Interactive systems with registers and voices

Gheorghe Stefanescu
Faculty of Mathematics and Computer Science
University of Bucharest

Pisa, Italy, 4-th June, 2007
Contents:

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

- **space-time duality “thesis”**

- **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
High level: temporal data

Software component

new job

new state

job requirement

memory state

High level: x: Int, y: Array[1..10] of Int;

tape

space

time

stream
Processes and transactions

Proc 1  Proc 2  Proc 3

Trans 1

Trans 2

Trans 3
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.
Three allocations of a temporal linked list on a stream: The 1st starts at time $t = 10$ and is

\[
\begin{array}{cccccccccccccccccccc}
\hline
.. & I & w & i & l & l & b & e & o & n & M & o & o & n & n & e & x & t & w & e & e & k & .. \\
\end{array}
\]

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

\[
\begin{array}{cccccccccccccccccccc}
\hline
.. & n & e & x & t & w & e & e & k & I & w & i & l & l & b & e & o & n & M & o & o & n & .. \\
\end{array}
\]

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

\[
\begin{array}{cccccccccccccccccccc}
\hline
.. & b & e & e & e & e & I & i & k & l & l & M & o & o & n & n & n & t & w & w & x & .. \\
.. & 3 & 4 & 1 & 6 & 2 & 7 & 2 & 6 & 3 & 2 & 3 & 5 & 1 & 7 & 1 & 2 & 3 & 6 & 2 & 0 & 2 & 3 & 1 & 0 & 2 & 4 & \hline
\end{array}
\]
Space-time converters; computation in space

\textit{space-to-time} and \textit{time-to-space} converters may be used to change \textit{computation in time} into a \textit{computation in space} paradigm.

\[
\begin{align*}
\text{n x y - computed in space} \\
\text{time-to-space} \quad \text{space-to-time}
\end{align*}
\]
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
A specification for perfect numbers:

3 components $C_x, C_y, C_z$ where:

- $C_x$: read $n$ from north and write
  
  \[ n \sim \lfloor n/2 \rfloor \sim (\lfloor n/2 \rfloor - 1) \sim \ldots \sim 2 \sim 1 \text{ on east}; \]

- $C_y$: read $n \sim \lfloor n/2 \rfloor \sim (\lfloor n/2 \rfloor - 1) \sim \ldots \sim 2 \sim 1$ from west and write
  \[ n \sim \phi(\lfloor n/2 \rfloor) \sim \ldots \sim \phi(2) \sim \phi(1) \text{ on east} \]
  
  \[ \phi(k) = \text{“if } k \text{ divides } n \text{ then } k \text{ else 0”}; \]

- $C_z$: read $n \sim \phi(\lfloor n/2 \rfloor) \sim \ldots \sim \phi(2) \sim \phi(1)$ from west and subtract from the first the other numbers.

These components are composed \textit{horizontally}. The global input-output specification: \textit{if the input number in $C_x$ is $n$, then the output number in $C_z$ is 0 iff $n$ is perfect.}
Two scenarios for perfect numbers:

Types are denoted as \( \langle \text{west} \mid \text{north} \rangle \rightarrow \langle \text{east} \mid \text{south} \rangle \)

**Our (s)rv-scenarios are similar with the tiles of Bruni-Gadducci-Montanari, et.al.**
The 1st AGAPIA program Perfect1 (construction by rows):

\[(X \# Y \# Z) \% \text{while}_t(x > 0) \{ U \# V \# W \}\]

Its type is **Perfect1** : \(\langle \text{nil}|\text{sn};\text{nil};\text{nil} \rangle \rightarrow \langle \text{nil}|\text{sn};\text{sn};\text{sn} \rangle\).

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

\[(X \ % \ \text{while}_t(x>0)\{U\} \ % \ U1)\]
\[# (Y \ % \ \text{while}_t(tx>-1)\{V\} \ % \ V1)\]
\[# (Z \ % \ \text{while}_t(tx>-1)\{W\} \ % \ W1)\]

Its type is Perfect2 : \(\langle\text{nil}|\text{sn};\text{nil};\text{nil}\rangle \rightarrow \langle\text{nil}|\text{nil};\text{nil};\text{sn}\rangle\).

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* ← [nfa]
- Rv-programs ← [flowchart programs]
- Structured rv-programs ← [while programs]
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions
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.
Causality in a grid/scenario:

\[ a \rightarrow b \rightarrow c \rightarrow d \]
\[ \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \]
\[ e \rightarrow f \rightarrow g \rightarrow h \]

Action vs. inter-action:

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

\[ b : \text{LangGrids}(V) \rightarrow \text{LangWords}(V) \]

maps sets of \textit{grids} to \textit{sets of strings} representing their topological sorting. Example:

—start with \( \begin{array}{c}
abcd \\
\hline
efgh
\end{array} \); there is one minimal element \( a \); after its deletion we get \( \begin{array}{c}
bcd \\
\hline
efgh
\end{array} \);

—the minimal elements are \( b \) and \( e \); suppose we choose \( b \);
what remains is \( \begin{array}{c}
\cd \\
\hline
efgh
\end{array} \); and so on;

—finally a usual word, say \( \text{abecfgdh} \), is obtained.

Actually, \( b( \begin{array}{c}
abcd \\
\hline
efgh
\end{array} ) = \) \[ \{ \text{abcdefgh}, \text{abcdedfgh}, \text{abcedfgh}, \text{abcefdgh}, \text{abcfgdh}, \text{abcdfgh}, \text{abecfdgh}, \text{abefcdgh}, \text{abefcgdh}, \text{aebcdfgh}, \text{aebcfdgh}, \text{aebfcdgh}, \text{aebfcdgh} \} \]
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** $\varepsilon_k$:
  
  —with $w(\varepsilon_k) = e(\varepsilon_k) = 0$ and $n(\varepsilon_k) = s(\varepsilon_k) = k$

- **horizontal identity** $\lambda_k$:
  
  —with $w(\lambda_k) = e(\lambda_k) = k$ and $n(\lambda_k) = s(\lambda_k) = 0$
**Signature:** two sets of regular algebra operators, sharing the additive part

\[ +, \ 0, \cdot, *, \ |, \triangleright, \dagger, \ - \]

\[ (+, 0, \cdot, *, \mid) \] - a Kleene signature for the vertical dimension

\[ (+, 0, \triangleright, \dagger, \mid) \] - a Kleene signature for the horizontal dimension

**Two-dimensional regular expressions** (denoted \(2\text{RegExp}(V)\)):

\[
E ::= a(\in V) \ | \ 0 \ | \ E + E \ | \ E \cap E \ | \ E \cdot E \ | \ E^* \ | \ | \ | \ E \triangleright E \ | \ E^\dagger \ | \ -
\]

**Theorem:**

\(2\text{RegExp}(V)\) *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)

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

- \(|a| = \{a\}; \; |0| = \emptyset; \; |E + F| = |E| \cup |F|
- \(|| = \{\varepsilon_0, \ldots, \varepsilon_k, \ldots\}
- \(|E \cdot F| = \{v \cdot w : v \in |E| \land w \in |F|\}
- \(|E^*| = \{v_1 \cdot \ldots \cdot v_k : k \in \mathbb{IN} \land v_1, \ldots, v_k \in |E|\} \cup ||
- \(|\neg| = \{\lambda_0, \ldots, \lambda_k, \ldots\}
- \(|E \triangleright F| = \{v \triangleright w : v \in |E| \land w \in |F|\}
- \(|E^\dagger| = \{v_1 \triangleright \ldots \triangleright v_k : k \in \mathbb{IN} \land v_1, \ldots, v_k \in |E|\} \cup |\neg|}
Example (finite interactive system): A FIS $S$ and its

**graphical representation:**

```
1 -> a -> B
|    |    |
A   a   B
|    |    |
V    V    V
|    |    |
c 2  2  2
```

**“cross” representation:**

- $A_1 a B_2$, $A_2 c A_2$, and $B_1 b B_1$
- $A$, $1$ are initial
- $B$, $2$ final

**textual representation**

- $a: <A|1> \rightarrow <B|2>$, etc.
Example: A successful scenario for recognizing a grid:

- Given a grid \( w = \begin{array}{c}
\text{abb} \\
\text{cab} \\
\text{cca}
\end{array} \), start with initial states/classes on north/west borders;

- parse the grid: 
  \[
  \begin{array}{c}
  w_0 = \begin{array}{ccc}
  1 & 1 & 1 \\
  \text{Aa} & \text{b} & \text{b} \\
  \text{Ac} & \text{a} & \text{b} \\
  \text{Ac} & \text{c} & \text{a}
  \end{array} \\
  w_1 = \begin{array}{ccc}
  1 & 1 & 1 \\
  \text{AaBb} & \text{b} & \text{2} \\
  \text{Ac} & \text{a} & \text{b} \\
  \text{Ac} & \text{c} & \text{a}
  \end{array} \\
  w_2 = \begin{array}{ccc}
  1 & 1 & 1 \\
  \text{AaBbBb} & \text{b} & \text{2} \\
  \text{Ac} & \text{a} & \text{b} \\
  \text{Ac} & \text{c} & \text{a}
  \end{array}
  \]

  \[
  w_3 = \begin{array}{ccc}
  1 & 1 & 1 \\
  \text{AaBbBb} & \text{2} & \text{1} \\
  \text{AcAa} & \text{b} & \text{2} \\
  \text{Ac} & \text{c} & \text{a}
  \end{array} \\
  \ldots \\
  w_9 = \begin{array}{ccc}
  1 & 1 & 1 \\
  \text{AaBbBbBb} & \text{2} & \text{1} \\
  \text{AcAaBbBb} & \text{2} & \text{2} \\
  \text{AcAcAaBb} & \text{2} & \text{2}
  \end{array}
  \]

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

\[ L(S) = \{ \text{a’s on the diagonal, top-right half of b’s, and bottom-left half of c’s} \}. \]
State projection and class projection

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

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

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

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

\[ L(S) = L(\text{state}(S))^\dagger \cap L(\text{class}(S))^* \]

Actually,

\[ L(S) = (b^* \cdot a \cdot c^*)^\dagger \cap (c^\dagger \triangleright a \triangleright b^\dagger)^* \]

**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.*
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

\[ \text{http://math.uni-heidelberg.de/logic/bb/2dpapers.html} \]

Notice: 2-dimensional languages are also known as “picture” 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 ← [nfa]
- Rv-programs ← [flowchart programs]
- Structured rv-programs ← [while programs]
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions
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 parsing for abb, cab, cca:

```
1 1 1 1 1 1 1 1 1
Aa b b AaBb b AaBbBb AaBbBb AaBbBb ...
2 1 2 1 2 1 2 1 2
Ac a b Ac a b Ac a b Ac a b Ac a b
2 1 2 1 2 1 2 1 2
Ac c a Ac c a Ac c a Ac c a Ac c a
2 1 2 1 2 1 2 1 2
```

Diagram:
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.
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];

Y::

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

Z::

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

U::

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

V::

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

W::

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

Operational semantics:

- defined in terms of scenarious

Relational semantics:

- input-output relation generated by all possible scenarious
..RV-programs

**Associating a FIS:**

or, equivalently,

\[
\begin{align*}
\begin{array}{ccc}
1 & 2 & 3 \\
A & X & B \\
3 & & \\
\end{array} & , & \begin{array}{ccc}
1 & 2 & 3 \\
B & Y & C \\
2 & & \\
\end{array} & , & \begin{array}{ccc}
1 & 2 \\
C & Z & D \\
2 & & \\
\end{array} \\
\begin{array}{ccc}
3 & 2 & 2 \\
A & U1 & B \\
3 & & \\
\end{array} & , & \begin{array}{ccc}
3 & 2 & 2 \\
A & U2 & B \\
2 & & \\
\end{array} & , & \begin{array}{ccc}
2 & 2 \\
B & V & C \\
2 & & \\
\end{array} & , & \begin{array}{ccc}
2 & 2 & 2 \\
C & W & D \\
2 & & \\
\end{array}
\end{align*}
\]
... and a grid language:

\[ \begin{array}{ccc}
X & Y & Z \\
U_1 & V & W \\
\ldots & \ldots & \ldots \\
U_1 & V & W \\
U_2 & V & W \\
\end{array} \]

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:

\[
X ::= \text{module}\{\text{listen } t\_\text{vars};\}\{\text{read } s\_\text{vars};\}
\{\text{code;}\}\{\text{speak } t\_\text{vars};\}\{\text{write } s\_\text{vars};\}
\]

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

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):

- \( P_1 \text{ comp}\{tv\}\{sv\} P_2 \)
- \( \text{while}\{tv\}\{sv\}\{C\}\{P\} \)
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*
Syntax of AGAPIA v0.1:

Interfaces

\[
\begin{align*}
SST & ::= \text{nil} \mid sn \mid sb \\
     & \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^* \\
ST & ::= (SST) \\
     \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^* \\
STT & ::= \text{nil} \mid tn \mid tb \\
     \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^* \\
TT & ::= (STT) \\
     \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^* \\
\end{align*}
\]

Expressions

\[
\begin{align*}
V & ::= x : ST \mid x : TT \\
     \mid V(k) \mid V.k \mid V.[k] \mid V@k \mid V@[k] \\
E & ::= n \mid V \mid E + E \mid E \ast E \mid E - E \mid E / E \\
B & ::= b \mid V \mid B&B \mid B|B \mid !B \mid E < E \\
\end{align*}
\]

Programs

\[
\begin{align*}
W & ::= \text{null} \mid new x : SST \mid new x : STT \\
     \mid x := E \mid if(B)\{W\}\{else\{W\} \\
     \mid W;W \mid while(B)\{W\} \\
M & ::= \text{module}\{\text{listen} x : STT\}\{\text{read} x : SST\} \\
     \mid W \}\{\text{write} x : SST\} \\
P & ::= \text{null} \mid M \mid if(B)\{P\}\{else\{P\} \\
     \mid P\%P \mid P\#P \mid P\$P \\
     \mid while_\bot(B)\{P\} \mid while_\bot(B)\{P\} \\
     \mid while_\bot(B)\{P\} \\
\end{align*}
\]
Example: A program for distributed termination detection

\[
P = I_1 \# \text{for}_{s}(tid=0; tid<tm; tid++) \{ I_2 \# $
\text{while}_{st}(! (\text{token.col==white} \ & \ \& \ \text{token.pos==0})) \{
\text{for}_{s}(tid=0; tid<tm; tid++) \{ R \}}
\}
\]

where:

\[
I_1 = \text{module}\{ \text{listen nil} \}\{ \text{read m} \}\{$
\text{tm=m; token.col=black; token.pos=0;}
\}\{ \text{speak tm, tid, msg[ ], token(col, pos)} \}\{ \text{write nil} \}
\]

\[
I_2 = \text{module}\{ \text{listen tm, tid, msg[ ], token(col, pos)} \}\{
\text{read nil} \}\{$
\text{id=tid; c=white; active=true; msg[id]=null;}
\}\{ \text{speak tm, tid, msg[ ], token(col, pos)} \}\{ \text{write id, c, active} \}$
\]
Example: Termination detection

R = module\{listen tm, tid, msg[], token(col, pos)\}
  \{read id, c, active\}{
    if(msg[id]!={}){ //take my jobs
      msg[id] = {};
      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\}
A *run* (for termination detection program)

\begin{verbatim}
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}}
\end{verbatim}
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
Simple spatial types are defined by:

\[ \text{SST} ::= \text{nil} \mid \text{sn} \mid \text{sb} \mid (\text{SST} \cup \text{SST}) \mid (\text{SST}, \text{SST}) \mid (\text{SST})^* \]

("","" - associative with "nil" neutral element; "∪" - associative)

Example:

\[
((((\text{sn})^*)^*, \text{sb}, (\text{sn}, \text{sb}, \text{sn})^*)^*, (\text{sb} \cup \text{sn}))
\]

represents the following data structure (for a process)

\[
x: \text{struc1}[], \text{where} \\
x \text{struc1} = (\text{a: Int}[], \text{b: Bool,} \text{c: struc2}[]), \text{where} \\
c \text{struc2} = (\text{p: Int, q: Bool, r: Int})
\]

\[
y: \text{Bool or Int}
\]

Simple temporal types — similar
Interface types

Spatial types are defined by::

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

(“;” - associative with “nil” neutral element; “∪” - associative)

Example:

$$((sn)^*)^*; \text{nil}; sb; ((sn)^*;)^*$$

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
Interface types

Reshaping types

- Interface types may be changed using special morphisms.
- Examples: $(sn;)^* \mapsto (sn)^*$ and $(tn;)^* \mapsto (tn)^*$ (left).
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

\[ E ::= n \mid V \mid E + E \mid E \times E \mid E - E \mid E / E \]

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

\[ W ::= \text{null} \mid \text{new } x : \text{SST} \mid \text{new } x : \text{STT} \]
\[ \mid x ::= E \mid \text{if}(B)\{W\}\text{else}\{W\} \]
\[ \mid W;W \mid \text{while}(B)\{W\} \]

Modules

\[ M ::= \text{module}\{\text{listen } x : \text{STT}\}\{\text{read } x : \text{SST}\} \]
\[ \{ W \}\{\text{speak } x : \text{STT}\}\{\text{write } x : \text{SST}\} \]

Agapia v0.1 programs

\[ P ::= \text{null} \mid M \mid \text{if}(B)\{P\}\text{else}\{P\} \]
\[ \mid P\%P \mid P\#P \mid P\$P \]
\[ \mid \text{while } \downarrow(B)\{P\} \mid \text{while } \downarrow s(B)\{P\} \mid \text{while } \downarrow t(B)\{P\} \]
Temporal (or vertical) composition and while
- denoted “%” and \textit{while}_t
- composition of modules/programs via spatial interfaces (“usual” composition)

Spatial (or horizontal) composition and while
- denoted “#” and \textit{while}_s
- composition of modules/programs via temporal interfaces

Spatio-temporal (or diagonal) composition and while
- denoted “$” and \textit{while}_st
- composition of modules/programs via both spatial and temporal interfaces
Scenarios:

1: \( x = 4 \), \( A: X \)

1: \( B: tx = 4 \) Y

1: \( C: Z \) D:

3: \( x = 2 \), \( A: U \)

2: \( y = 4 \), \( B: tx = 2 \) V

2: \( z = 4 \), \( C: W \) D:

2: \( z = 2 \)

(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 “∪”)
- .$k$ - the $k$-th element of a structure (separated by “,”)
- .[$k$] - the $k$-th element of an array (defined by $(...)^*$)
- @$k$ - the $k$-th process/transaction (separated by “;”)
- @$[k]$ - the $k$-th process/transaction of an array of processes/transactions (defined by “((...;)*”)

**Examples:**

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

The typing morphism - defined by a mapping

\[ \sigma : P \mapsto (st_\sigma(P), \langle w_\sigma(P) | n_\sigma(P) \rangle \mapsto \langle e_\sigma(P) | s_\sigma(P) \rangle) \]

where:

- \( st \in \{ \text{ok}, \text{war0}, \text{war1}, \text{err} \} \) says the program is:
  - \( \text{ok} \) - well-typed;
  - \( \text{war0}, \text{war1} \) - partially well-typed with two levels of warnings;
  - \( \text{err} \) - 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*;
   - *war0* - if their types are *equal*, but *not a singleton*;
   - *war1* - 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 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 \textit{only the variables} occurring in the \textit{listen/speak} and \textit{read/write} statements with their associated types.
Typing structured rv-programs: On programs, the typing morphism is inductively defined by:

**Vertical composition:**

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

where

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

**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

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

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

where denoting

- $$P_1 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \text{singleton}$$
- $$Q_1 := s_{\sigma(S)} = n_{\sigma(S)} = \text{singleton}$$
- $$P_2 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \neg\text{singleton}$$
- $$Q_2 := s_{\sigma(S)} = n_{\sigma(S)} = \neg\text{singleton}$$
- $$P_3 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) \neq \emptyset$$
- $$Q_3 := s_{\sigma(S)} \cap n_{\sigma(S)} \neq \emptyset$$
- $$P_4 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) = \emptyset$$
- $$Q_4 := s_{\sigma(S)} \cap n_{\sigma(S)} = \emptyset$$

we have

$$st = \begin{cases} 
\min(\text{ok}, st_B, st_{\sigma(S)}) & \text{if } P_1 \land Q_1 \\
\min(\text{war0}, st_B, st_{\sigma(S)}) & \text{if } P_2 \land (Q_1 \lor Q_2) \lor (P_1 \lor P_2) \land Q_2 \\
\min(\text{war1}, st_B, st_{\sigma(S)}) & \text{if } P_3 \land (Q_1 \lor Q_2 \lor Q_3) \lor (P_1 \lor P_2 \lor P_3) \land Q_3 \\
\text{err} & \text{if } P_4 \lor Q_4 
\end{cases}$$
Spatial while: $\sigma(\text{while}_s(B)\{S\})$

- is similar to the temporal while

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

- similar to the temporal while
- ...but slightly more complicate as 3 pairs of interfaces are to be compared:
  - first, $\sigma_B$ 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 (termination detection) Denote
\[ a = (tm, tid, msg[], token), \quad b = (id, c, active), \quad c = (m). \] Then:

Init:
\[
\begin{align*}
I_1 &\leftarrow (ok, \langle \text{nil}|c \rangle \rightarrow \langle a|\text{nil} \rangle) \\
I_2 &\leftarrow (ok, \langle a|\text{nil} \rangle \rightarrow \langle a|b \rangle) \\
\text{for}_s(\ )\{I_2\} &\leftarrow (ok, \langle a|\text{nil} \rangle \rightarrow \langle a|(b;)^* \rangle) \\
I_1&#\text{for}_s(\ )\{I_2\} &\leftarrow (ok, \langle \text{nil}|c \rangle \rightarrow \langle a|(b;)^* \rangle)
\end{align*}
\]

Repeat:
\[
\begin{align*}
R &\leftarrow (ok, \langle a|b \rangle \rightarrow \langle a|b \rangle) \\
\text{for}_s(\ )\{R\} &\leftarrow (ok, \langle a|(b;)^* \rangle \rightarrow \langle a|(b;)^* \rangle) \\
\text{while}_st\{\text{for}_s(\ )\{R\}\} &\leftarrow (\text{war}0, \langle a|(b;)^* \rangle \rightarrow \langle a|(b;)^* \rangle)
\end{align*}
\]

Full program:
\[
\begin{align*}
P &\leftarrow (\text{war}0, \langle \text{nil}|c \rangle \rightarrow \langle a|(b;)^* \rangle)
\end{align*}
\]
Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems \(\leftarrow [nfa]\)
- Rv-programs \(\leftarrow [flowchart\ programs]\)
- Structured rv-programs \(\leftarrow [while\ programs]\)
- \textit{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 a 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
Example: The translation of *if* is based on the following component whose implementation as a rv-program is rather tedious.
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 flowcharts:

- a program for perfect numbers
- *cut-points* and *assertions*, e.g.,
  \[ \phi_B : \left( 0 \leq x \land y = n \geq 2 \right) \land \left( z = n - \sum_{d|n, x<d<n} d \right) \]
- *invariance conditions*, e.g.,
  \[ \phi_B \land C_p(B,E_1,B) \Rightarrow \sigma_2(\phi_B) \]
- *termination*: no infinite computation
Grids and scenarios

Grids:

aabbabb
abbcdbb
bbabbbca
ccccaaa

(a) (b) (c)

Standard interpretation:

- **columns** - processes
- **rows** - process interactions (nonblocking message passing)
- left-to-right 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: $a^2b^3cb^3ab^2caca$. 

Scenarios:

(a)

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

(b)

```
1
A

2
A

1
B
```

Scenario = Grid + Data [around its letters]

Contour-and-contents representation of scenarios:

The scenario in (b) is represented as:

- Contour: $e_1s_Be_1s_Be_1s_Bw_2n_Aw_2n_Aw_2n_A$
  (or, shortly, $(e_1s_B)^3(w_2n_A)^3$)
- Contents: $aaa$. 
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.*
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.
**Basic step** (for contour \( C = e_3 e_2 e_2 s_D w_2 w_2 w_3 n_A \), i.e., middle row):

- **Assertion:** \( \exists k. \ 0 < k \leq \lfloor x_0/2 \rfloor \) such that
  \[
  \begin{align*}
  e_3 \{x = k\} & e_2 \{y = x_0\} \\
  & \cdot e_2 \{z = x_0 - \sum d | x_0, k < d < x_0 \} s_D \\
  & \cdot w_2 \{z = x_0 - \sum d | x_0, k - 1 < d < x_0 \} \\
  & \cdot w_2 \{y = x_0\} w_3 \{x = k - 1\} n_A
  \end{align*}
  \]

- Possible **scenarious:** \( U1 \triangleright V \triangleright W \)

- Backwards **substitution** \( \sigma \) [south-east from north-west]

- **Invariance condition** \( \psi_C \land \psi_W \land \psi_N \Rightarrow \sigma(\psi_E) \land \sigma(\psi_S) \) is reduced to:
  \[
  \begin{align*}
  z &= x_0 - \sum d | x_0, k < d < x_0 d \\
  \Rightarrow z - \phi(k) &= x_0 - \sum d | x_0, k - 1 < d < x_0 d \\
  x = k &\Rightarrow x - 1 = k - 1
  \end{align*}
  \]
Partial correctness, using a row partition:

**Top row:**

- cut-contour: \( e_1 e_1 e_1 s_D w_2 w_2 w_3 n_A \)

- assertion:
  \[
  e_1 \{ x = x_0 \} e_1 e_1 s_D \\
  \cdot w_2 \{ z = x_0 \} w_2 \{ y = x_0 \} w_3 \{ x = \lfloor x_0/2 \rfloor \} n_A
  \]

- scenario: \( X \triangleright Y \triangleright Z \)

- invariant condition: true
(..Partial correctness, using a row partition)

**Middle row:**

- **cut-contour:** \( e_3 e_2 e_2 s_D w_2 w_2 w_3 n_A \)
- **assertion:**
  \[
  \exists k. \ 0 < k \leq \lfloor x_0/2 \rfloor \text{ such that:} \\
  e_3 \{ x = k \} e_2 \{ y = x_0 \} \\
  \cdot e_2 \{ z = x_0 - \sum_{d \mid x_0, k < d < x_0} d \} s_D \\
  \cdot w_2 \{ z = x_0 - \sum_{d \mid x_0, k-1 < d < x_0} d \} \\
  \cdot w_2 \{ y = x_0 \} w_3 \{ x = k - 1 \} n_A
  \]
- **scenario:** \( U_1 \triangleright V \triangleright W \), *provided the condition \( k - 1 > 0 \) is true*
- **invariant condition:** true
(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

\[(X \triangleright Y \triangleright Z) \cdot (U_1 \triangleright V \triangleright W)^r \cdot (U_2 \triangleright V \triangleright W)\]

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 scenerious [the 1st column is finite]

Verification by column partition:

- similar proof, but slightly more complicated
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 their completeness...
Contents:

- Generalities
- A glimpse on AGAPIA programming
- Finite interactive systems
- Rv-programs
- Structured rv-programs
- Compiling srv-programs
- Floyd-Hoare logics for (s)rv-programs
- Miscellaneous
- Conclusions
Miscellaneous

Contents:

- **State-explosion & flattening**
- Representing Message Sequence Charts
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
Suppose all actions of a grid \( w \) are distinct. Then

**Proposition:** For any \( z \in b(w) \) 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:

<table>
<thead>
<tr>
<th>$m \backslash n$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>14</td>
<td>42</td>
<td>132</td>
<td>429</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>42</td>
<td>462</td>
<td>6,006</td>
<td>87,516</td>
<td>1,385,670</td>
</tr>
<tr>
<td>4</td>
<td>—</td>
<td>—</td>
<td>24,024</td>
<td>1,662,804</td>
<td>140,229,804</td>
<td>13,672,405,890</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>701,149,020</td>
<td>396,499,770,810</td>
<td>278,607,172,289,160</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1,671,643,033,734,960</td>
<td>9,490,348,077,254,318,440</td>
</tr>
<tr>
<td>7</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>475,073,684,264,389,879,228,560</td>
</tr>
</tbody>
</table>
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.
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

\[
\Phi_{l_1; l_2; \ldots; l_m} = \frac{p!}{k_1 \cdots k_p}
\]

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;
—\(\Phi_{l_1; l_2; \ldots; l_m} = \frac{9!}{(1)(2)(3)(1\cdot5)(1\cdot2\cdot4\cdot8)} = 189\)

This is the famous Frame-Robinson-Thrall theorem; the formula in the theorem is known as “hook formula”.
Corollaries

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

2. The complexity of \( \varphi(n, n) \) is \( O(n^{n^2}) \).
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)
We use a particular alphabet

\[ \text{—(sendL)} \text{ send a message to a left neighbor; } \]
\[ \text{—(sendR)} \text{ send a message to a right neighbor; } \]
\[ \text{—(recL)} \text{ receive a message from a left neighbor; } \]
\[ \text{—(recR)} \text{ receive a message from a right neighbor; } \]
\[ \text{—(passL)} \text{ pass a message from right to left; } \]
\[ \text{—(passR)} \text{ pass a message from left to right; } \]
\[ \text{—(init)} \text{ start a process; } \]
\[ \text{—(void)} \text{ idle a process; } \]
\[ \text{—(end)} \text{ end a process, respectively.} \]
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

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

(β) each column is of the type

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

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.*
Rv-systems

Contents:

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