On compiling structured interactive programs with registers and voices

Cezara Dragoi and *Gheorghe Stefanescu*

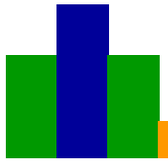
Faculty of of Mathematics and Computer Science
University of Bucharest

Sofsem'08, January 21-th, 2008

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
- Conclusions



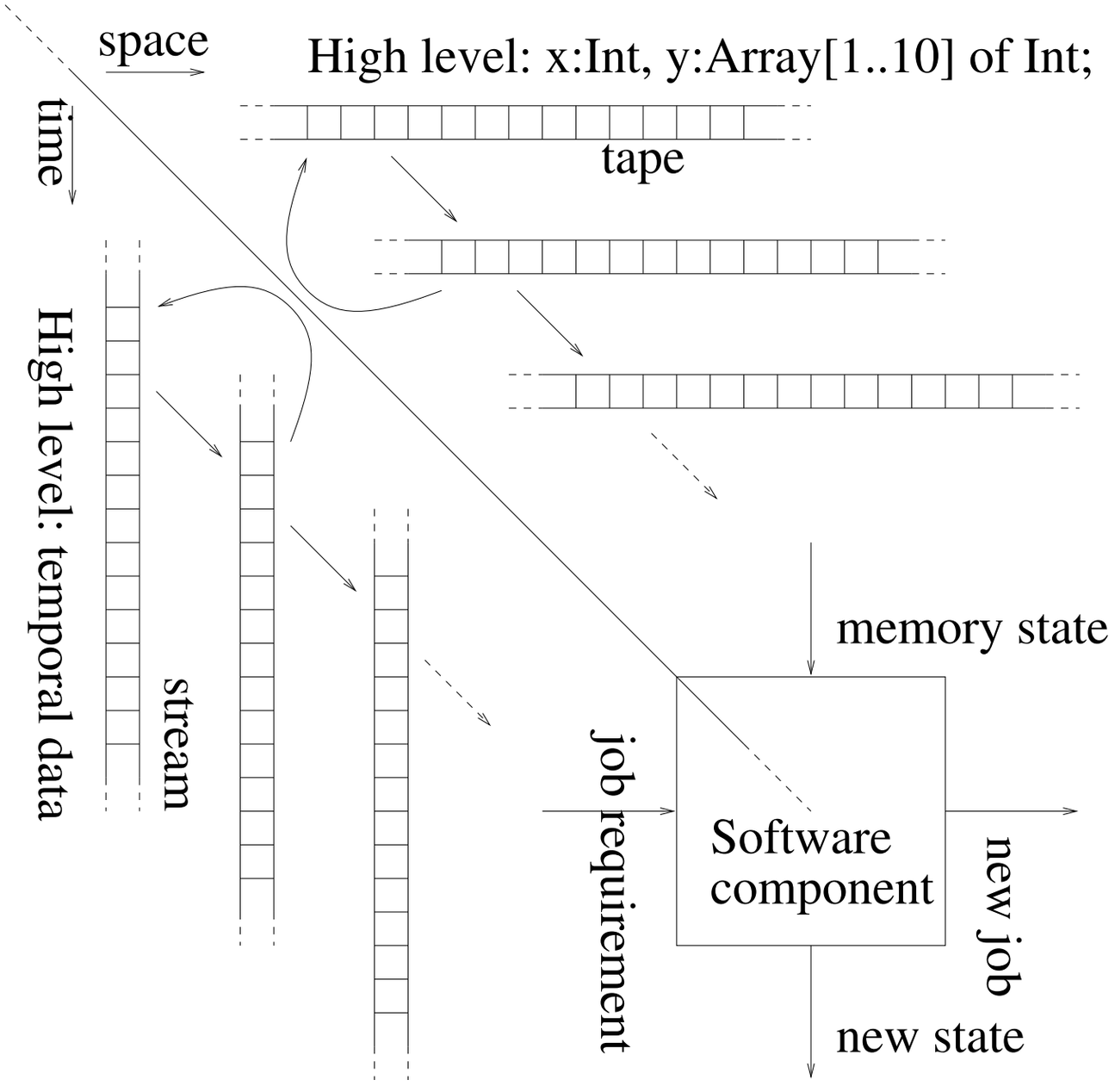
History

History

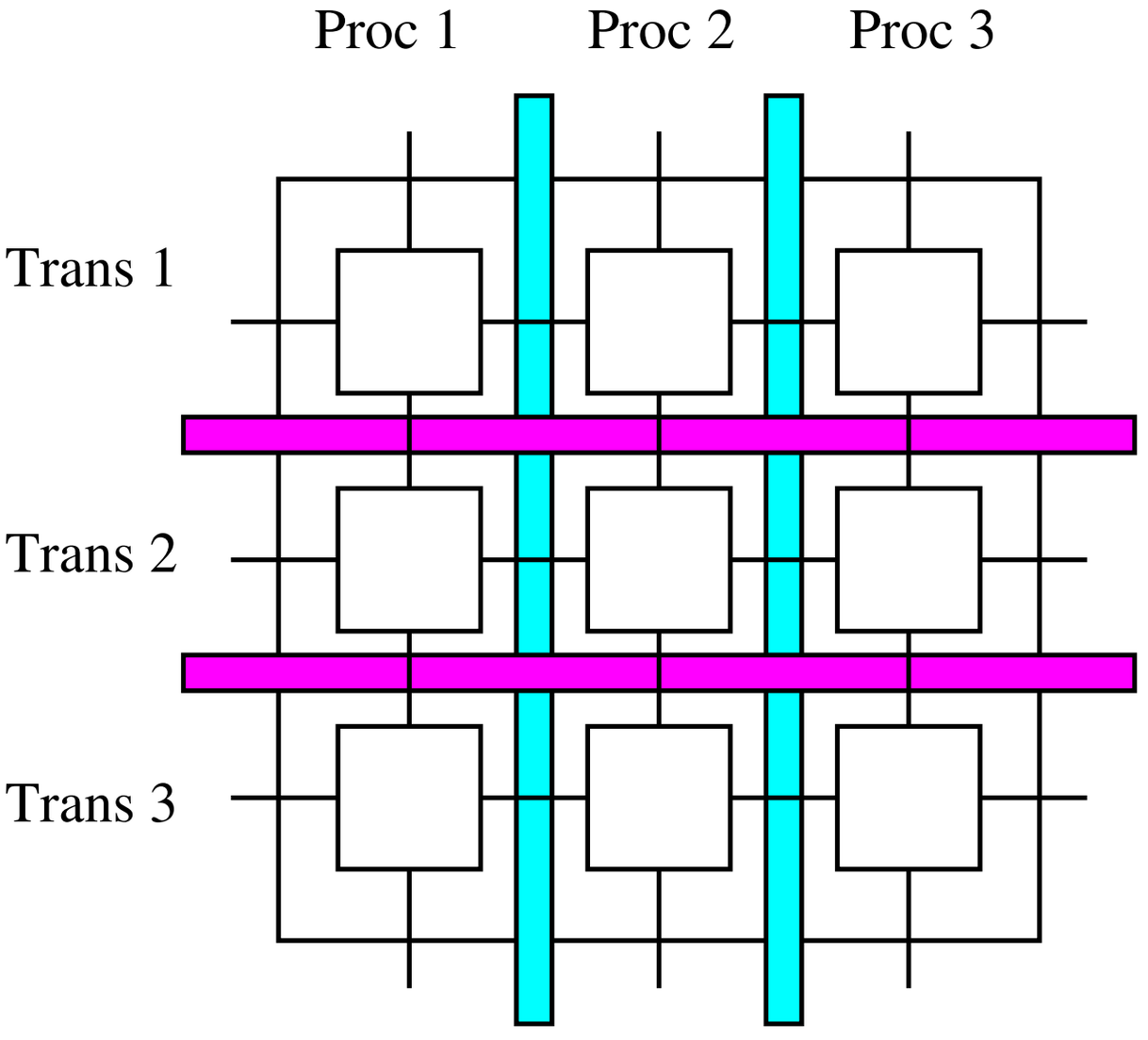
- *space-time duality "thesis"*
  - Stefanescu, *Network algebra*, Springer 2000
- *finite interactive systems*
  - Stefanescu, Marktoberdorf Summer School 2001
- *rv-systems* (interactive systems with registers and voices)
  - Stefanescu, NUS, Singapore, summer 2004
- *structured rv-systems*
  - Stefanescu,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

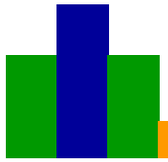
ST-Dual picture

ST-Dual picture



Processes and transactions





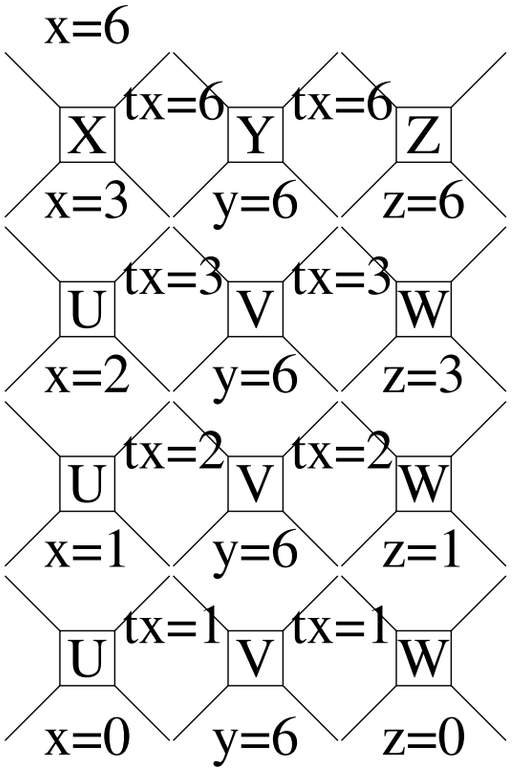
Rv-systems

Contents:

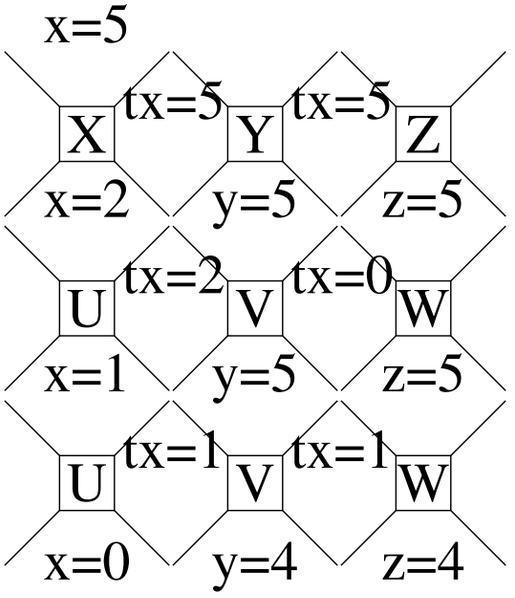
- Generalities
- *A glimpse on AGAPIA programming*
- Finite interactive systems ← *[nfa]*
- Rv-programs ← *[flowchart programs]*
- Structured rv-programs ← *[while programs]*
- Compiling srv-programs
- Conclusions

..Srv-programs for perfect numbers

Two scenarios for perfect numbers:



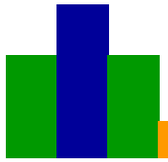
(a)



(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

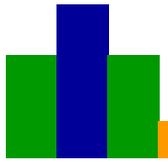
The 1st AGAPIA program Perfect1 (construction by rows):

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

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

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

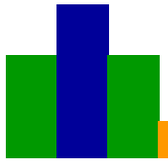
The 2nd AGAPIA program Perfect2 (construction by columns):

```
(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:

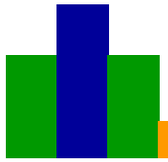
```
U1:: module{listen nil;}{read x:sn;}
          {tx:tn; tx=-1;}{speak tx;}{write nil;}
V1:: module{listen tx:tn;}{read y:sn;}
          {null;}{speak tx;}{write nil;}
W1:: module{listen tx:tn;}{read z:sn}
          {null;}{speak nil;}{write z;}
```



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
- 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
```

)

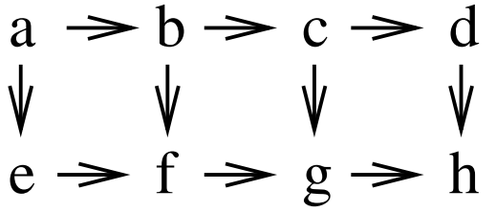
A grid p has a *north* (resp. *south, west, east*) border denoted as

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

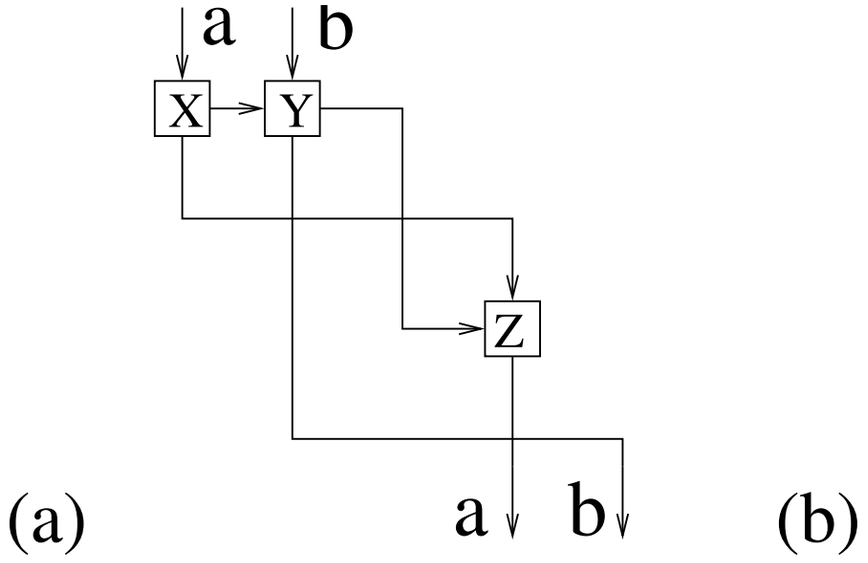
Notice: The requirement to have a rectangular area may be weakened, e.g., one may require to have a connected area, not a rectangular one.

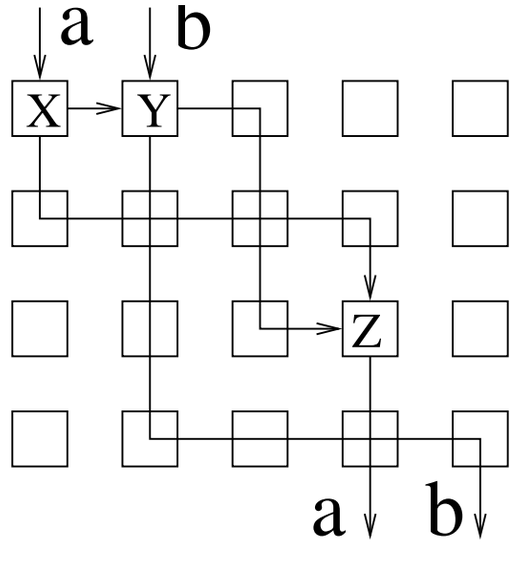
..Grids (or planar words)

Causality in a grid/scenario:



Action vs. inter-action:





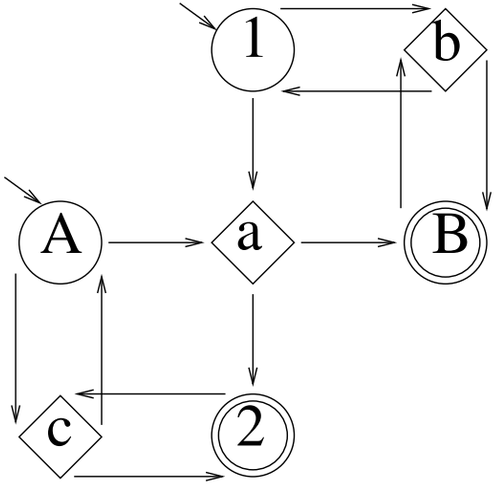
(b)

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

Finite interactive systems

Finite interactive systems:

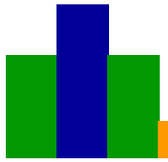
- *states*: 1,2 [1-initial; 2-final]
- *classes*: A, B [A-initial; B-final]
- *transitions*: a, b, c



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
```



FIS vs. 2-dimensional languages

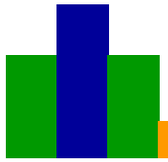
Theorem:

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);

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:

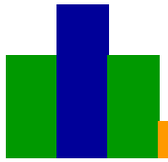
1. *Context-sensitive word languages coincide with the projection on the 1st row of the* FIS *recognizable grid languages.*

2. *The emptiness problem for* FIS*'s is undecidable.*

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
- Conclusions

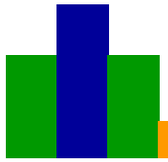


RV-programs

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];

Y::

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

Z::

(C,1)	z : sInt
tx : tInt	z = tx; 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::

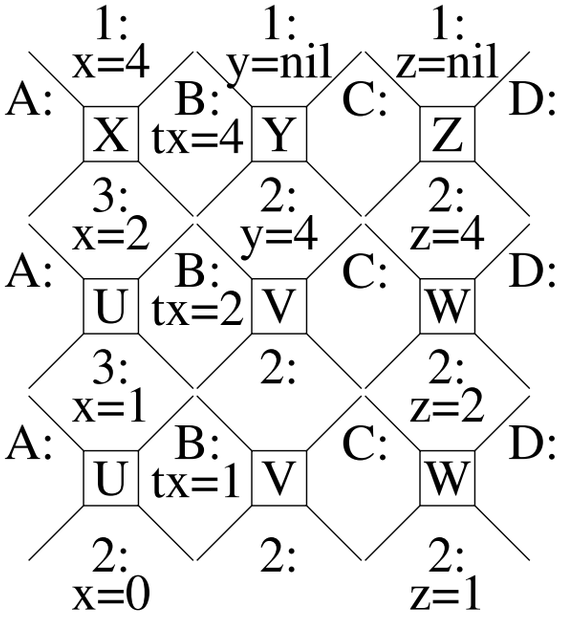
(B,2)	y : sInt
tx : tInt	if(y%tx != 0) tx = 0; goto [C,2];

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

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
- Conclusions

Structured rv-programs

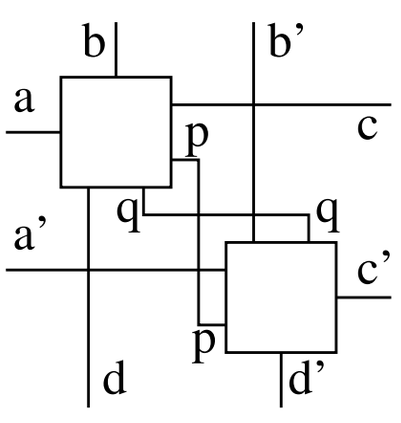
Syntax:

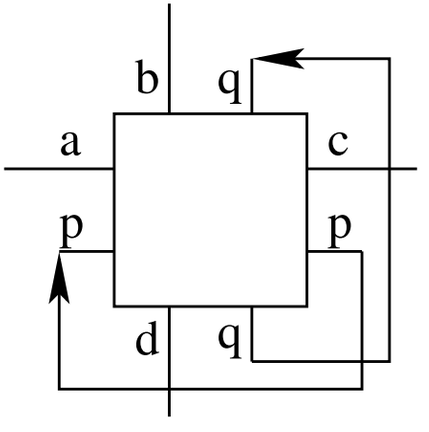
$$X ::= module\{listen\ t\_vars;\}\{read\ s\_vars;\}$$
$$\{code;\}\{speak\ t\_vars;\}\{write\ s\_vars;\}$$

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

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*):

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

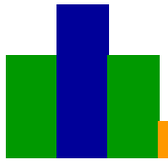




AGAPIA

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

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

Expressions

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

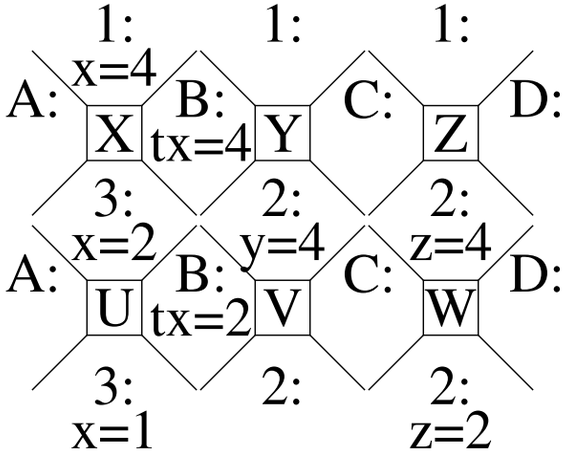
Programs

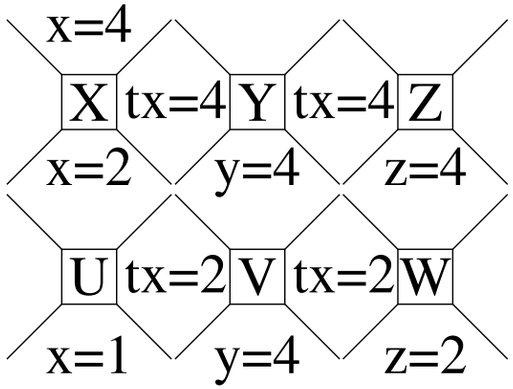
$$W ::= null \mid new\ x : SST \mid new\ x : STT \mid x := E \mid if(B)\{W\}else\{W\} \mid W; W \mid while(B)\{W\}$$
$$M ::= module\{listen\ x : STT\}\{read\ x : SST\}\{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}$$
$$P ::= null \mid M \mid if(B)\{P\}else\{P\} \mid P\%P \mid P\#P \mid P\$P \mid while\_t(B)\{P\}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

Scenarios

Scenarios:

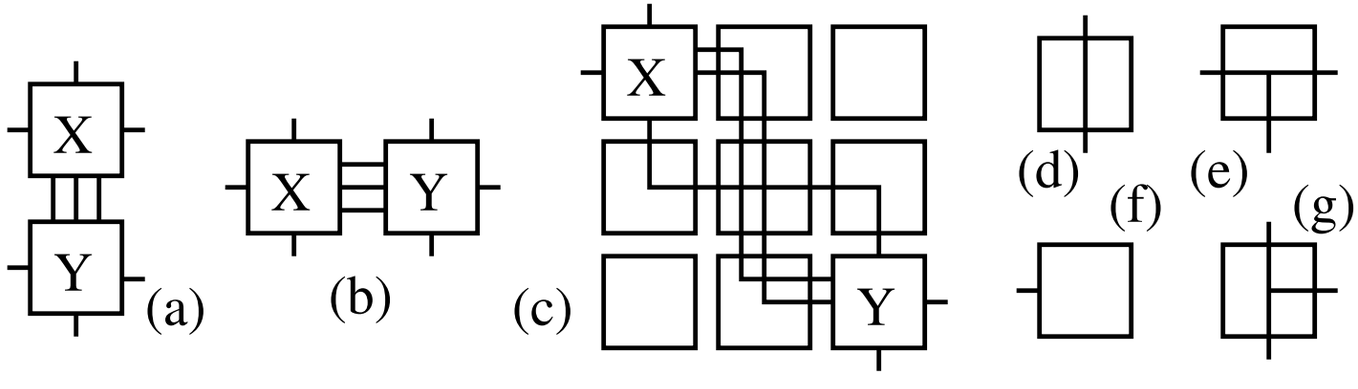
```
 1 1 1
AaBbBbB
 2 1 1
AcAaBbB
 2 2 1
AcAcAaB
 2 2 2
```





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

Srv-scenario operations:

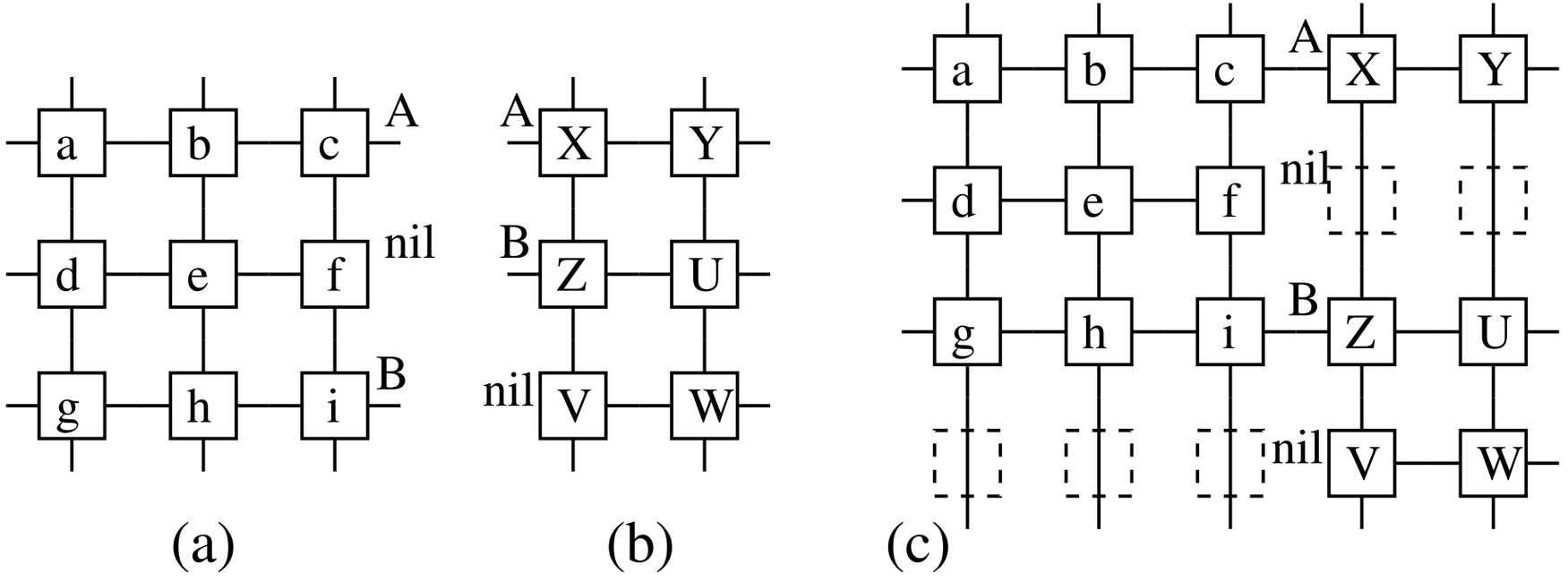


(4)

..Operations on srv-scenarios

..Srv-scenario operations:

- Details for horizontal composition



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

Example: Termination detection

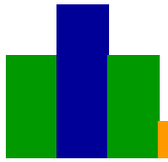
Example: A program for distributed termination detection

```
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}
```

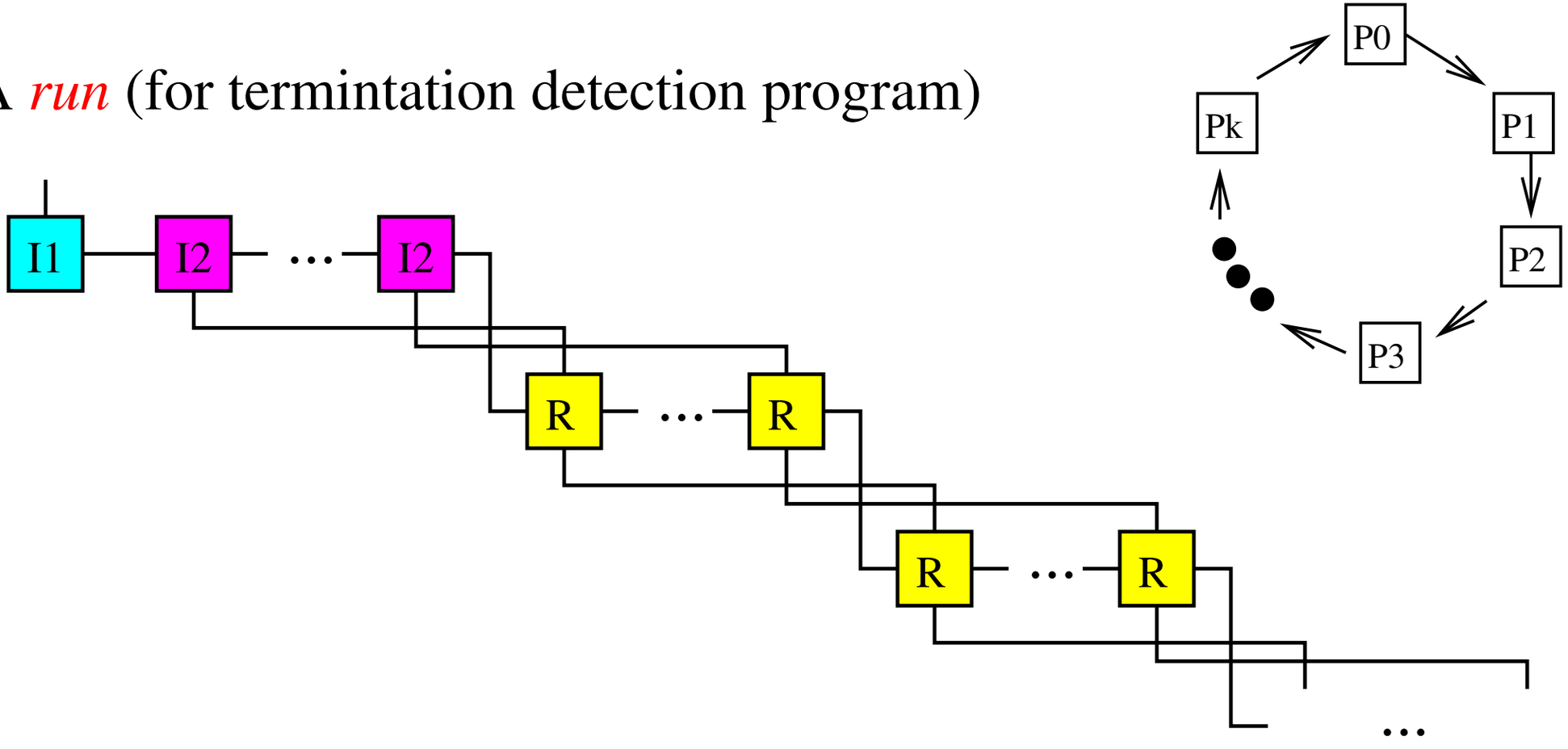


..Example: Termination detection

```
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)

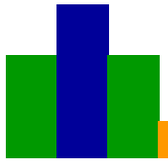



```
I1# for_s(tid=0;tid<tm;tid++){I2}#
$ while_st(!(token.col==white && token.pos==0)){
    for_s(tid=0;tid<tm;tid++){R}}
```

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*
- Conclusions



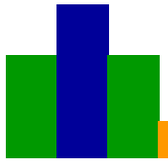
Compiling srv-programs

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

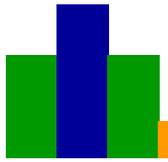


A transformation on rv-programs

Lemma:

(i) *For each rv-program P there is an equivalent rv-program P' where all initial/final states/classes only occur on one border of the scenarios, and never inside.*

(ii) *(ii) Moreover, for each border, one can manage to have a unique state/class for each interface type of the scenario cells.*



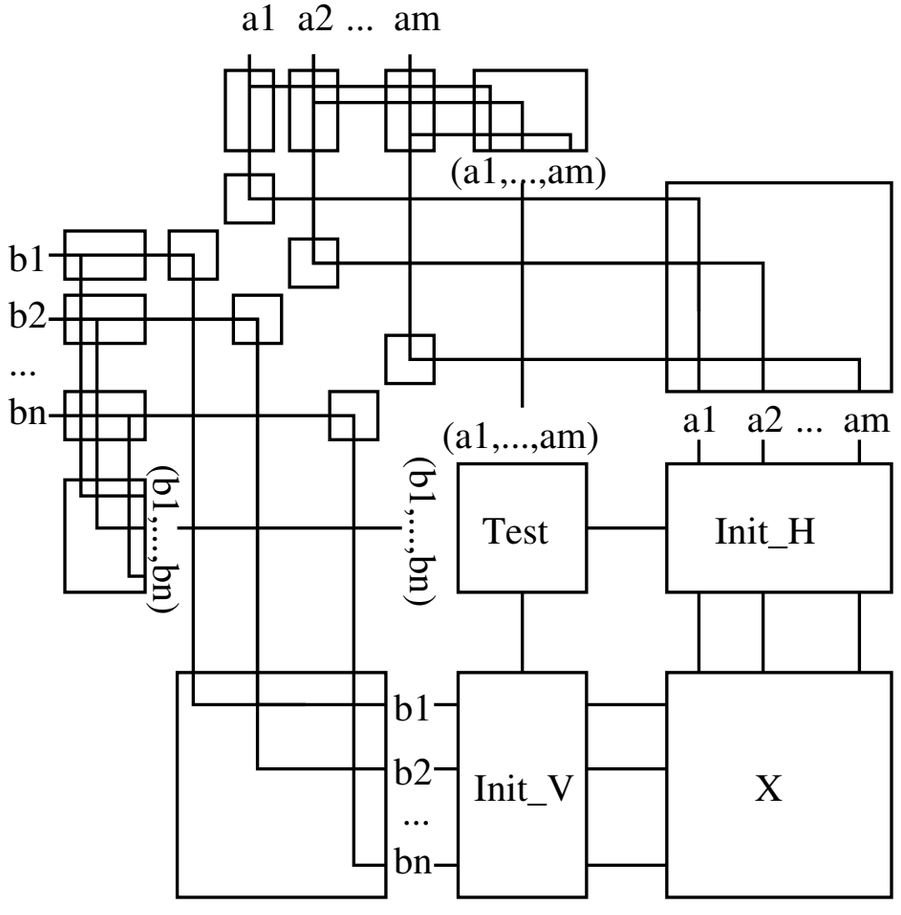
The translation

Definiton:

- in three steps: $Tr(\bullet) = Tr3(Tr2(Tr1(\bullet)))$:
  - $Tr1$ from structured rv-programs to rv-programs
  - apply Lemma (i)
  - apply Lemma (ii)
- natural translation, inductively defined on: modules, compositions, if, while (the last two a bit more complicate)

..Compiling srv-programs

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



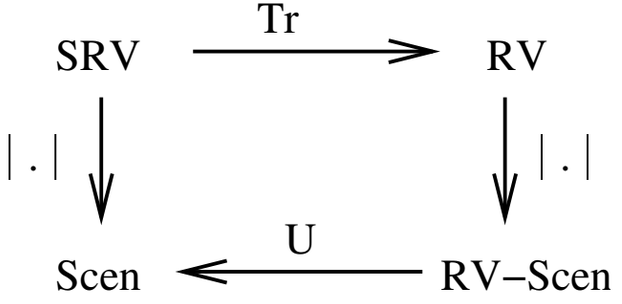
whose implementation as a rv-program is rather tedious.

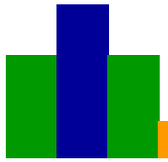
Translation correctness

Theorem:

(i) *The above translation Tr, from structured rv-programs to rv-programs, is correct with respect to the input-output semantics.*

(ii) *(ii) Moreover, the translation weakly preserves the set of running scenarios. That is, up to mild scenario transformations regarding the use of tests and constants (recorders, speakers, etc.) the associated scenarios are the same.*





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
- *Conclusions*