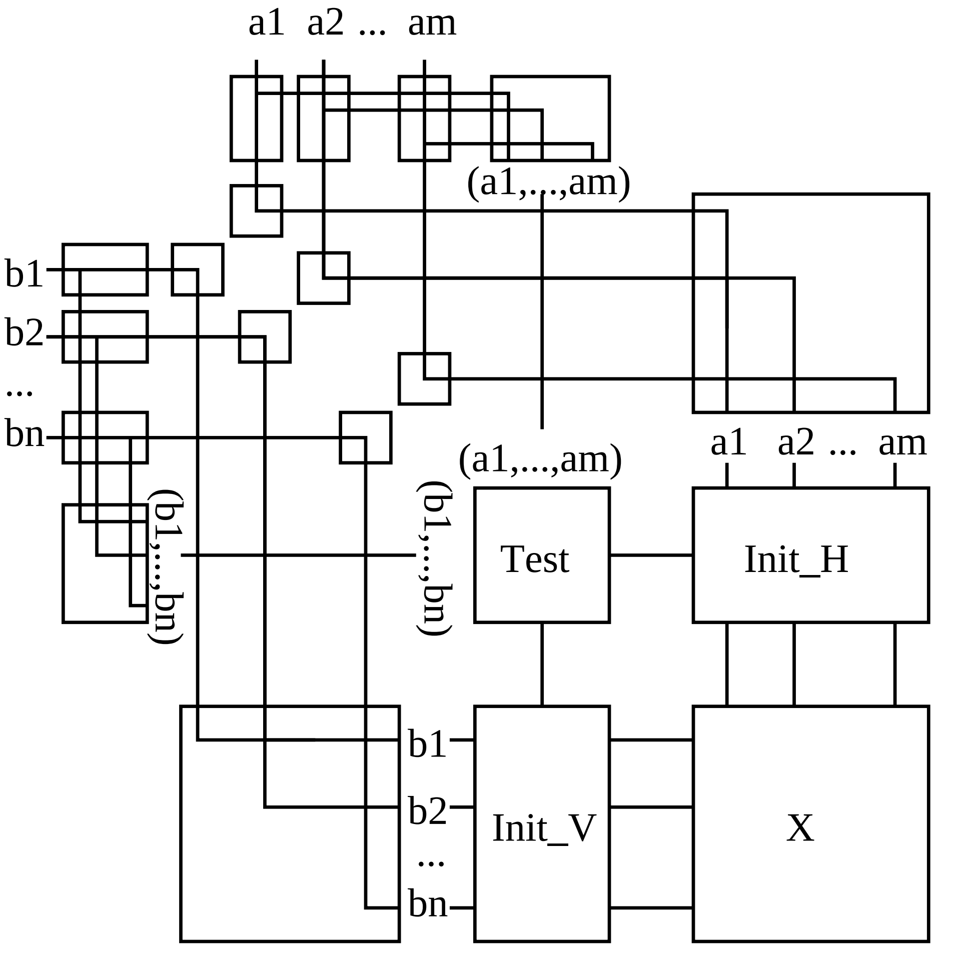
Temporal types — similar

Reshaping types

- interface types may be changed using special morphisms
- examples $(sn;)^* \mapsto (sn)^*$ and $(tn;)^* \mapsto (tn)^*$ (left)







AGAPIA v0.1: Syntax

Expressions

Variables

$$V ::= x : ST \mid x : TT \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k]$$

Arithmetic expressions

$$E ::= n \mid V \mid E + E \mid E * E \mid E - E \mid E/E$$

Boolean expressions

$$B ::= b \mid V \mid B\&\&B \mid B||B \mid !B \mid E < E$$

..AGAPIA v0.1: Syntax

Programs

Simple while programs

$$\begin{aligned} W ::= \ & null \mid new\ x : SST \mid new\ x : STT \\ & \mid x := E \mid if(B)\{W\}else\{W\} \\ & \mid W;W \mid while(B)\{W\} \end{aligned}$$

Modules

$$\begin{aligned} M ::= \ & module\{listen\ x : STT\}\{read\ x : SST\} \\ & \qquad \{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\} \end{aligned}$$

Agapia v0.1 programs

$$\begin{aligned} P ::= \ & null \mid M \mid if(B)\{P\}else\{P\} \\ & \mid P\%P \mid P\#P \mid P\$P \\ & \mid while\_t(B)\{P\} \mid while\_s(B)\{P\} \mid while\_st(B)\{P\} \end{aligned}$$

..AGAPIA v0.1: Syntax

Temporal (or vertical) composition and while

- denoted “%” and *while_t*
- composition of modules/programs via spatial interfaces (“usual” composition)

Spatial (or horizontal) composition and while

- denoted “#” and *while_s*
- composition of modules/programs via temporal interfaces

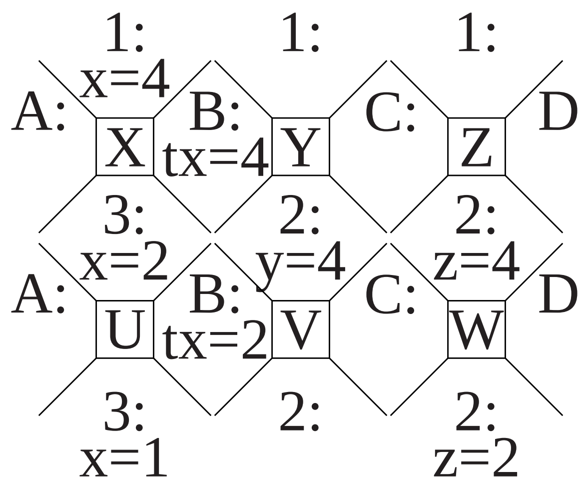
Spatio-temporal (or diagonal) composition and while

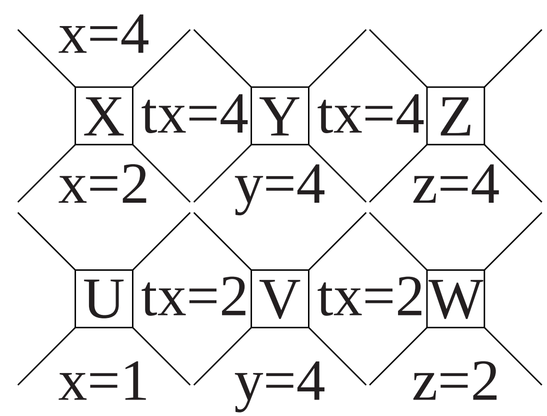
- denoted “$” and *while_st*
- composition of modules/programs via both spatial and temporal interfaces

Scenarios

Scenarios:

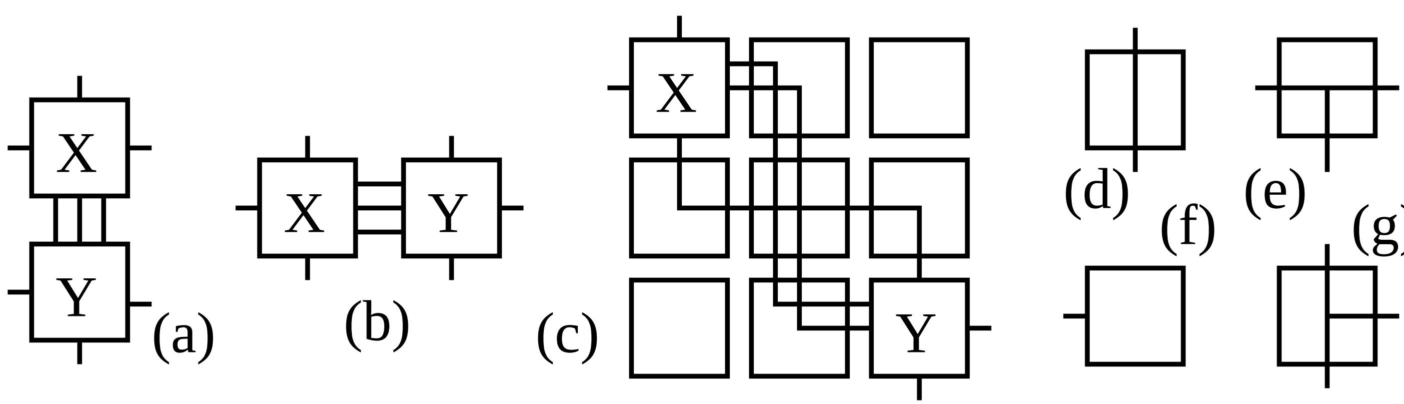
```
 1  1  1
AaBbBbB
 2  1  1
AcAaBbB
 2  2  1
AcAcAaB
 2  2  2
```





(1) FIS's scenario (2) rv-scenario (3) srv-scenario

Srv-scenario operations:

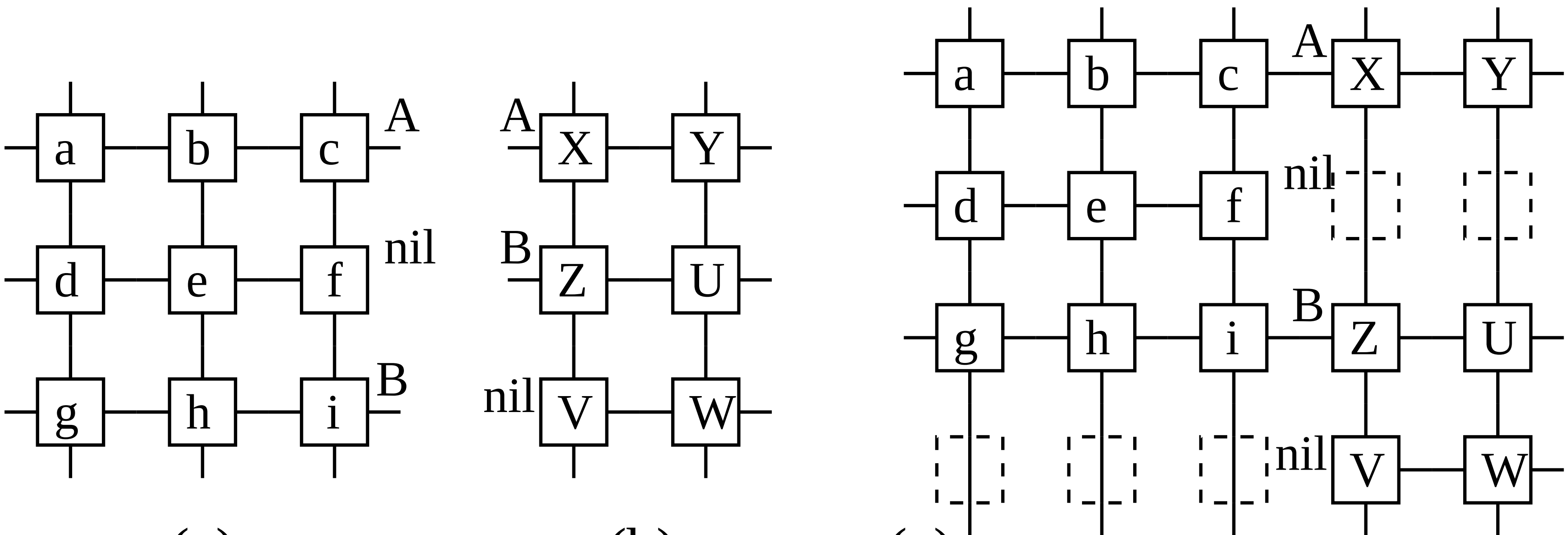


(4)

..Operations on srv-scenarios

..Srv-scenario operations:

- Details for horizontal composition



(a) (b) (c)

- Similar procedures applies to the vertical and the diagonal srv-scenario compositions

Typing expressions

Typing declarations Start from $x : ST$ or $x : TT$ and use

(k) - the k-th element of an alternative choice (separated by "$\cup$")

.k - the k-th element of a structure (separated by ",")

.[k] - the k-th element of an array (defined by $(\ldots)^*$)

@k - the k-th process/transaction (separated by ";")

@[k] - the k-th process/transaction of an array of processes/transactions (defined by "$(\ldots;)^*$")

Examples:

- $w : ((((sn)^*)^*, sb, (sn, sb, sn)^*)^*, (sb \cup sn))$
  $w.1.[i].3.[j].2$ (the 2nd sb)
- $w : ((sn)^*)^*; nil; sb; ((sn \cup sb)^*;)^*$
  $w@3@[i].[j](1)$ (the last sn)

Typing programs

The typing morphism - defined by a mapping

$$\sigma : P \mapsto (st_{\sigma(P)}, \langle w_{\sigma(P)} | n_{\sigma(P)} \rangle \to \langle e_{\sigma(P)} | s_{\sigma(P)} \rangle)$$

where:

- $st \in \{\mathtt{ok}, \mathtt{war0}, \mathtt{war1}, \mathtt{err}\}$ says the program is:
  - *ok* - well-typed;
  - $war0, war1$ - partially well-typed with two levels of warnings;
  - *err* - wrongly typed
- On each *west, north, east*, or *south* interface, the type $w_{\sigma(P)}$, $n_{\sigma(P)}$, $e_{\sigma(P)}$, or $s_{\sigma(P)}$ consists of *a set of variables* with *their types*

Typing programs

Type matching on an interface:

1. Check if *the same* set of *variables* is used;
2. For each variable, its status flag is:
   - *ok* - if their types in these interfaces are *equal* and *singleton*;
   - *war*0 - if their types are *equal*, but *not a singleton*;
   - *war*1 - if their types are *not equal*, but have a *nonempty intersection*;
   - *err* - if their types have an *empty intersection*;
3. Finally, the overall status flag is the *minimum* of the status flags for each variable in the interface set.

..Typing programs

Typing simple while programs and modules:

Simple while programs:

- usual typing, extended with $\{ok, war0, war1, err\}$ flag

Modules:

- take the type of the body program and export on the interfaces *only the variables* occurring in the `listen/speak` and `read/write` statements with their associated types.

..Typing programs

Typing structured rv-programs: On programs, the typing morphism is inductively defined by:

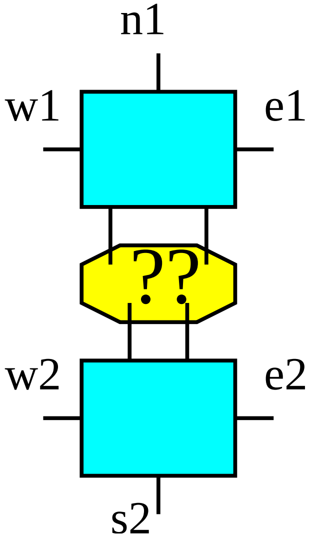
Vertical composition:

$$\sigma(S1\%S2) = (st, \langle w_{\sigma(S1)}; w_{\sigma(S2)} | n_{\sigma(S1)} \rangle \rightarrow \langle e_{\sigma(S1)}; e_{\sigma(S2)} | s_{\sigma(S2)} \rangle), \text{ where}$$

$$st = \begin{cases} min(\texttt{ok}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} = n_{\sigma(S2)} = \text{singleton} \\ min(\texttt{war0}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} = n_{\sigma(S2)} = \neg\text{singleton} \\ min(\texttt{war1}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} \cap n_{\sigma(S2)} \neq \emptyset \\ \texttt{err} & \text{if } s_{\sigma(S1)} \cap n_{\sigma(S2)} = \emptyset \end{cases}$$

Horizontal composition: - similar

Diagonal composition: - similar

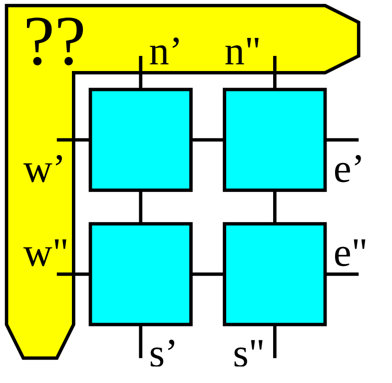


..Typing programs

If:

$\sigma(if(B)\{S1\}else\{S2\}) = (st, \langle w_{\sigma(S1)} \cup w_{\sigma(S2)} | n_{\sigma(S1)} \cup n_{\sigma(S2)} \rangle \rightarrow \langle e_{\sigma(S1)} \cup e_{\sigma(S2)} | s_{\sigma(S1)} \cup s_{\sigma(S2)} \rangle)$ where

$$st = \begin{cases} min(\texttt{ok}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{—if } \sigma(B) \subseteq w_{\sigma(S1)} \cup n_{\sigma(S1)} \cup w_{\sigma(S2)} \cup n_{\sigma(S2)} = \text{singleton} \\ min(\texttt{war0}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{—if } \sigma(B) \subseteq w_{\sigma(S1)} \cup n_{\sigma(S1)} \cup w_{\sigma(S2)} \cup n_{\sigma(S2)} = \neg\text{singleton} \\ min(\texttt{war1}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{—if } \sigma(B) \cap (w_{\sigma(S1)} \cup n_{\sigma(S1)} \cup w_{\sigma(S2)} \cup n_{\sigma(S2)}) \neq \emptyset \\ \texttt{err} \\ \qquad \text{—if } \sigma(B) \cap (w_{\sigma(S1)} \cup n_{\sigma(S1)} \cup w_{\sigma(S2)} \cup n_{\sigma(S2)}) = \emptyset \end{cases}$$



..Typing programs

Temporal while:

$\sigma(while\_t(B)\{S\}) \;=\; (st, \langle (w_{\sigma(S)};)^*|n_{\sigma(S)} \cup s_{\sigma(S)}\rangle \;\to\; \langle (e_{\sigma(S)};)^*|n_{\sigma(S)} \cup s_{\sigma(S)}\rangle)$ where denoting

$$
\begin{array}{ll}
P1 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \text{singleton}, & Q1 := s_{\sigma(S)} = n_{\sigma(S)} = \text{singleton}, \\
P2 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \neg\text{singleton}, & Q2 := s_{\sigma(S)} = n_{\sigma(S)} = \neg\text{singleton}, \\
P3 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) \neq \emptyset, & Q3 := s_{\sigma(S)} \cap n_{\sigma(S)} \neq \emptyset, \\
P4 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) = \emptyset, & Q4 := s_{\sigma(S)} \cap n_{\sigma(S)} = \emptyset
\end{array}
$$

we have

$$
st = \begin{cases}
min(\mathtt{ok}, st_B, st_{\sigma(S)}) & \text{if } P1 \wedge Q1 \\
min(\mathtt{war0}, st_B, st_{\sigma(S)}) & \text{if } P2 \wedge (Q1 \vee Q2) \vee (P1 \vee P2) \wedge Q2 \\
min(\mathtt{war1}, st_B, st_{\sigma(S)}) & \text{if } P3 \wedge (Q1 \vee Q2 \vee Q3) \vee (P1 \vee P2 \vee P3) \wedge Q3 \\
\mathtt{err} & \text{if } P4 \vee Q4
\end{cases}
$$

..Typing programs

Spatial while: $\sigma(while\_s(B)\{S\})$

- is similar to the temporal while

Spatio-temporal while: $\sigma(while\_st(B)\{S\})$

- similar to the temporal while
- ...but slightly more complicate as 3 pairs of interfaces are to be compared:
  - first, σ_B vs. $w_{\sigma(S)} \cup n_{\sigma(S)}$;
  - then, $n_{\sigma(S)}$ vs. $s_{\sigma(S)}$;
  - and, finally, $w_{\sigma(S)}$ vs. $e_{\sigma(S)}$

Example

Example (termination detection) Denote

$a = (tm, tid, msg[\,], token)$, $b = (id, c, active)$, $c = (m)$. Then:

Init:

$$\texttt{I1} \mapsto (\texttt{ok}, \langle nil | c\rangle \longrightarrow \langle a | nil\rangle)$$

$$\texttt{I2} \mapsto (\texttt{ok}, \langle a | nil\rangle \longrightarrow \langle a | b\rangle)$$

$$\texttt{for\_s}(\,)\{\texttt{I2}\} \mapsto (\texttt{ok}, \langle a | nil\rangle \longrightarrow \langle a | (b;)^*\rangle)$$

$$I1\#\texttt{for\_s}(\,)\{\texttt{I2}\} \mapsto (\texttt{ok}, \langle nil | c\rangle \longrightarrow \langle a | (b;)^*\rangle)$$

Repeat:

$$\texttt{R} \mapsto (\texttt{ok}, \langle a | b\rangle \longrightarrow \langle a | b\rangle)$$

$$\texttt{for\_s}(\,)\{\texttt{R}\} \mapsto (\texttt{ok}, \langle a | (b;)^*\rangle \longrightarrow \langle a | (b;)^*\rangle)$$

$$\texttt{while\_st}\{\texttt{for\_s}(\,)\{\texttt{R}\}\} \mapsto (\texttt{war0}, \langle a | (b;)^*\rangle \longrightarrow \langle a | (b;)^*\rangle)$$

Full program:

$$\texttt{P} \mapsto (\texttt{war0}, \langle nil | c\rangle \longrightarrow \langle a | (b;)^*\rangle)$$

Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- *Compiling srv-programs*
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions

Compiling srv-programs

Implementation: Currently, *we have*

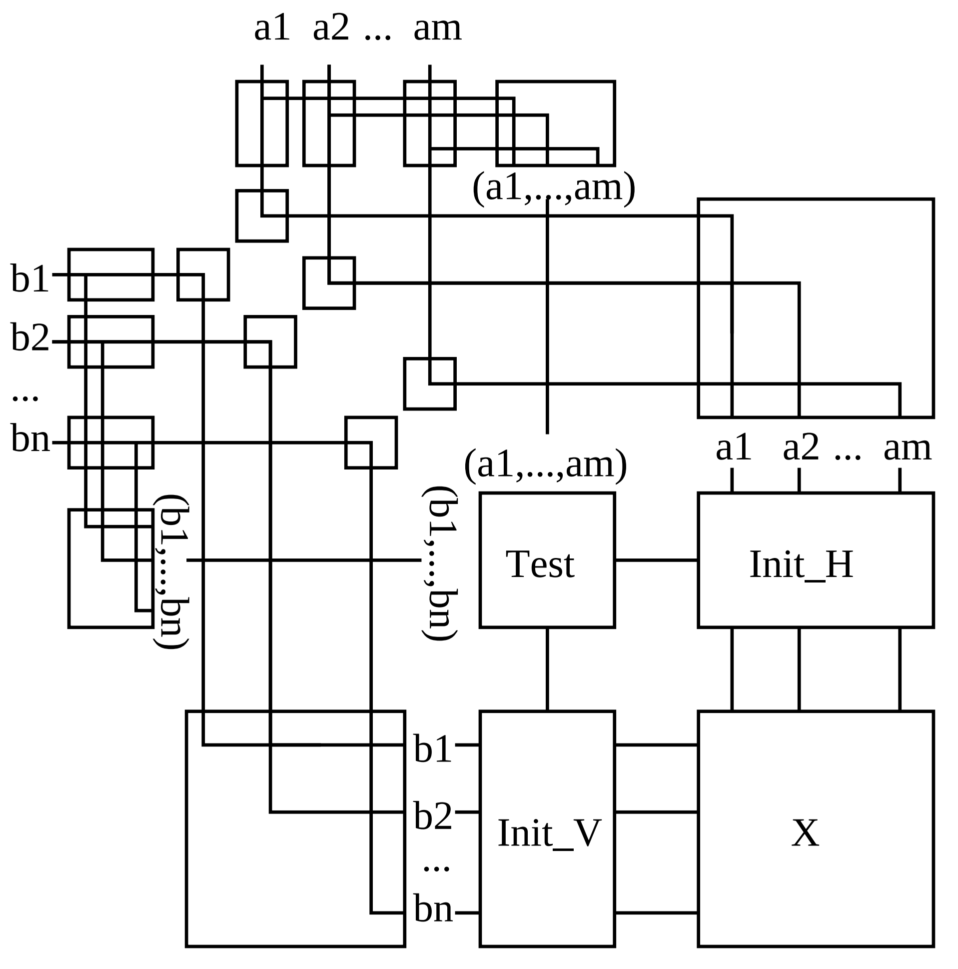
- a simulator for running rv-programs
- a translation from srv- to rv-programs and o proof of its correctness
- a mechanical procedure based on the above translation

Currently, we *do not have*

- an implementation of the translation
- a study of the blow-up induced by the translation
- optimization procedures

..Compiling srv-programs

Example: The translation of *if* is based on the following component



whose implementation as a rv-program is rather tedious.

..Compiling srv-programs

A much more challenging task:

- extend assembly language like MIPS with interactive features (voices)
- design interactive processors
- use such a setting as the target for compiling high-level interactive programming languages including features from AGAPIA

Intermediary step: Add srv-programming features to certain mature programming languages as Eifel, Real Time Java, etc.

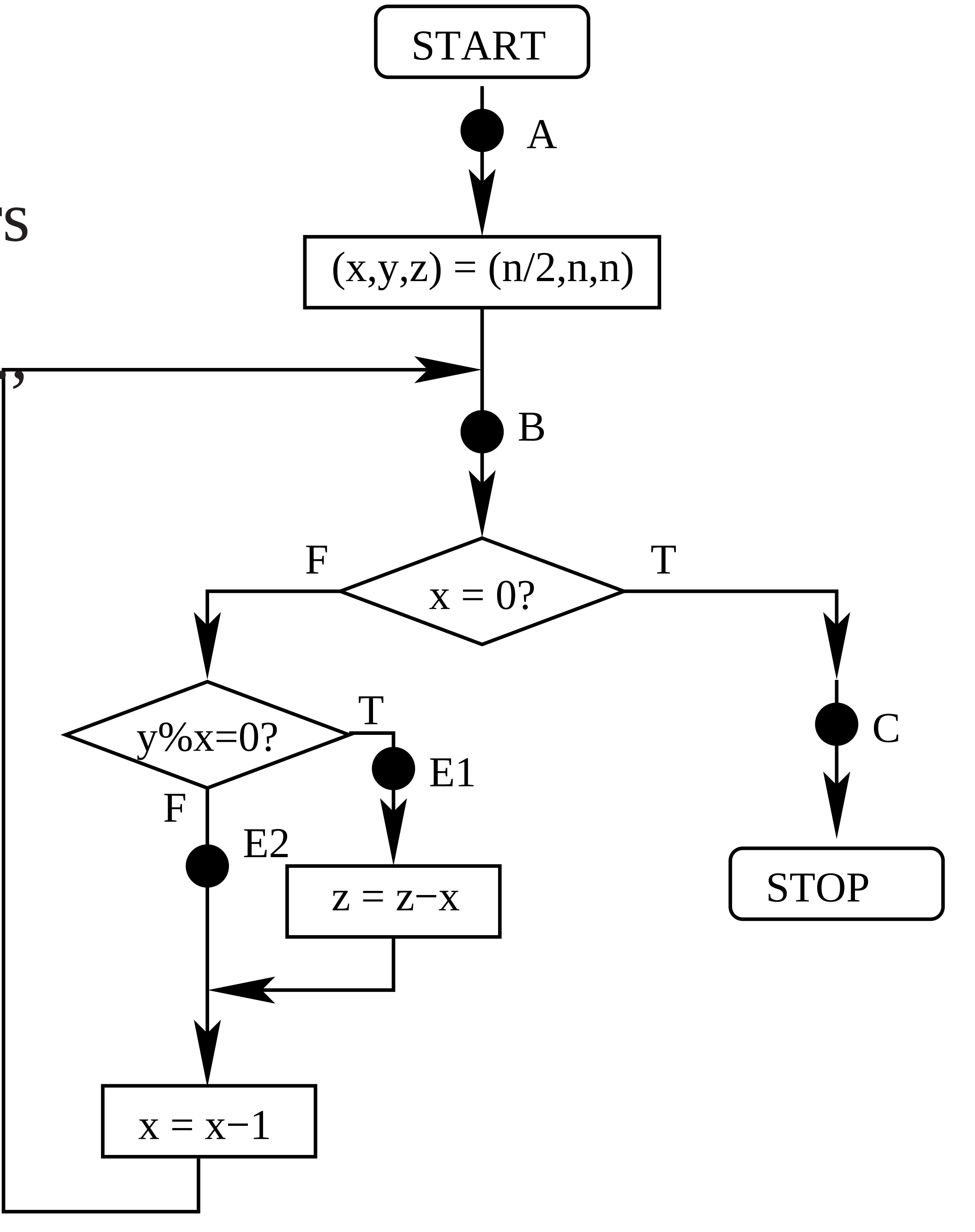
Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- Compiling srv-programs
- *Floyd-Hoare logics for (s)rv-programs*
- Miscellaneous
- Conclusions

Floyd's method for flowchart programs

Floyd's method for flowcharts:

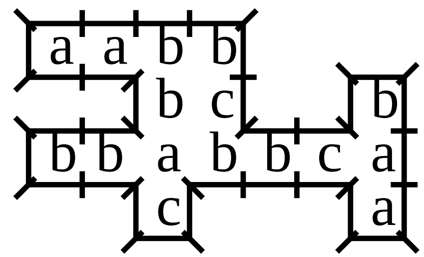
- a program for perfect numbers
- *cut-points* and *assertions*, e.g.,
  ϕ_B : "$0 \leq x \wedge y = n \geq 2$
  $\wedge z = n - \sum_{d|n, x<d<n} d$"
- *invariance conditions*, e.g.,
  $\phi_B \wedge C_{p(B,E1,B)} \Rightarrow \sigma_2(\phi_B)$
- *termination*: no infinite computation

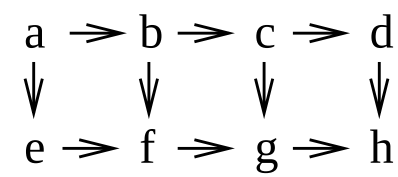


Grids and scenarios

Grids:

aabbabb
abbcdbb
bbabbca
ccccaaa





(a) (b) (c)

Standard interpretation:

- *columns* - processes
- *rows* - process interactions (nonblocking message passing)
- left-to-rigth and top-to-bottom causality

Contour-and-contents representation of grids:

The grid in (b) is represented as:

- Contour: $e^4s^2e^2n^1e^1s^3w^1n^1w^3s^1w^1n^1w^2n^1e^2n^1w^2n^1$
- Contents: $a^2b^3cb^3ab^2caca$.

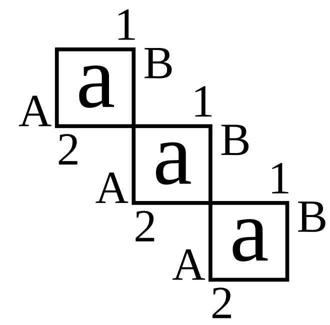
..Grids and scenarios

Scenarios:

(a)

```
 1 1 1
AaBbBbB
 2 1 1
AcAaBbB
 2 2 1
AcAcAaB
 2 2 2
```

(b)



Scenario = Grid + Data [around its letters]

Contour-and-contents representation of scenarios:

The scenario in (b) is represented as:

- Contour: $e_1 s_B e_1 s_B e_1 s_B w_2 n_A w_2 n_A w_2 n_A$
  (or, shortly, $(e_1 s_B)^3 (w_2 n_A)^3$)
- Contents: aaa.

Verification of rv-programs

A framework for rv-program verification:

Three steps:

- find an appropriate set of *contours* and *assertions* (it should be a *finite* and *complete* set);

  [complete = all scenarious of the associated FIS may be decomposed into such contours]
- fill in the contours with all *possible scenarios*; and
- prove the *invariance condition*, i.e., these scenarios respect the border assertions.

Except for the guess of assertions, the proof is finite and fully automatic.

..Verification of rv-programs

Assertions:

- *contours* with *assertions on state and class variables*;
- example:

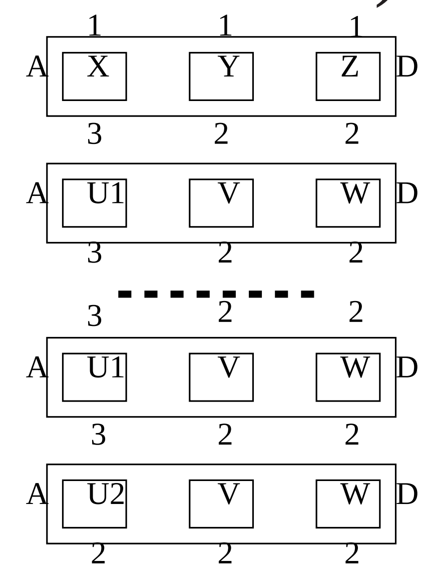
$$e_1\{x = x_0\}s_D e_2\{z = x_0 - 1\}\ldots$$

means:

— go towards east having on left the state 1 satisfying condition $x = x_0$;
— then go towards south having on left the class D with no condition (i.e., $True$);
— then go towards east having on left the state 2 with condition $z = x_0 - 1$; etc.

..Verification of rv-programs

Basic step (for contour $C = e_3e_2e_2s_Dw_2w_2w_3n_A$, i.e., middle row):



- *Assertion*: $\exists k.\ 0 < k \leq \lfloor x_0/2 \rfloor$ such that
  $e_3\{x = k\}e_2\{y = x_0\}$
  $\cdot e_2\{z = x_0 - \sum_{d|x_0, k<d<x_0} d\}s_D$
  $\cdot w_2\{z = x_0 - \sum_{d|x_0, k-1<d<x_0} d\}$
  $\cdot w_2\{y = x_0\}w_3\{x = k-1\}n_A$
- Possible *scenarious*: $U1 \triangleright V \triangleright W$
- Backwards *substitution* σ [south-east from north-west]
- *Invariance condition* $\psi_C \wedge \psi_W \wedge \psi_N \Rightarrow \sigma(\psi_E) \wedge \sigma(\psi_S)$ is reduced to:

$$z = x_0 - \sum_{d|x_0, k<d<x_0} d$$
$$\Rightarrow z - \phi(k) = x_0 - \sum_{d|x_0, k-1<d<x_0} d$$
$$x = k \Rightarrow x - 1 = k - 1$$

..Verification of rv-programs

Partial correctness, using a row partition:

Top row:

- cut-contour: $e_1e_1e_1s_Dw_2w_2w_3n_A$
- assertion:
  $$e_1\{x = x_0\}e_1e_1s_D \cdot w_2\{z = x_0\}w_2\{y = x_0\}w_3\{x = \lfloor x_0/2 \rfloor\}n_A$$
- scenario: $X \triangleright Y \triangleright Z$
- invariant condition: true

..Verification of rv-programs

(..Partial correctness, using a row partition)

Middle row:

- cut-contour: $e_3e_2e_2s_Dw_2w_2w_3n_A$
- assertion:

  $\exists k.\ 0 < k \leq \lfloor x_0/2 \rfloor$ such that:

  $e_3\{x = k\}e_2\{y = x_0\}$

  $\cdot e_2\{z = x_0 - \sum_{d|x_0, k<d<x_0} d\}s_D$

  $\cdot w_2\{z = x_0 - \sum_{d|x_0, k-1<d<x_0} d\}$

  $\cdot w_2\{y = x_0\}w_3\{x = k-1\}n_A$
- scenario: $U1 \triangleright V \triangleright W$, *provided the condition $k-1>0$ is true*
- invariant condition: true

..Verification of rv-programs

(..Partial correctness, using a row partition)

Bottom row:

- cut-contour: $e_3e_2e_2s_Dw_2w_2w_2n_A$
- assertion:

  $\exists k.\ (0 < k \leq \lfloor x_0/2 \rfloor)$ such that:

  $e_3\{x = k\}e_2\{y = x_0\}$

  $\cdot e_2\{z = x_0 - \sum_{d|x_0,k<d<x_0} d\}s_D$

  $\cdot w_2\{z = x_0 - \sum_{d|x_0,0<d<x_0} d = 0\}w_2w_3n_A$
- scenario: $U2 \triangleright V \triangleright W$, *provided the condition $\neg(k-1 > 0)$ is true*
- invariant condition: true

..Verification of rv-programs

(..Partial correctness, using a row partition)

Final step:

- Partial correctness of this rv-program:
  for each scenario

$$(X \triangleright Y \triangleright Z) \cdot (U1 \triangleright V \triangleright W)^r \cdot (U2 \triangleright V \triangleright W)$$

  the assertion

$$e_1\{x = x_0\}e_1e_1(s_D)^r$$
$$\cdot\, w_2\{z = 0 \text{ iff } x_0 \text{ is a perfect number}\}$$
$$\cdot\, w_2w_2(n_A)^r$$

  is true.

..Verification of rv-programs

Termination:

- no infinite scenarious [the 1st column is finite]

Verification by column partition:

- similar proof, but slightly more complicated

Verification of structured rv-programs

Hoare logics for *structured rv-programs*:

- it has been *partially developed*
- it was used to verify the *correctness of the termination detection protocol*
- its rules are *sound*, but we have *no claim on thier completeness*...

Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems $\leftarrow$ *[nfa]*
- Rv-programs $\leftarrow$ *[flowchart programs]*
- Structured rv-programs $\leftarrow$ *[while programs]*
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- *Miscellaneous*
- Conclusions

Miscellaneous

Contents:

- *State-explosion & flattening*
- Representing Message Sequence Charts

State-explosion & flattening

It looks that this

- flattening operator is responsible for the well-known *state-explosion problem* which occurs in the verification of concurrent (object-oriented) systems

We hope that

- the lifting of the verification techniques from paths to grids may avoid this problem

State-explosion & flattening

Suppose all actions of a grid w are distinct. Then

> *Proposition: For any $z \in \flat(w)$ there exist timing weights for actions such that the overall time provided by the schedule z is minimal.*

[Rules for time analysis:

—each action may start as soon as possible;

—if two actions are completed at the same time, then they may be put in the flattening sequence in any order.]

This shows that

- *any static* scheduling procedure (e.g., by rows, or by columns, or by diagonals, etc.) does *not* provide the *maximal speedup*;
- we need to consider *all flattened words* as possible execution sequences

State-explosion & flattening

Experimental results (for a rectangular grid of type $m \times n$):

The number of sequential executions $\varphi(m,n)$ associated to a (small) rectangular $m \times n$ grid:

$m \backslash n$	2	3	4	5	6	7
2	2	5	14	42	132	429
3	—	42	462	6,006	87,516	1,385,670
4	—	—	24,024	1,662,804	140,229,804	13,672,405,890
5	—	—	—	701,149,020	396,499,770,810	278,607,172,289,160
6	—	—	—	—	1,671,643,033,734,960	9,490,348,077,234,178,440
7	—	—	—	—	—	475,073,684,264,389,879,228,560

..State-explosion & flattening

Theoretical results:

- a *partial grid* is the part of a usual grid which remains after a number of steps of the flattening procedure have been applied
- a partial grid is of *type* $(l_1; l_2; \ldots; l_m)$ if it has l_1 elements in the 1st line, l_2 in the 2nd line, etc., where $l_1 \leq l_2 \leq \ldots \leq l_m$
- let $\varphi_{l_1;l_2;\ldots;l_m}$ denotes the *number of words* associated by the flattening operator to a partial grid of type $(l_1; l_2; \ldots; l_m)$
- finally, assign to each cell a_i of a partial grid a number k_i representing the *sum of the distances* (number of cells) from a_i to the west and north borders, counting a_i only ones

..State-explosion & flattening

Theorem:

For a partial grid of type $(l_1;l_2;\ldots;l_m)$ with p cells carrying the distances $k_1,\ldots,k_p$, we have

$$\varphi_{l_1;l_2;\ldots;l_m} = \frac{p!}{k_1\ldots k_p}$$

An example is on right:
—its type is (1;1;1;2;4) (9 cells);
—the numbers in the cells show the sums of west plus north distances;
—$\varphi_{l_1;l_2;\ldots;l_m} = \frac{9!}{(1)\cdot(2)\cdot(3)\cdot(1\cdot5)\cdot(1\cdot2\cdot4\cdot8)} = 189$

			1
			2
			3
		1	5
1	2	4	8

This is the famous Frame-Robinson-Thrall theorem; the formula in the theorem is known as "*hook formula*".

..State-explosion & flattening

Corollaries

1. $\varphi(m,n) = \frac{(m\cdot n)!}{[1\cdot 2\cdot \ldots \cdot n]\cdot[2\cdot 3\cdot \ldots \cdot (n+1)]\cdot \ldots \cdot [m\cdot (m+1)\cdot \ldots \cdot (m+n-1)]}$

2. *The complexity of* $\varphi(n,n)$ *is* $O(n^{n^2})$.

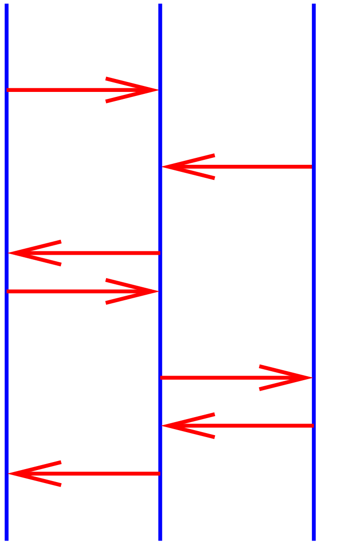
Miscellaneous

Contents:

- State-explosion & flattening
- *Representing Message Sequence Charts*

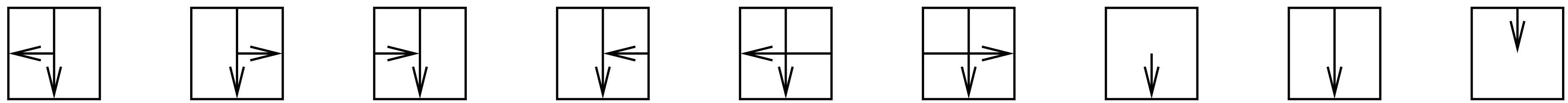
Message sequence charts:

- a model for specifying process interaction in a simple way using message passing and vertical time ordering
- adopted in UML as a basic tool for system specification
- possible extensions, e.g. LSC (live sequence charts)

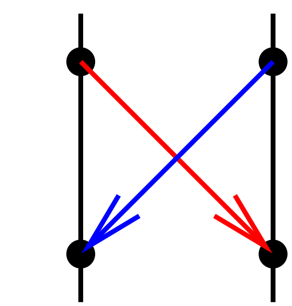


..FIS **vs. MSC**

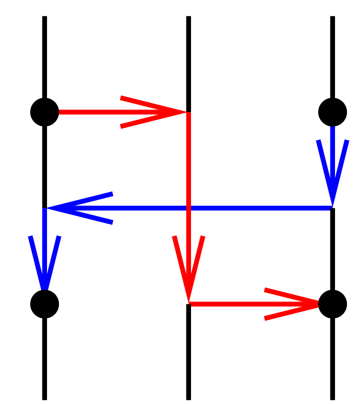
We use a particular alphabet



whose letters are interpreted as:



—(sendL) send a message to a left neighbor;
—(sendR) send a message to a right neighbor;
—(recL) receive a message from a left neighbor;
—(recR) receive a message from a right neighbor;



—(passL) pass a message from right to left;
—(passR) pass a message from left to right;
—(init) start a process;
—(void) idle a process;
—(end) end a process, respectively.

..FIS vs. MSC

Over this alphabet, finite interactive systems are more powerful than MSC. With additional restrictions we may capture the power of usual MSC's:

(α) each line has the following type: $init^\dagger$ or $end^\dagger$ or

$$(sendR \triangleright passR^\dagger \triangleright recL + recR \triangleright passL^\dagger \triangleright sendL + void)^\dagger;$$

(β) each column is of the type

$$init \cdot (sendL + sendR + recL + recR + passL + passR)^\star \cdot end$$

A *MSC-like* FIS is a FIS over V_{MSC} which satisfies (α) and (β).

Theorem:

Grid languages recognized by horizontally acyclic MSC-like FIS*'s over V_{MSC} correspond to MSC's.*

Contents: