Interactive systems with registers and voices

Gheorghe Ştefănescu

Department of Computer Science, National University of Singapore
Email: gheorghe@comp.nus.edu.sg

Abstract

We present a model, a core programming language, specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems.

The model consists of *rv-systems* (*interactive systems with registers and voices*); it includes register machines, is space-time invariant, is compositional, may describe computations extending in both time and space, and is applicable to open, interactive systems. To achieve modularity in space the model uses *voices* (a voice is the time dual of a register) - they provide a high level organization of temporal data and are used to describe interaction interfaces of processes.

The programming language uses novel techniques for syntax and semantics to support computation in space paradigm. We describe *rv-programs* and base their syntax and operational semantics on FISs (*finite interactive systems*) and their grid languages (a FIS is a kind of 2-dimensional automaton specifying both control and interaction used in rv-programs).

The specification of rv-systems uses relations between input registers and voices and their output counterparts. The paper describes simple specifications for an OO-system and for an interactive game. The analysis techniques developed for rv-programs use finite automata, FISs, and an intermediary class of decomposable FISs.

Draft

Last updated: July 16, 2004

Introduction

Interactive systems are omnipresent - they range from describing low level interacting processes on the same machine, cluster, or distributed system to communicating agents in the Internet, human-computer, or human-human interaction.

This paper focusses on the foundations of interactive systems, presenting *rv-systems*, a model for interactive systems based on register machines and space-time duality. Actually, we identify three useful levels of abstraction for the study of interactive systems. On an increasing scale of complexity, they are:

— **Decomposable finite interactive systems:** This is the most abstract (or the simplest) level. It gives a rough approximation of the actual computations of an interactive system, including many scenarios for which no execution is possible. A decomposable FIS may be reduced to a pair of finite automata, one for control, the other for interaction. Decomposable FISs share with finite automata a few pleasant properties, including the decidability of language equivalence and a clean algebraic theory (the latter is not presented in this paper).

— **Finite interactive systems:** (introduced in [29]) This level is of a medium complexity, e.g., emptiness problem is undecidable[1], but membership problem is decidable. The model is able to filter out some impossible scenarios present in the previous setting, giving a finer approximation of the actual computations of an interactive systems.

— **Interactive systems with registers and voices:** (rv-systems) This is the most concrete/complex level. It includes register machines, is space-time invariant, is compositional, may describe computations extending in both time and space, and is applicable to open, interactive systems. To achieve modularity in space the model uses *voices* (a voice is the time dual of a register) - they provide a high level organization of temporal data and are used to describe interaction interfaces of processes.

Except for the above, other contributions of the paper include: the introduction of a specification formalism (for spatio-temporal specifications), of a core programming language (consisting of rv-programs), of a scenario-based operational semantics (for rv-programs), and of techniques for analyzing rv-programs.

On the style of the paper: Due to space limitation and novelty of the model, the paper pays more attention on introducing the concepts and interpreting the results, rather than on proofs. Some proof techniques may be borrowed from [28]; different techniques were used to discover Theorem 1.2, but the result is not new and a proof may be found in the literature. To keep the focus of the paper, some comments are collected in endnotes.

1 Grids and grid languages

1.1 Grids

Grids A *grid* [2] is a *rectangular* two-dimensional area filled in with letters of a given alphabet V. Their set is denoted by $V^{\dagger\star}$. Each letter in V is a two-dimensional atom having its own `north`, `south`, `west`, and `east` type. This typing is naturally extended to grids. More general grids may be used (removing the condition to have a rectangular area), but these general grids will be hardly used in this paper. It is important to notice that our grids are *logical*, not geometrical objects.

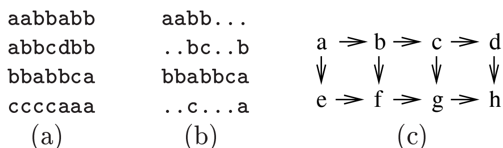


Figure 1: Grids

Examples of grids (see Fig. 1): The grid in (a) is a normal, rectangular grid - by default, a grid is considered of this type; in (b), a more general grid is presented; finally, (c) describes our standard order used in grids: each cell directly depends on its top and left neighbors.

In our standard interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a *nonblocking message passing discipline*: a process sends a message on the right, then continues its execution. We will see later that the convention to send messages left-to-right only is not restrictive.

Action vs. inter-action The convention of having only a left-to-right causality in grids may raise an important question:

> How *inter*-action may appear? A left process may send a message to a process on the right, but can not receive an answer back!

Or, put in other words, it looks that grids properly describe *actions* (like sending messages from masters to slaves), but not real *inter*-actions.

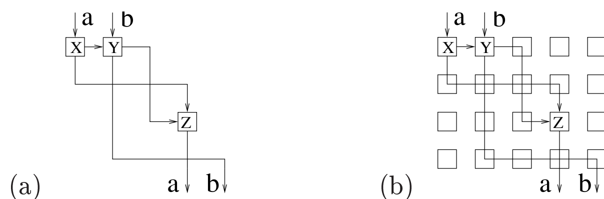


Figure 2: Action vs. inter-action

Let us recall that our grids are logical, not geometrical objects. A two-ways communication situation is isomorphic to the situation described in Fig. 2(a). If the alphabet of grid letters is rich enough to contain empty cell, identities, corners, and crossing, then, as in Fig. 2(b), the picture may be converted into a grid which faithfully captures the situation.

The flattening operator The *flattening operator*

$$\flat \; : V^{\dagger\star} \quad \to \quad \mathcal{P}(V^{\star})$$

maps grids to sets of words representing their topological sorting, as follows:
—Each grid w may be considered as an acyclic directed graph $g(w)$ drawing horizontal edges from each letter to its right neighbor and vertical edges from each letter to its bottom neighbor - see Fig. 1(c).
—Being acyclic, $g(w)$ and all of its subgraphs have *minimal vertices*, i.e., vertices without incoming arrows.
—The topological sorting procedure selects a minimal vertex, deletes it and its outgoing edges, and repeats this as long as possible. This way a word containing all the atoms in w is obtained.
—The set $\flat(w)$ consists of all words obtained as above varying vertex selection in all possible ways.

This flattening operator $\flat$ is naturally extended to a set of grids L by $\flat(L) = \{\flat(w) : \; w \in L\}$.

To have an example, take $\substack{\texttt{abcd}\\\texttt{efgh}}$; there is only one minimal element `a` and after its deletion we get $\substack{\texttt{bcd}\\\texttt{efgh}}$; now, there are two minimal elements `b` and `e` and suppose we choose `b` - what remains is $\substack{\texttt{cd}\\\texttt{efgh}}$; and so on; finally a word, say `abecfgdh`, is obtained. Actually,
$\flat(\substack{\texttt{abcd}\\\texttt{efgh}}) = \{$`abcdefgh`, `abcedfgh`, `abcefdgh`, `abcefgdh`, `abecdfgh`, `abecfdgh`, `abecfgdh`, `abefcdgh`, `abefcgdh`, `aebcdfgh`, `aebcfdgh`, `aebcfgdh`, `aebfcdgh`, `aebfcgdh`$\}$.

Why flattening? As we said, in the standard interpretation the letters in a grid represent statements. A

$m \backslash n$	2	3	4	5	6
2	2	5	14	42	132
3	5	42	462	6,006	87,516
4	14	462	24,024	1,662,804	140,229,804
5	42	6,006	1,662,804	701,149,020	396,499,770,810
6	132	87,516	140,229,804	396,499,770,810	1,671,643,033,734,960

Table 1: The number of words associated to a grid

grid gives a constraint on their execution: a statement can start only if its top and left neighbors have finished their work; except for this, there is no other restriction and independent statements may be executed in parallel. The maximal speedup is reached when the following *eager evaluation* is used: a statement starts precisely when the laziest of its top and left neighbors has finished his work. Let us apply this evaluation and record the statements in the order[3] they terminate. Then:

Proposition 1.1 *Suppose all letters in a grid w are distinct and their set is A. For each $v \in \flat(w)$, there exists a function $f : A \to \mathbb{N}$ assigning execution time to statements (letters) such that v is a termination sequence produced by the eager evaluation using f.*

Consequently, there is no static schedule of the statements (e.g., by rows, or by columns, or by diagonals, etc.) which is compatible to the maximal speedup and all sequences in $\flat(w)$ have to be considered as possible sequential versions.

Grids vs. words Let $\varphi(m,n)$ denotes the number of elements in $\flat(w)$, for an $m \times n$ grid w with distinct letters. We want to have an estimation of $\varphi(m,n)$. The results in a few particular cases are presented in Table 1. Below we give a mathematical formula for $\varphi(m,n)$ working in a larger context, defined below.

```
      1
      2
      3
    1 5
1 2 4 8
```

(1) A *partial grid* is that part of a rectangular grid which remains after a number of steps in the flattening procedure have been applied; the flattening operator $\flat$ is obviously defined on these grids. (2) A partial grid has *type* $(l_1; l_2; \ldots; l_m)$ if it has l_1 elements in the first line, l_2 elements in the second line, etc., where $l_1 \leq l_2 \leq \ldots \leq l_m$. (3) Let $\varphi_{l_1;l_2;\ldots;l_m}$ denote the number of words in $\flat(w)$, for a partial grid w of type $(l_1; l_2; \ldots; l_m)$ and having distinct letters. (4) Finally, to each cell a_i of a partial grid we assign a number k_i representing the *sum of the distances* (number of cells) from a_i toward the west and north borders, counting a_i only once.

An example of partial grid is above. Its type is (1;1;1;2;4) (9 cells). The numbers in the cells show the sums of distances toward the north and the west borders. As the theorem below shows, $\varphi_{1;1;1;2;4} = \frac{9!}{(1)\cdot(2)\cdot(3)\cdot(1\cdot5)\cdot(1\cdot2\cdot4\cdot8)} = 189.$

Theorem 1.2 *For a partial grid of type $(l_1; l_2; \ldots; l_m)$ with p cells carrying the distances $k_1, \ldots, k_p$*

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

So, $\varphi(m,n) = \frac{(m\cdot n)!}{[1\cdot2\cdot\ldots\cdot n]\cdot[2\cdot3\cdot\ldots\cdot(n+1)]\cdot\ldots\cdot[m\cdot(m+1)\cdot\ldots\cdot(m+n-1)]}.$

Thus, the order of magnitude of $\varphi(n,n)$ is[4] $\mathcal{O}(n^{n^2})$.

The result in the theorem is actually a famous result in combinatorics: Frame-Robinson-Thrall Theorem [15] (the formula is known as "*hook formula*") — it was independently rediscovered[5] in [9].

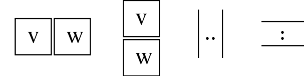
1.2 Specifying grid languages

Regular expressions *Basic 2-dimensional regular expressions* are defined by the following BNF syntax:

$E ::= a \mid 0 \mid E{+}E \mid E{\cap}E \mid E{\cdot}E \mid E^\star \mid \shortmid \mid E{\triangleright}E \mid E^\dagger \mid -$

Their set is denoted by $\mathsf{2RegExp}(V)$, where V is the specified alphabet. The *signature* consists of intersection and two collections of usual regular algebra operators, sharing the additive part. Specifically, $(+, 0, \cdot, ^\star, \shortmid)$ is a usual Kleene signature to be used for the vertical dimension, while $(+, 0, \triangleright, ^\dagger, -)$ is another Kleene signature to be used for the horizontal dimension.

From expressions to grids First, a few basic operations on grids.



For two grids v, w, the horizontal composite $v \triangleright w$ is the grid obtained putting v on left of w (it is defined only if $\mathtt{east}(v) = \mathtt{west}(w)$); the vertical composite $v \cdot w$ is the grid obtained putting v on top of w (it is defined only if $\mathtt{south}(v) = \mathtt{north}(w)$). For each natural number k, vertical identity ϵ_k is a grid with $\mathtt{west}(\epsilon_k) = \mathtt{east}(\epsilon_k) = 0$ and $\mathtt{north}(\epsilon_k) = \mathtt{south}(\epsilon_k) = k$; horizontal identity λ_k is a grid with $\mathtt{west}(\lambda_k) = \mathtt{east}(\lambda_k) = k$ and $\mathtt{north}(\lambda_k) = \mathtt{south}(\lambda_k) = 0$.

Now, the interpretation

$$|\ \ | : \mathsf{2RegExp}(V) \to \mathcal{P}(V^{\dagger\star})$$

from expressions to grid languages is defined by:

- $|a| = \{a\}$; $|0| = \emptyset$;
- $|E+F| = |E| \cup |F|$; $|E \cap F| = |E| \cap |F|$;
- $|E \cdot F| = \{v \cdot w : v \in |E| \wedge w \in |F|\}$;
- $|E^\star| = \{v_1 \cdot \ldots \cdot v_k : k \in \mathbb{N} \wedge v_1, \ldots, v_k \in |E|\}$;
- $|\shortmid| = \{\epsilon_0, \ldots, \epsilon_k, \ldots\}$;
- $|E \triangleright F| = \{v \triangleright w : v \in |E| \wedge w \in |F|\}$;
- $|E^\dagger| = \{v_1 \triangleright \ldots \triangleright v_k : k \in \mathbb{N} \wedge v_1, \ldots, v_k \in |E|\}$;
- $|-| = \{\lambda_0, \ldots, \lambda_k, \ldots\}$.

Examples (regular expressions)

1.
$$(a \cdot d^\star \cdot g) \triangleright (b \cdot e^\star \cdot h)^\dagger \triangleright (c \cdot f^\star \cdot i)$$
specifies the following grids

```
a b . . b c
d e . . e f
. . . . . .
. . . . . .
d e . . e f
g h . . h i
```

2.
$$(b^\star \cdot a \cdot c^\star)^\dagger \cap (c^\dagger \triangleright a \triangleright b^\dagger)^\star$$
specifies square grids as

```
abbb
cabb
ccab
ccca
```

1.3 Scenarios

As we already said, grids are used to describe computations - a letter in a grid represents a statement to be executed. A *scenario* is a grid enriched with information about data used at the borders of its letters.

The additional information on data around each letter may be given in an abstract form as in this picture (a name $A, B, 1, 2$) or in a more detailed form as in Fig. 3.

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

We do not describe the details of a scenario like the one in Fig. 3 now. At this stage, just notice that the letters of the underlying grid are those in the boxes (X,U,V,...), while the neighboring areas are used to put extra information.

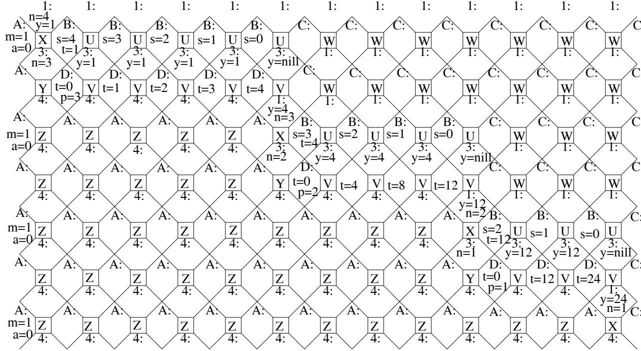


Figure 3: 1st scenario for `MAFact` (see pag.16, too)

2 Finite interactive systems

2.1 Finite interactive systems

Intuition A finite interactive system may be seen as a kind of two-dimensional automaton mixing a state-transforming automaton with an automaton used for the interaction of the processes created by the first automaton. (Alternatively, one may consider a finite interactive system to be a kind of iterate transducer). Syntactically, it may be described by a graph as in Fig. 4.

While the above presentation may be somehow useful, it is also misleading. Actually, not two different automata are combined, but these two views are melted together to create the concept of finite interactive systems. For instance, there are two FISs with the same projection automata, but different grid languages - see Prop. 2.1.

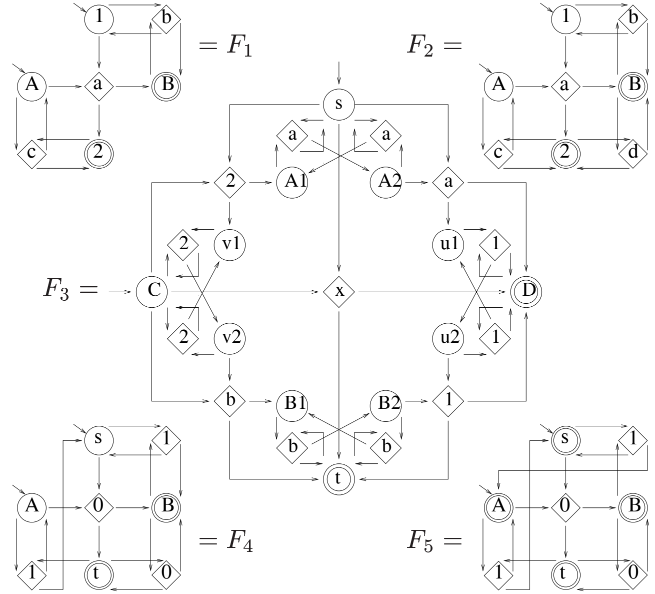


Figure 4: Examples of finite interactive systems

Definition A *finite interactive system* (FIS) is a finite hyper-graph with two types of vertices and one type of (hyper) edges:

— a first type of vertices is for *states*; we label them using numbers or lower case letters;

— the second type of vertices is for *classes*; we use capital letters as labels;

— the edges (also called *transitions*) are labeled by letters denoting atoms of the grids; they obey the following constraints: (1) each transition has two incoming arrows: one from a class vertex, the other from a state vertex, and (2) each transition has two outgoing arrows: one to a class vertex, the other to a state vertex.

Some classes/states may be *initial* (in the graphical representation this is specified by small incoming arrows) or *final* (the double circle representation is used).

One may use a semi-textual representation for FISs (e.g., F_1 in Fig. 4 is defined by: $A, 1$ initial; $B, 2$ final; and transitions $\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \hline A & a & B \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$, $\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \hline A & c & A \\ \hline \multicolumn{1}{c|}{} & 2 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$, $\begin{array}{|c|c|c|} \cline{2-2} \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \hline B & b & B \\ \hline \multicolumn{1}{c|}{} & 1 & \multicolumn{1}{c}{} \\ \cline{2-2} \end{array}$) or a fully-textual one (e.g., `a:(A,1)->(B,2)` or $(A,1) \xrightarrow{a} (B,2)$, or even shorter, `a:A1->B2`, if no confusion arises).

Parsing procedure Given a FIS F and a grid w, insert[6] at the north (resp. west) border of w initial states (resp. classes) and parse the grid selecting minimal unprocessed atoms — "minimality" is that used in the definition of the flattening operator. For each

```
(a) abbb    (b) 100    (c) 010101    (d) 2aaaa
    cabb        010        100110        22aa1
    ccab        110        000100        22x11
    ccca        001        111000        2bb11
                101                      bbbb1
                011
                111
```

1. (square grids) Both, F_1 and F_2 recognizes square grids, see (a).
2. (exponential function) F_4 recognizes $\{0, 1\}$ grids[7] with rows representing numbers 1 to $2^n - 1$, see (b).
3. (Pascal triangle) F_5 recognizes $\{0, 1\}$ grids with alternating 0/1 sequences on north and west borders and satisfying the recurrence rule of Pascal triangle modulo 2 along the secondary diagonals, see (c).
4. (spiral grids) F_5 recognizes 'spiral' grids, see (d) - to get real spirals, replace $a, 1, b, 2$ by $\rightarrow, \downarrow, \leftarrow, \uparrow$.

Table 2: The languages of the FISs in Fig. 4

such atom a, if s (resp. C) is its north state (resp. west class), then choose a transition $C \,\underset{s'}{\overset{s}{a}}\, C'$ from F (if any), insert s' (resp. C') at its south (resp. east) border and consider this atom to be already processed. Repeat the above as long as possible.

The grid w is *recognized* if there is a parsing such that all of its atoms are processed and the south (resp. east) border contains final states (resp. classes), only.

The *language* $L(F)$ is the set of grids recognized by F.

As an example, use the grid

```
abb
cab
cca
```

and F_1 (in Fig. 4); a parsing is

```
 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
```

showing this grid is recognized by F_1. (Actually, F_1 is deterministic, so the parsing is unique.) A similar parsing shows that

```
abb
cab
```

is not recognized.

Table 2 describes the languages of the FISs in Fig. 4.

2.2 Decomposable FISs

FISs vs. finite automata There is a natural way to associate finite automata (NFAs) to FISs: by projection. The question[8] here is whether a FIS may be reduced to its projections or not.

The *state projection* NFA associated to a FIS F, denoted $s(F)$, is the NFA obtained neglecting its class part. Similarly, the *class projection* NFA, denoted $c(F)$, is the NFA obtained neglecting the state part of F.

Let F be a FIS defined by transitions $(A, 1) \xrightarrow{a} (B, 2)$, $(A, 1) \xrightarrow{a} (C, 3)$ with $A, 1$ initial and $B, 3$ final. Its state projection $s(F)$ is defined by transitions $1 \xrightarrow{a} 2$, $1 \xrightarrow{a} 3$, with 1 initial and 3 final. Its class projection $c(F)$ is defined by transitions $A \xrightarrow{a} B$, $A \xrightarrow{a} C$, with A initial and B final. Notice that $L(F) = \emptyset$.

The slightly modified FIS F', defined by $(A, 1) \xrightarrow{a} (B, 3)$, $(A, 1) \xrightarrow{a} (C, 2)$ with $A, 1$ initial and $B, 3$ final, has the same projections, but $L(F') = \{a\}$. Hence,

Proposition 2.1 *There exist two* FIS*s F and F' such that: $s(F) = s(F') \wedge c(F) = c(F')$, but $L(F) \neq L(F')$.*

Decomposable FISs The above result shows FISs can not be reduced to finite automata — they are more complex than a simple combination of their projections. However, there is a natural subclass of FISs for which such a reduction is possible.

A *decomposable* FIS is a FIS obtained from two automata by the following procedure: Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be NFAs. We build a FIS $\phi(\mathcal{A}_1, \mathcal{A}_2)$ as follows:

— the states of $\phi(\mathcal{A}_1, \mathcal{A}_2)$ are the states of $\mathcal{A}_1$;
— the classes of $\phi(\mathcal{A}_1, \mathcal{A}_2)$ are the states of $\mathcal{A}_2$;
— a transition $a : (A, s) \rightarrow (B, t)$ in $\phi(\mathcal{A}_1, \mathcal{A}_2)$ corresponds to a pair of transitions $s \xrightarrow{a} t$ in $\mathcal{A}_1$ and $A \xrightarrow{a} B$ in $\mathcal{A}_2$;
— a state (resp. class) is initial or final if it has that property in $\mathcal{A}_1$ (resp. $\mathcal{A}_2$)

The *decomposable closure* of a FIS F is $\delta(F) = \phi(s(F), c(F))$, i.e., the decomposable FIS generated by its projections.

Examples: F_1 is decomposable; if we rename its actions b, c as a, then we get an indecomposabile FIS F'_1 recognizing squares of a's; the decomposable cover of F'_1 is a FIS recognizing rectangles of a's. It is clear that,

Fact 2.2 — *(1) A decomposable* FIS *is uniquely determined by its class and state projections.*
— *(2) A* FIS *is decomposable if all transitions are labeled with distinct letters.*
— *(3) Language equivalence for decomposable* FIS*s is decidable.*

2.3 Towards an algebraic theory

A representation theorem For F_1 (in Fig. 4), the class projection $c(F_1)$ is defined by: A initial, B final, and transitions $A \xrightarrow{a} B$, $A \xrightarrow{c} A$, $B \xrightarrow{b} B$. Its language is $c^{\dagger} \triangleright a \triangleright b^{\dagger}$. Similarly, $s(F_1)$ is defined by: 1 initial, 2 final, and transitions $1 \xrightarrow{a} 2$, $2 \xrightarrow{c} 2$, $1 \xrightarrow{b} 1$. Its language is $b^{\star} \cdot a \cdot c^{\star}$.

F_1 has an interesting property: $L(F_1) = (b^\star \cdot a \cdot c^\star)^\dagger \cap (c^\dagger \triangleright a \triangleright b^\dagger)^\star = L(s(F_1))^\dagger \cap L(c(F_1))^\star$. However, for F_5 such a property does not hold: $L(s(F_5)) = \{0,1\}^\star$ and $L(c(F_5)) = \{0,1\}^\dagger$, so $L(s(F_5))^\dagger \cap L(c(F_5))^\star = \{0,1\}^{\star\dagger} \neq L(F_5)$. Actually,

Proposition 2.3 *If F is a decomposable* FIS*, then*[9] *$L(F) = L(s(F))^\dagger \cap L(c(F))^\star$; particularly, this is true when all transitions in F are labeled by distinct letters.*

We may rename the transitions of a FIS to be all distinct, apply Prop. 2.3, then use a letter-to-letter homomorphism to re-obtain the original label. Thus,

Corollary 2.4 *Basic 2-dimensional regular expressions with letter-to-letter homomorphisms are equivalent to* FIS*s.*

FIS transformations Regular languages have a clean algebraic theory based on finite automata and their simulation via relations, see [28]. A key property is: Two automata are equivalent (recognize the same language) iff there is a chain of simulations via relations connecting them.

A NFA $\mathcal{A}$ with n states may be represented by a matrix $\begin{bmatrix} 0 & B \\ C & D \end{bmatrix}$, where: B is a 0/1 row (b_i) with $b_i = 1$ iff i-th state is initial; C is a 0/1 column (c_i) with $c_i = 1$ iff i-th state is final; D is a $n \times n$ matrix (d_{ij}), where d_{ij} is the sum of the letters having a transition from i-th to j-th state. Let $\mathcal{A} = \begin{bmatrix} 0 & B \\ C & D \end{bmatrix}$ and $\mathcal{A}' = \begin{bmatrix} 0 & B' \\ C' & D' \end{bmatrix}$ be two NFAs; they are *similar* via a relation ρ between their states if $\begin{bmatrix} 0 & B \\ C & D \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \rho \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \rho \end{bmatrix} \begin{bmatrix} 0 & B' \\ C' & D' \end{bmatrix}$.

This simulation may be naturally lifted to decomposable FISs: Two decomposable FISs F, F' are *similar* via a pair of relations $\rho = (\rho_s, \rho_c)$ if $s(F), s(F')$ are similar via ρ_s and $c(F), c(F')$ are similar via ρ_c.

Proposition 2.5 *Simulation via relations is a correct and complete transformation for decomposable* FIS*s.*

This result opens the way towards an axiomatization of the grid langauges represented by decomposable FISs.

While the above definition of simulation makes sense for all FISs, it does not always provide a correct transformation - use Prop. 2.1. Automata simulation is a translation of the standard network algebra simulation by passing from relations on *actions* to relations on *states* — probably, for general FISs one has to return to the original network algebra simulation to get a correct transformation.

3 Spatio-temporal specifications

3.1 Data with temporal representation

Turing tape was a first spatial[10] memory model used in machine-oriented models of computation; complex data structures may be implemented on top of this simple model. We propose[11] to use a similar approach to create complex data structures in time.

Voices Registers holding numbers may be implemented on a Turing tape. At their higher level, the tedious aspects of identifying the positions of the numbers on the tape, the need to shift data when more space is needed, the computation of arithmetical or logical operations at the bit level, etc. are all hidden and one can get a more readable specification of the problem and of its solution.

Similarly, we start with a simple linear temporal data model - the *stream* structure. A stream, as used in the dataflow setting, is a finite or infinite sequence of data ordered in time. A stream is denoted as $a_0 {}^\frown a_1 {}^\frown a_2 {}^\frown \ldots$, where $a_0, a_1, a_2, \ldots$ are its data (tokens) at time $0, 1, 2, \ldots$, respectively.

The contents of our streams will be always finite, but unbounded in time, in the same way the contents of a Turing tape is always finite, but unbounded in space. One may think of a stream as the result of observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle.

A *voice* is a temporal structure that holds numbers.

Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape. For instance, voices may be specified giving their starting addresses and their lengths: e.g., a voice v holding 2004 is defined by its temporal address t_0 (its starting time on the stream) and its length 4 (if decimal representation is used) - this means, from that point in time t_0, the cell corresponding to the stream is showing the digits 2,0,0,4 during 4 consecutive clock cycles.

At this new and higher level of abstraction we are interested in voices and their contents only. We are not interested in the implementation details as: representation (*continuous* time slots: 1st voice, 2nd voice,...; or *alternating* digits: 1st digit of 1st voice,..., 1st digit of last voice, 2nd digit of 1st voice,...; etc.), position on the stream, low-level manipulation, etc.

More data with temporal representation The above setting is good for theoretical purposes (a voice may represent an arbitrary long number), but in practice more concrete data scructures are needed. Most

of usual data strucures have natural temporal representations — we add a "t" in front of normal types to denote these new temporal types. Examples: `tBool` (booleans), `tInt` (integers, of various lengths) , `tArray` (arrays), `tLinkedList` (linked lists), etc.

3.2 Specifications

Relational spatio-temporal specifications A few conventions: To distinguished between the products of data with spatial or temporal representation, we use the notation '$\otimes$' for the spatial product and '$\frown$' for the temporal one (mathematically, they are just Cartesian product). Moreover, $\mathbb{N}^{\otimes k}$ denotes $\mathbb{N}^{\otimes} \ldots ^{\otimes} \mathbb{N}$ (k terms) and $\mathbb{N}^{\frown k}$ denotes $\mathbb{N}^{\frown} \ldots ^{\frown} \mathbb{N}$ (k terms).

A *spatio-temporal specification* is a relation

$$S \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$$

between input and output registers and voices. It is denoted as $S : (m,p) \to (n,q)$, where m (resp. p) is the number of input voices (resp. input registers) and n (resp. q) is the number of output voices (resp. output registers). On elements, it is defined as a relation between concrete tuples, written as $\langle v \mid r\rangle \mapsto \langle v' \mid r'\rangle$, where v, v' (resp. r, r') are tuples of voices (resp. registers).

Examples The constants used in Fig. 2

$c0 =$ □, $c1 =$ □, $c2 =$ □, $c3 =$ □, $c4 =$ □, $c5 =$ □

have a natural relational interpretation:
$c0 = \emptyset$; $c1 = \{\langle \mid x\rangle \mapsto \langle \mid x\rangle : x \in \mathbb{N}\}$;
$c2 = \{\langle x \mid \rangle \mapsto \langle x \mid \rangle : x \in \mathbb{N}\}$;
$c3 = \{\langle \mid x\rangle \mapsto \langle x \mid \rangle : x \in \mathbb{N}\}$ (space-to-time converter);
$c4 = \{\langle x \mid \rangle \mapsto \langle \mid x\rangle : x \in \mathbb{N}\}$ (time-to-space converter);
$c5 = \{\langle x \mid y\rangle \mapsto \langle x \mid y\rangle : x, y \in \mathbb{N}\}$.

Composing specifications Specifications may be composed horizontally and vertically, as long as their types agree. For two specifications $S_1 : (m_1, p_1) \to (n_1, q_1)$ and $S_2 : (m_2, p_2) \to (n_2, q_2)$:
— the *horizontal composition* $S_1 \triangleright S_2$ is defined only if $n_1 = m_2$; the type of $S_1 \triangleright S_2$ is $(m_1, p_1 + p_2) \to (n_2, q_1 + q_2)$; the composite is defined as expected:

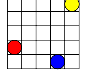
$$\langle v \mid r_1, r_2\rangle \mapsto \langle v'' \mid r_1', r_2'\rangle \text{ in } S_1 \triangleright S_2 \text{ iff}$$
$$\exists v'.\langle v \mid r_1\rangle \mapsto \langle v' \mid r_1'\rangle \text{ in } S_1 \wedge \langle v' \mid r_2\rangle \mapsto \langle v'' \mid r_2'\rangle \text{ in } S_2$$

— the *vertical composition* $S_1 \cdot S_2$ is defined in a similar way - it is defined only if $q_1 = p_2$; its resulting type is $(m_1 + m_2, p_1) \to (n_1 + n_2, q_2)$.

Let us make a comment on the type of composed specifications. It may look a bit strange to see that the type of composed specification is expanded and one gets bigger and bigger types. However, this is natural when one tries to get a modular specification and focuses on specifying *parts* of the system - the whole system may still have a reasonable small input-output type.

For instance, in an OO system one has to specify input data for all objects used in a run of a program - hence, specifications have these expanded types. Such a tedious task is avoided by: (1) using constructors providing initial data when objects are created and (2) collecting the final results in the main program. In other words, except for the main program, the overall type of an object is $(m, 0) \to (n, 0)$. Repeated horizontal composition of objects does not change the spatial type, which is reduced to the type of the main program.

3.3 Specifying a simple OO system



Object-orientation is a very rich field and it may be virtually impossible to touch even its most basic aspects in a brief subsection. The aim here is very modest: we show how a simple OO-system may be specified, how it may be decomposed, and discuss about the usefulness of temporal types.

Colored balls The system under construction consists of colored balls (points) in a rectangular 2-dimensional area. A ball may be moved up-down and left-right using two buttons - one for vertical, the other for horizontal movement. The color of a ball is changed to blue/yellow/red when it touches (north-or-south border)/a corner/(west-or-east border). For simplicity, we use one ball and a 5×5 rectangle.

In the design of such a system we use: three fields x, y, z (for X-coordinate, Y-coordinate, and color; for colors, use 9/0/1 to represent blue/yellow/red); two methods: $h(d)$ (horizontal movement; d is 9/0/1 for move-one-cell-left/stay/move-one-cell-right); $v(d)$ (vertical movement; d is 9/0/1 for move-one-cell-down/stay/move-one-cell-up); an attribute c (it returns the color z).

Before working on the specification, we have to fix an interactive behavior of the system: Suppose the input interaction is via horizontal-vertical movement buttons, while the output is just a display of the ball color.

Global specification The specification uses a register for each field and a voice for each method or attribute (without loss of generality, we suppose a voice is able to represent the parameters/values of its associated method/attribute as a digit - just use a large enough set of digits[12]). The type of this global specification is $S : (3,3) \to (3,3)$, so we have to specify a set of tuples $\langle h, v, c \mid x, y, z\rangle \mapsto \langle h', v', c' \mid x', y', z'\rangle$.

An example is

$$(1)\ \langle h_1, v_1, c_1 \mid 3,3,0\rangle \mapsto \langle h_1', v_1', c_1' \mid 3,3,9\rangle$$

where $h_1 = 11091990, v_1 = 90199911, c_1' = 01110999$ (c_1, h_1', v_1' are irrelevant). Each position in h and v describes the status of the input buttons at a given clock cycle; they are independent, so any combination is possible; if a^i denotes the i-th digit of a voice a, then at the i-th clock cycle the system passes from its state (x, y, z) to a new state (x', y', z') by actions h_1^i, v_1^i, written $x, y, z \xrightarrow{h_1^i, v_1^i} x', y', z'$; the system evolution is

$$3,3,0 \xrightarrow{1,9} 4,2,0 \xrightarrow{1,0} 5,2,1 \xrightarrow{0,1} 5,3,1 \xrightarrow{9,9} 4,2,1 \xrightarrow{1,9} 5,1,0 \xrightarrow{9,9} 4,1,9 \xrightarrow{9,1} 3,2,9 \xrightarrow{0,1} 3,3,9$$

This sequence gives the final state and it is also used to compute the voice c' of the output colors.

Horizontally (de)composed specification The above specification may be horizontally decomposed[13] and described in a modular way: Use a class for one-dimensional points in a bounded interval; these points may be moved in both directions, reporting when the border is touched — let *hit* be a voice used to record these events. Take two such points (one for horizontal, the other for vertical direction) and "compose" them the get the original system — the link is realized using an additonal object which receives the *hit* information to generate the final color. Formally, take

$$S_1 = S_h \triangleright S_v \triangleright S_c$$

where

$S_h : (5,1) \to (5,1)$ is defined by
$$\langle h, v, c, hith, hitv \mid x\rangle \mapsto \langle h, v, c, hith', hitv \mid x'\rangle$$
$S_v : (5,1) \to (5,1)$ is defined by
$$\langle h, v, c, hith, hitv \mid y\rangle \mapsto \langle h, v, c, hith, hitv' \mid y'\rangle$$
$S_c : (5,1) \to (5,1)$ is defined by
$$\langle h, v, c, hith, hitv \mid z\rangle \mapsto \langle h, v, c', hith, hitv \mid z'\rangle$$

for appropriate mapping describing the primed variables (i.e., $hith', x', \ldots$) in terms of the inputs. S is obtained restricting S_1 to its first three voices h, v, c.

To have an example, case (1) above is decomposed as follows (0/1 is the code for non-hit/hit event):
—with h_1 in S_h one gets $hith_1' = 01101000$ (and $x_1' = 3$)
—with v_1 in S_v, one gets $hitv_1' = 00001100$ (and $y_1' = 3$)
—with $hith_1'$ and $hitv_1'$ in S_c, one gets $c' = 01110999$ (and $z_1' = 9$)

Usefulness of temporal types The above specification is based on a global typing for the temporal data, i.e., all components S_h, S_v, S_c are allowed to see (or "hear") all voices. This peculiarity may be used to describe specifications where the components cheat in their interaction.

In the above decomposition S_h may cheat: while not his business, he may change the content of v, preventing component S_v of having a right reaction. For instance, using (the assumption that initially $y = 3$ and) $\langle h, v, c, hith, hitv \mid x\rangle \mapsto \langle h, v', c, hith', hitv \mid x'\rangle$ with $v' = 91919191$ for any input v, S_h fakes the meaning of the vertical button, preventing S_v of touching the borders - now, S_v keeps moving back and forth.

A simple solution to avoid such abnormal behavior is to use more restricted temporal types. From the analysis of the system, we may provide a more restricted typing. The initial specification has the restricted type $S : (2,3) \to (1,3)$ mapping $\langle h, v \mid x, y, z\rangle \mapsto \langle c \mid x', y', z'\rangle$. It may be decomposed as

$$S = (S_h {}^\frown Id) \triangleright (Id {}^\frown S_v) \triangleright S_c$$

where Id denotes appropriate identities and S_h, S_v, S_c are (with appropriate mappings for $hith, x', \ldots$)

$S_h : (1,1) \to (1,1)$, defined by $\langle h \mid x\rangle \mapsto \langle hith \mid x'\rangle$
$S_v : (1,1) \to (1,1)$, defined by $\langle v \mid y\rangle \mapsto \langle hitv \mid y'\rangle$
$S_c : (2,1) \to (1,1)$, defined by $\langle hith, hitv \mid z\rangle \mapsto \langle c' \mid z'\rangle$

Notice that $S_h {}^\frown Id : \langle h, v \mid x\rangle \mapsto \langle hith, v \mid x'\rangle$ and $Id {}^\frown S_v : \langle hith, v \mid y\rangle \mapsto \langle hith, hitv \mid y'\rangle$, so the type of the right-hand side is $\langle h, v \mid x, y, z\rangle \mapsto \langle c \mid x', y', z'\rangle$, equal to the type of S.

With such a detailed decomposition and restricted typing mechanism, a cheating as above is no more possible.

This OO-system specification is flat, i.e., only one level of the interaction is considered. If we look to a scenario (for FISs or rv-programs), we see that during their lifetime two processes (columns) may interact via different classes, each with its particular temporal type. Let us emphasize this interesting observation: *Temporal typing is changing in time; a concrete temporal type for process interaction lasts for a limited time period.*

3.4 Specifying an interactive game

1	0	
	0	1
	0	

This subsection describes a spatio-temporal specification of a simple interactive game. An additional aim of the analysis is to get a better understanding of the temporal aspects of these specifications looking to "local vs. global time" dilemma.

01-game The game uses an $n \times n$ table and 2 players, one using '1', the other '0'. The players alternate placing their symbols on the table (one symbol at a time). To win the game, a player has to place 3 consecutive symbols on the same line, column, or diagonal. To keep the specification simple, we restrict ourself to the trivial case $n = 3$. The cells are identified by numbers 1 to 9 (counting: left-to-right, then top-to-bottom).

The specification The specification $S : (2,1) \to (2,1)$ is defined by $\langle p0, p1 \mid t\rangle \mapsto \langle a0, a1 \mid t\rangle$ where
— t is a register to record the status of the table - i-th digit of t describes the content of i-th cell, namely: 9 (empty cell); 0 (cell holding 0); 1 (cell holding 1).
— $p0, p1$ are input voices to record players' actions and $a0, a1$ are output voices to score the results. In $p0, p1$ the code is: 0 (no action); $k \in \{1, \ldots, 9\}$ (place a symbol on k-th cell). In $a0, a1$ the code is: 9 (wrong-move/loss); 0 (good move); 1 (win).

A few examples are below (star means 'don't care'[14]): (a) describes a complete play ending with a tie; (b) describes a better strategy for player 0, which wins; in (c), player 1 does not provide a move when it is his turn, so player 0 wins; (d) describes a case where both players move at the same time and player 2 loses (it was not his turn).

(a) $\langle 503040801, 060207090 \mid 999999999\rangle \mapsto$
$\langle 000000000, 000000000 \mid 010001101\rangle$
(b) $\langle 50208{*}{*}{*}{*}, 06010{*}{*}{*}{*} \mid 999999999\rangle \mapsto$
$\langle 00001{*}{*}{*}{*}, 00009{*}{*}{*}{*} \mid 10{*}{*}01{*}0{*}\rangle$
(c) $\langle 5030{*}{*}{*}{*}{*}, 0600{*}{*}{*}{*}{*} \mid 999999999\rangle \mapsto$
$\langle 0001{*}{*}{*}{*}{*}, 0009{*}{*}{*}{*}{*} \mid {*}{*}0{*}01{*}{*}{*}\rangle$
(d) $\langle 503{*}{*}{*}{*}{*}{*}, 061{*}{*}{*}{*}{*}{*} \mid 999999999\rangle \mapsto$
$\langle 001{*}{*}{*}{*}{*}{*}, 009{*}{*}{*}{*}{*}{*} \mid {*}{*}0{*}01{*}{*}{*}\rangle$

Above, we included a few cases to illustrate the method. It is clear that a full specification of the game may be described in this formalism.

The specification language is quite powerful: one may specify a situation where both players move at the some time, or one player misses his move, etc. So, let us concentrate on a few aspects related to a possible lower level implementation (schedule) to see if such high-level specifications are natural or not.

A lesson from register machines A spatio-temporal specification describes a relational transformation between the input registers and voices and their output counterparts. Voices are specified in time - at a lower level we have to schedule them on streams. For 01-game one may find an arrangement of their actions in time to fit the intuition: for (503040801, 060207090) use $5\frown 0\frown 0\frown 6\frown 3\frown 0\frown 0\frown 2\frown 4\frown 0\frown 0\frown 7\frown 8\frown 0\frown 0\frown 9\frown 1\frown 0$.

However, notice that the input voices will be scheduled on the input stream, while the output ones on the output stream. In a real situation, it is supposed that the players will see the results *before* giving a next move. So, it should be a relation between the time used on the input stream and the time used on the output stream.

Let us recall the situation with register machines. In that case, the custom is to clearly differentiate between input and output registers, for instance using different variables. As they are different, when we implement them on a tape there is no constraint to have a specified relation between their positions on the tape.

To have modular high-level spatio-temporal specifications, perhaps we have to follow a similar rule by not relating the physical time on the output stream to the physical time on the input stream. This argument may not be fully convincing. However, there is a stronger argument leading to a similar conclusion - see below.

The blind player As we said, in real situations it is supposed that the players will *see* the results *before* giving a next move. Now, looking to another part of the statement, we see that we actually introduce assumptions on the players - on their sight capabilities. However, I do not want to mix my relatively easy task of specifying a simple game with the daunting task of specifying a human player!

Here we may apply a reasoning somehow similar to that used in Turing test[15]: Suppose we have a blind player which "by pure chance" choose to play the same moves as a player with sound sight capabilities. From the point of view of the game, only the moves matter, so there is no difference between having a blind or a sound player, as long as they make the same moves. Once we have reached this point, the next step of the argument is easy to guess: for the blind player, there is really no difference on whether the output of a move is shown before the next move or not.

Relaxing time constraints To conclude, in our spatio-temporal specifications it is desirable to weaken the assumption of having a clear and tight relationship between the physical time on the input stream and the physical time on the output stream. Then,

Proposition 3.1 *With a local time view, if enough memory space is provided, then any computable relation between input and output voices may be implemented.*

Indeed, if enough memory space is provided, then one may record the contents of all voices, then do the transformation within a conventional state-transforming model, and finally deliver the results in time.

The assumption of having enough memory may be practically violated. Then, at a lower implementation level we have to refine the specification and to come up with a detailed program where the physical timing is essential. But this refinement will come later: it is not part of the starting abstract specification, it will be the very heart part of the implementation.

4 Rv-systems and rv-programs

An *rv-system* (*interactive system with registers and voices*) is a FIS enriched with: (i) registers associated to its states and voices associated to its classes; and (ii) appropriate spatio-temporal transformations for letters. A computation is described by a scenario like in a FIS, but with concrete data around each letter. In the following we focus on the class of programmable rv-systems, more precisely those described by rv-programs.

4.1 Rv-programs

Rv-programs - a minimal instruction set Register programs (see, e.g., [13]) can compute all partial computable functions using a small set of instructions, e.g. **add 1** to a register, **decrease 1** from a register, and **test if** a register **is 0**; moreover, the last two instructions may be combined, reducing the number of instruction types to 2.

One may introduce rv-programs using a similar small number of instructions. An *rv-program* (*program for interactive systems with registers and voices*) uses a finite set of registers and voices; it consists of a list of statements of the following type

```
(A,a):  x = x+1 goto [B,b];   or
(A,a):  if x == 0 goto [B,b]
                   else x = x-1 goto [C,c];
```

where `x` is a register or a voice and `A,B,C,a,b,c` are labels. As this is part of a much expanded setting described below, we skip the details.

Rv-programs - a more practical setting The running example is more or less random: as factorial function is a popular example used to illustrate various programming concepts, we are using here a kind of interactive factorial function.

The program, denoted `MAFact`, starts with an input `n` (of type `sInt` - spatial integer) and computes

$$f(\mathtt{n}) = ((\ldots((1 *_1 \mathtt{n}) *_2 (\mathtt{n}-1)) *_3 \ldots *_{\mathtt{n}-1} 2) *_{\mathtt{n}} 1)$$

The operation to be performed at a step $*_i$ is either multiplication or addition and it is known by reading the values of the interaction variables `m,a` (of type `tBool` - temporal booleans) at stage i: if `m` is true, then multiplication is chosen, otherwise addition (provided `a` is true). Moreover, multiplication is implemented using additions (to illustrate how computation may be extended in space). Its actual code is shown in Fig. 5.

Syntax The syntax is based on the syntax used in imperative programming languages. The basic block is

```
in: A,1; out: C,4
X::
  (A,1)  | n,y : sInt
---------+------------------------------------------
  a,m :  | s,t : tInt;
  tBool  | if(m == 1){
         |    if(n == 1){
         |       goto [C,4];
         |    }else{
         |       s = n; t = y; n = n-1; goto [B,3];
         |    }
         | } elseif(a == 1){
         |    y = y+n; n = n-1; goto [C,1];
         | }
Y::
  (A,3)  | n : sInt
---------+------------------------------------------
         | t,p : tInt;
         | t = 0; p = n; goto [D,4];
Z::
  (A,4)  |
---------+------------------------------------------
         | goto [A,4];
U::
  (B,1)  |
---------+------------------------------------------
  s,t :  | y : sInt;
  tInt   | if(s == 0){
         |    y = nill; goto [C,3];
         | }else{
         |    y = t; s = s-1; goto [B,3];
         | }
V::
  (D,3)  | y : sInt
---------+------------------------------------------
  t,p :  | n : sInt;
  tInt   | if(y != nill){
         |    t = t+y; goto [D,4];
         | }else{
         |    y = t; n = p; goto [C;1];
         | }
W::
  (C,1)  |
---------+------------------------------------------
         | goto [C,1];
```

Figure 5: `MAFact` program

a module. To explain the syntax, let us focus on the first module of `MAFact`. It has a name `X` and 4 areas.

(1) In the top-left part we have a pair of labels `(A,1)` which specifies the interaction and control coordinates where this module has to be applied. Notice that similar pairs of labels are used inside the code (described in the bottom-right area), but the format `[B,3]` and the meaning are different. We will return to this later.

(2) The top-right part declares the spatial input variables. These variables specify the memory state before the application of the module. To distinguish them from the variables used for interaction interfaces, we put an "`s`" in front of their types. For module `X` there are two spatial input variables of type integer, denoted `sInt`.

(3) The bottom-left part is similar to the top-right one, but it declares the temporal input variables. These variables are used for the data appearing to the interaction interfaces between modules. We put a "`t`" in front of

normal types to indicate these data have temporal representations. Module `X` has two temporal input variables of type boolean, denoted `tBool`.

(4) The body of a module is its bottom-right part. It may include type declarations for new variables and the code to be executed by the module, similar to C code. The computation within this part makes no distinction between a spatial and a temporal variable. The exit from the module is realized by a `goto` statement. The computation within an interactive systems has two dimensions - one vertical, the other horizontal. A statement like `goto [B,3]` indicates that:
— The data of the spatial variables in the current module will be used in a next module with control state `3`. (What will be the new interaction data is not known from this module alone - it depends on the global properties of the interactive system.)
— The data of the temporal variables in the current module will be used for the interaction interface of a new module with interaction label `B`. (Again, it is not known from this module alone what will be the control state of the new process when it receives these data at its temporal interface.)

4.2 Semantics

Operational semantics The operational semantics of rv-systems is given in terms of scenarios. For `MAFact`, a first example of such a scenario is illustrated in Fig. 3 - it corresponds to a particular case where the (spatial) input variable `n` is `4` and the interaction interface `A` keeps choosing `m` to be true, hence the result is $(((1 \times 4) \times 3) \times 2) \times 1 = 24$. A second scenario is illustrated in Fig. 6 - this latter case is like the former, but in the 3rd round `m` is false and `a` is true, hence the result is $(((1 \times 4) \times 3) + 2) \times 1 = 14$.

Figure 6: 2nd scenario for `MAFact` (see pag.16, too)

Scenarios are constructed using the following procedure (see, e.g, the scenario in Fig. 6):

(1) Each cell of the underlaying grid has a label in `{X,Y,Z,U,V,W}` specifying the module of the program used in that particular cell. A cell has associated states on its top and bottom neighboring areas and associated classes on its left and right neighboring areas. The grid is constructed starting with its top row and left column (containing the inputs) and progressively inserting new cells when their left and top neighboring areas are already computed/updated.

(2) An area may have an additional information as `s=2` - that means, in that area `s` has been updated to be `2`.

(3) The full information on the current state of a process is obtained going vertically up and collecting the last updated values of the spatial variables. For instance, the bottom `Z` in the first column has `n=3,y=1`.

(4) The full information on temporal variables in a current place is obtained collecting their last updated values going horizontally on left. For instance, `X` in the 6-th column has `m=1,a=0`.

(5) The 1st column has an input class (here `A`) and particular tuples of values for its temporal variables. The 1st row has an input state (here `1`) and particular tuples of values for its spatial variables.

(6) Once the above points were clarified, the local computation of a cell α is easy to describe:
—(i) Take a module β of the program bearing the class label of the left neighboring area of α and the state label of the top neighboring area of α.
—(ii) Follow the code in β using the spatial and temporal variables of α with their current values.
—(iii) If the local execution of β is finished with a `goto` $[\Gamma, \gamma]$ statement, then the label of the right neighboring area of α is set to Γ and the label of the bottom neighboring area of α is set to γ.
—(iv) Insert in the right neighboring area of α the values of the temporal variables updated by β and in the bottom neighboring area of α the values of the spatial variables updated by β.

(7) A *partial scenario* (for an rv-program) is a scenario constructed using the above rules; it is a *complete scenario* if the bottom row has only final states and the rightmost column has only final classes.

Both scenarios in Fig. 3 and Fig. 6 are complete scenarios for `MAFact` program.

Input-output denotation The operational semantics may be used to define input-output denotation of rv-programs. Given its restricted syntax[16], each module has a clear input-output denotation, i.e., it specifies a relation $R \subseteq (\mathbb{N}^{\frown m} \times \mathbb{N}^{\otimes p}) \times (\mathbb{N}^{\frown n} \times \mathbb{N}^{\otimes q})$ between input registers and voices and their output counterparts.

These denotations are lifted to the scenario level using the composition operators on relations defined in Sec. 3.2. The *input-output denotation* of an rv-program is the result produced by the above procedure.

The type of these denotations is not unique; e.g., in `MAFact` the temporal interface needs more or less data depending on the value of `n`. Actually, `MAFact` is more like an internal piece of code, rather than an independent program with a clear input-output interface.

New in-out type: in: E,5,n,mt[n],at[n]; out: F,C,4,y
Change in module `X`*: add* write y *before* goto [C,4];
Add new modules:

X1::

(E,5)	n : sInt

mt[n], at[n]: tArray of tBool	i : sInt, ms[n],as[n] : sArray of sBool;
	read n;
	listen mt[1..n], at[1..n];
	ms[1..n] = mt[1..n]; as[1..n] = at[1..n];
	i = 1; goto [F,6];

X2::

(E,6)	n,i : sInt, ms[n],as[n] : sArray of sBool

	m,a : tBool;
	m = ms[i]; a = as[i]; i = i+1;
	if (i > n) then goto [A,4] else goto [A,6];

X3::

(F,5)	
p : tInt	n : sInt; n = p; goto [F,1];

Figure 7: `MAFact` with input-output

Input-otput operations are specified by **read/write** instructions for spatial data and **listen/speak** instructions for temporal data.

The changes needed for `MAFact` to accomodate input-output instructions are described in Fig. 7. To shrink the temporal interface we collect all booleans `m,a` in boolean arrays[17] and listen them at the top-left cell of scenarios. Then, internally the program supplies the temporal data needed by the previous version of `MAFact` program at its `A` interface.

The input output transformation corresponding to the modified `MAFact` program is

$$\langle \mathtt{m}[1..\mathtt{n}], \mathtt{a}[1..\mathtt{n}] \mid \mathtt{n} \rangle \mapsto \langle \mid ((\ldots((1 *_1 \mathtt{n}) *_2 (\mathtt{n}-1)) *_3 \ldots *_{\mathtt{n}-1} 2) *_{\mathtt{n}} 1) \rangle$$

where $*_i$ is '$\times$' if $\mathtt{m}(i) = 1$ and '$+$' if $\mathtt{m}(i) = 0 \wedge \mathtt{a}(i) = 1$.

4.3 Space-to-time/time-to-space converters

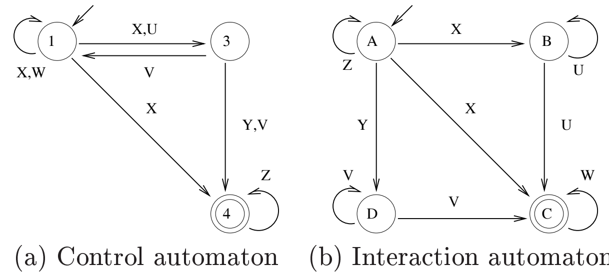
The information may be converted from a spatial to a temporal representation and vice-versa. An example is present in Fig. 6: the value of the spatial variable `n` in the 1st column is copied in a temporal variable `p`, communicated to the 6th column, and converted back to the spatial variable `n`. This is an instance of a more general procedure where: one may copy the memory state of a process (or a part of it), communicate it using temporal variables to a different process, and reuse there to (re)set the memory state.

Let us return to Fig. 2 for a deeper understanding of modeling a two-ways communication. We described the relational semantics of the involved constants (identities, corners, etc) in Sec. 3.2. The full state of Process `a` in the 1st column is transmitted in time to the 4th column, then `a` terminates. In the 4th column the information is received in time and Process `a` is recreated. Normally, as process creation and termination are expensive, a compiler will detect and avoid this termination-recreation situation by reusing Process `a`. There are subtle differences when the system is distributed and it happens the 1st and the 4th columns in Fig. 2 describe processes located on different machines. Then the termination and recreation are effective and Process `a` migrates from one machine to another.

5 Analysis of rv-programs

Using NFAs Two finite automata may be naturally associated to an rv-program R: a *control automaton* $s(R)$ and an *interaction automaton* $c(R)$. These approximations make abstraction on both spatial and temporal data and drop one dimension of the model.

Control and interaction automata may be used to verify properties of rv-systems generated by rv-programs referring to individual processes or to individual scenario rows ("interaction waves").



(a) Control automaton (b) Interaction automaton

Figure 8: Projection automata of `MAFact`

The control automaton of `MAFact` is described in Fig. 8(a). Its states are 1,3,4 with 1 initial, 4 final; its transitions use the labels of `MAFact` modules. Its language is specified by

$$E_s = (X + (X+U)V + W)^*(X + (X+U)(Y+V))Z^*$$

Each column of a valid `MAFact` scenario is in this language. Adding registers, one may state and verify more

detailed properties on individual processes using (relay-guarantee conditions for their open interaction interfaces and) Floyd-Hoare logic [23].

The interaction automaton of `MAFact` program is described in Fig. 8(b). Its "states" are A,B,C,D with A initial and D final; its transitions use the labels of `MAFact` modules. Its associated language is specified by

$$E_c = Z^*(XU^* + YV^+)W^*$$

Each row of a valid `MAFact` scenario is in this language - typical examples are the 3rd and 4th rows of the scenario in Fig. 3. Adding voices, one may state and verify more detailed properties on individual scenario rows.

Using decomposable FISs The views given by control and interaction automata a completely separate - we have traditional finite automata (paths, not grids!). To get a full picture of the system we have to combine them - grids and the generated FIS may help in this respect.

The *decomposable* FIS *associated to an rv-program* R is $\phi(s(R), c(R))$, i.e., the FIS generated by the control and interaction automata associated to R.

Using grids we can see the effect of applying both control and interaction automata at the same time. For `MAFact`, the grid language is specified by

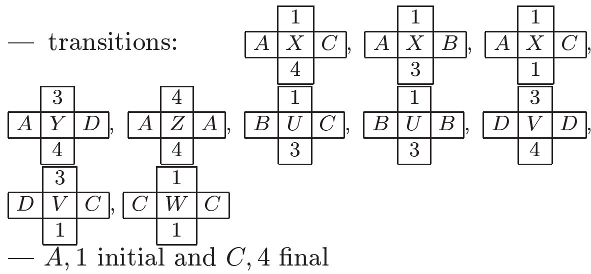
$$E = E_c'^{\star} \cap E_s^{\dagger}$$

where E_c' is E_c with a change of the notation, i.e., $E_c' = Z^\dagger \triangleright (X \triangleright U^\dagger + Y \triangleright V \triangleright V^\dagger) \triangleright W^\dagger$.

Using FISs A finer approximation of an rv-program R is provided by the *associated* FIS $f(R)$. In this case, we still make abstraction on concrete spatial and temporal variables, but the combined control and interaction structure of the rv-program is preserved.

For `MAFact`, the associated FIS is defined by

— transitions:



— $A, 1$ initial and $C, 4$ final

Its grid language consists of blocks

```
X
```

and

```
XUU..U
YVV..V
```

put on the diagonal and having the top-right part filled in with W and the bottom-left part filled in with Z;

```
XWWWWWWWWWWWWWWWWWWWWWW
ZXUUUWWWWWWWWWWWWWWWWWW
ZYVVVWWWWWWWWWWWWWWWWWW
ZZZZXUUUUUUWWWWWWWWWWWW
ZZZZYVVVVVVWWWWWWWWWWWW
ZZZZZZZZZZXWWWWWWWWWWWW
ZZZZZZZZZZZXWWWWWWWWWWW
ZZZZZZZZZZZXUUUUUUUUUUU
ZZZZZZZZZZZYVVVVVVVVVVV
ZZZZZZZZZZZZZZZZZZZZZZX
```

moreover: (1) the second block has at least one U and the number of Us is equal to the number of Vs; (2) the next block following a block is horizontally shifted with k positions to the left, where $k \in \{0, 1\}$ for X and $k = 1$ for the second block; and (3) the top-left and bottom-right corners are labeled by X.

This approximation is finer; indeed

```
XWWWW
ZXUUU
ZZYVV
ZZZZX
```

is not in this language, but it is in the language of the decomposable FIS associated to `MAFact`.

A main goal here would be to verify the correctness of an rv-program against a spatio-temporal specification. We do not have such a formalism completely developed. While useful, its applicability may depend very much on the concrete program and the type of checked property - notice that even at the pure FISs level a general question may be hard (e.g., emptiness is undecidable).

To illustrate the formalism, consider `MAFact`: the basic blocks for its grid language are

```
.2.
1X.
.3.
```

and

```
.2.....
1XU..U.
.YV..V.
.....3.
```

(1,2,3 are labels). For the first block, the key assertions in 1,2,3 are: (`m=0,a=1`); (`n,y`); (`y'=y+n,n'=n-1`). For the second, the key assertions in 1,2,3 are: (`m=1`); (`n,y`); (`y'=y`$\times$`n,n'=n-1`). Such assertions should be proved for these basic blocks, then use the structure of the grid language to lift them to all scenarios. The formalism should be able to detect a few "bags" in the program: it is partially correct, but it fails to terminate when at an interaction step (`m=0,a=0`), or finally (`m=0,a=1`).

A suggestion We just presented three levels of increasing complexity for the analysis of rv-programs: NFAs, decomposable FISs, and FISs[18]. Decomposable FISs appear to be the best choice for starting developing analysis tools of rv-programs: they are expressive enough and their theory is more tractable.

6 Space-Time Duality

Space-time duality interchanges information in space and information in time, e.g., registers and voices. Then, it is naturally lifted to grids, scenarios, FISs, spatio-temporal specifications, rv-systems, and rv-programs which are all space-time invariant.

Briefly, one may define a space-time operator $^\vee$ by:
—on grids: transpose the grid; replace each letter by a dual letter;
—on FISs: interchange states and classes; replace each letter by a dual letter;
—on scenarios: apply $^\vee$ to the underlaying grid; around each letter interchange input registers with input voices

and output registers with output voices ;
—on rv-programs, in each module: interchange class and state labels; interchange temporal and spatial data; switch top-left and bottom-right corners; (notice that, except for label and variable type change, no more modifications are needed in the body of a module).

Theorem 6.1 *For any rv-program R, its space-time dual $R^\vee$ is an rv-program and $(R^\vee)^\vee = R$. Moreover, space-time respects operational semantics and input-output denotation.*

This result suggests to introduce a new programming language construct: `stdual`. Applied to a piece of code, `stdual` interchanges space and time according to the above rules. For instance, if one has an estimation of the time spent by a usual piece of code `P` and it is too high, than using

```
if (time(P) < tooHigh) then {P} else
   {s2t(memory(P)); stdual(P); t2s(memory(P))}
```

it may replace the computation in time of `P` with a computation in space of $\mathtt{P}^\vee$ (`s2t`/`t2s` are space-to-time/time-to-space converters). This is a "heavy" and probably difficult to implement programming construct.

7 Related and future work

Related work (a selection)

—**The model:** Rv-systems belong to a class of machine-oriented models of computation including: register machines [13], data-flow networks [22, 31], asynchronous automata [33], etc. (Petri nets [27] partially belong to this class.) Rv-systems inherit a key feature from Mealy-Moore machines - the "transducer view", whose importance for process algebra models of interaction is emphasized in [1]. Rv-systems allow for "extension in space", namely a potentially unbound number of processes may interact in a scenario row (macrostep) - in the abovementioned models the corresponding interaction is usually atomic and bounded by the transitions' breadth. This unbounded interaction appear to be a key requirement for passing from concurrent to concurrent, OO systems - in the latter, the chain of method invocations may be unbounded.

Actor calculi [3, 4], classical process algebras [6, 19, 24], or more recent versions including π-calculus [25] and bigraphs [21, 26] are more language-oriented - often, they are very powerful in describing process interaction. Usually, they are given an interleaving semantics described by transition systems or term models modulo some equivalences - testing equivalences [14] partially fit with rv-systems transducer view. Rv-systems may be seen as an attempt to close the language-machine gap by incorporating some of the actor and process algebra features into a machine-oriented model. Rv-systems have a true-concurrency semantics based on grids/scenarios. Due to their machine-oriented backbone, rv-systems may be easier to implement on existing or slightly modified architectures.

—**The language:** UML [7] is a popular and very powerful visual formalism - it provides many separate views for the system under construction, including statechart diagrams [17] and interaction diagrams. An integrated view of much simplified versions of statechart and interaction diagrams is captured by FISs [29], on which the syntax and the semantics of rv-programs is based. Rv-programs are simpler, so their theory may be more tractable - pieces of it are scattered throughout the paper. MSCs[19] are often used for system specification in UML. Scenarios, seen as extensions of MSCs, may also be used for programming [18]. Compared with the above, our scenarios for rv-programs have a more relaxed view on time (see below).

—**Specification and analysis:** There are many specification formalisms used for timed systems - see [2, 5, 8, 10] for a selection. Our high-level organization of temporal data based on voices, as well as their use in spatio-temporal specifications and rv-systems, look to be new. The time on our streams is finite, while in traditional models of reactive systems it is infinite; nevertheless, we hope some model checking methods [12] apply to this setting, too. Spatial logics for mobile ambients [11] are based on a branching time/space model - appropriate linear time/space versions may be useful for reasoning about rv-programs. Tile logic [16] based on rewriting models may be useful, too.

Future work (a selection)

There is a large number of research directions and open questions - they may be classified into two main groups. On the one hand, there is a huge amount of practical and theoretical work dealing with interactive systems - as our model is new, its strengths and limitations are still to be investigated and compared with those of related models. On the other hand, there are intrinsic problems generated by this approach - from this latter group, a sample list is included below:
—**Exploit space-time duality:** Lift concepts and results from classical programming to interactive systems, sticking on getting space-time invariant models - a key feature of our approach here. Examples: lift *structured programming* (use it for rv-programming in space and clarify the relationship to OO class manipulation); develop *assembly languages* (low-level versions of rv-programs at bit level), appropriate *architectures* (register-and-voice based architectures), and *compiling*

techniques;

—**Representing** FIS **languages:** Find a *Kleene theorem* for FIS languages (use expressions without intersection or homomorphisms), study its *associated algebras* (keeping an eye on Jones's planar algebras [20]), and use it to develop *verification techniques for rv-programs*.

—**Subclasses of** FISs**:** Except for decomposable FISs, find other expressive enough and tractable subclasses; e.g., check decidability of emptiness problem for *deterministic* FIS*s* (it may be related to the equivalence problem for deterministic pushdown automata).

—**Typing systems:** Develop typing systems for rv-programs, focusing on *types for temporal data* and their role to *security protocols*.

—**Develop programming languages:** Many challenges, including: writing style (avoid labels); integrate OO-programming; develop interpretors/compilers; select a focus area for applications (e.g., cluster/peer-to-peer/grid computing, agents and e-commerce, interactive games, natural language processing, or biological processes) - our beggining attempt is TIME$\mathcal{2}$SING.

Acknowledgments The paper is dedicated to the person with whom I had the most complex interaction: my wife, Elena.

References

[1] S. Abramsky. Retracting Some Paths in Process Algebra. In: *Proc. CONCUR 1996*, LNCS 1119, 1-17.

[2] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.* 15(1993):73-132.

[3] G. Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, 1986.

[4] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *J. Funct. Program.* 7(1997):1-72.

[5] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Asp. Comput.* 3(1991):142-188.

[6] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Inform. Control* 60(1984):109-137.

[7] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley,1998.

[8] M. Broy. Compositional refinement of interactive systems. *JACM*, 44(1997):850-891.

[9] M. Broy, G. Ciobanu, R. Grosu, and G. Stefanescu. Draft on "*Finite interactive systems*", December, 2001.

[10] M. Broy and E.R. Olderog. Trace-oriented models of concurrency. In: *Handbook of process algebra* (Eds. J.A. Bergstra et.al.), North-Holland, 2001, 101-196.

[11] L. Cardelli and A. Gordon. Anytime, anywhere. In: *Proc. POPL 2000*, 365-377.

[12] E. Clarke, O. Grumberg, and D Peled. *Model checking.* MIT Press, 1999.

[13] M. Davis, R. Sigal, and E. Weyuker. *Computability, complexity, and languages: Fundamentals of theoretical computer science.* Academic Press, 1994.

[14] R. De Nicola and M. Hennessy. Testing Equivalences for Processes. *Theor. Comput. Sci.* 34(1984):83-133.

[15] J.S. Frame, G. Robinson, and R.M. Thrall. The hook graphs of the symmetric group. *Canad. J. Math.* 6(1954):316-324.

[16] F. Gadducci and U. Montanari. The tile model. In: *Proof, language, and interaction: Essays in honour of Robin Milner.* MIT Press, 1999, 133-168.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* 8(1987):231-274.

[18] D. Harel and R. Marelly. *Come, let's play: Scenario-based programming using LSCs and the play-engine.* Springer, 2003.

[19] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[20] V.F.R. Jones. Planar algebras. *NZ J. Math.*, to appear.

[21] O.H. Jensen and R. Milner. Bigraphs and transitions. In: *Proc. POPL 2003*, 38-49.

[22] B. Jonsson. A fully abstract trace model for dataflow networks. In: *Proc. POPL 1989*, 155-165.

[23] Z. Manna. *Mathematical theory of computation.* McGraw-Hill, 1974.

[24] R. Milner. *Communication and concurrency.* Prentice Hall, 1989.

[25] R. Milner. *Communicating and mobile systems: the Pi-Calculus.* Cambridge University Press, 1999.

[26] R. Milner. *Bigraphical reactive systems: basic theory.* Technical Report 523, Computer Laboratory, Cambridge University, 2001.

[27] W. Reisig. *Petri Nets: An Introduction.* Springer, 1985.

[28] G. Stefanescu. *Network algebra.* Springer, 2000.

[29] G. Stefanescu. Algebra of networks: Modeling simple networks as well as complex interactive systems. In: *Proof and System Reliability (Proc. 2001 Marktoberdorf Summer School)* 49-78. Kluwer, 2002.

[30] Web page of papers on 2-dimensional languages: http://math.uni-heidelberg.de/logic/bb/2dpapers.html

[31] W. Wadge and E.A. Ashcroft. *Lucid, the dataflow programming language.* Academic Press, 1985.

[32] P. Wegner. Interactive foundations of computing. *Theor. Comput. Sci.* 192(1998):315-351.

[33] W. Zielonka. Notes on finite asynchronous automata. *Theor. Inf. Appl.* 21(1987):99–135.

Fig. 3 repeated (larger picture)

Fig. 6 repeated (larger picture)

Appendix: Complements

FIS **vs. picture languages** (see [29])

In our interpretation of grids, the vertical dimension is used for the progress in time, while the horizontal one is for the progress in space. However, such a distinction is not visible at the grid level - it becomes apparent when we pass to rv-scenarios. If we drop the distinction we get a homogeneous model, similar to those used for 2-dimensional (or picture) languages.

It is known (see the survey papers listed in [30]) that the following statements are equivalent for a 2-dimensional language L (called *recognizable 2-dimensional language*; their class is REC):
— L is recognized by a *on-line tessellation automaton*;
— L is defined by a *basic regular expression and homomorphisms*;
— L is defined by a *local lattice language plus homomorphisms* (or *tiling systems*);
— L is defined by an *existential monadic second order formula*.

However, the situation is more complex in 2 dimensions, e.g: $DFA \subset NFA \subset REC \neq AFA \supset NFA$, where DFA/NFA/AFA stands for deterministic/nondeterministic/alternating finite automata.

Regular expressions and local lattice languages do not cover the class of recognizable languages, but they reach this using homomorphisms - this may be seen as an inconvenience of these mechanisms. This problem does not occur for FISs or for on-line tessellation automata, as they are closed under homomorphisms.

As was shown in [29],

A set of grids is recognizable by a FIS *if and only if it is recognizable by a tiling system.*

By this property, many results known for REC may be used in the FIS setting, too. Two important ones are:

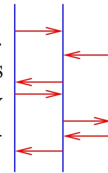
— *(1) If one restricts himself to the top row of grid languages recognized by* FIS*s, then one precisely gets context-sensitive (word) languages.*
— *(2) The emptiness problem for* FIS*s is undecidable.*

There is a rich theory of 2-dimensional languages, mostly related to picture languages, cellular automata, and logics. Along the line developed in this paper, one may try to apply them to interactive and OO programming.

FISs **vs. MSCs** (see [9])

For this case, we retain the space-time distinction, but drop horizontal cyclicity.

Message sequence charts (MSCs) is a popular specification language, adopted in UML. As in the case of grids, in MSCs the vertical dimension is used for time, while the horizontal one is used for space - or "processes". In standard MSCs the number of processes is fixed, so horizontal expansion is forbidden.

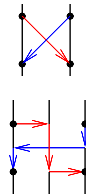


An example of MSC is given in the figure. Each vertical line specifies a process. Process interaction is by message passing, each arrow specifying such a communication. (It is allowed to draw sloped arrows.)

To represent MSCs in the FIS setting one may use the following alphabet

$$V_{MSC} = \{ , , , , , , , , \}$$

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.



The figure illustrates the passing from MSCs to grids. To find a grid representing a crossing we need an extra process to handle communication. Formally, the result is

$$(\triangleright \triangleright) \cdot (\triangleright \triangleright) \cdot (\triangleright \triangleright)$$

FISs over this alphabet are more powerful than MSCs. For instance, one may specify some kind of collision (two processes send messages to each other in the same time), or loss of messages (an process send a message, but the opposite receiver does not exist), or incomplete process specification (an processes may start, but the corresponding end does not exist), etc. While such extensions may be useful in certain cases, we actually put some restrictions on this FIS representation to capture precisely the standard MSCs. These restrictions are:

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

The class of FIS languages contains the language described by the above expressions and it is closed to intersection. Thus, one may meaningfully define a subclass of MSC-like FISs: a *MSC-like* FIS is a FIS over V_{MSC} which satisfies (α) and (β).

As was stated in [9],

Horizontally acyclic MSC-like FIS*s over* V_{MSC} *are equivalent to standard MSCs.*

As a byproduct of this FIS representation of MSCs one gets a natural extension of MSCs where the number of interacting processes is unbounded - they correspond to the full class of MSC-like FISs.

Notes

[1]There is hope that, for the basic class of deterministic FISs, emptiness problem may be decidable (this is currently an open question).

[2]In the literature, the term *picture* is often used as a substitute for grid or two-dimensional word, especially in the context of *picture languages* - see [30] for pointers to papers related on two-dimensional languages. However, our view is more semantical, hence we prefer to use the term 'grid' considered as a two-dimensional version of 'path'. Sometimes we will also use the term 'planar word' (as a two dimensional version of 'word'), but never 'picture'.

[3]If two statements a and b terminate in the same time, then we put them in our sequence in an arbitrary order: ab or ba.

[4]In practice the economy may be smaller - the grids may have repeated letters or may be general grids (formally, such grids may be considered rectangular grids, like the scenario in Fig. 3, but many letters may simply denote identities).

[5]We discovered Theorem 1.2 in December 2001 and reported the result in [9]. In September 2002, while looking for a simpler proof and discussing with I. Tomescu, we found that we simply rediscovered Frame-Robinson-Thrall Theorem.

[6]At an abstract level, this kind of parsing is more or less conventional - one can build a recognizing scenario starting with the south-east borders, too. Generally, there is no clear left-to-right top-to-bottom causality in abstract grids, as example on spiral words or the model for MSCs (in Appendix) suggest. However, this convention is essential for rv-programs as the programs are usually deterministic and the computation can not be reversed.

[7]Notice that the collection of the words laying on the east border $1, 0^1 1^2, 0^3 1^4, \ldots$ is a non context-free language.

[8]Sometimes it is desirable to be able to identify an object through its projections. Then the complexity of the model is simplified. For instance, that is a key property in bigraphs [26, 21]: a bigraph may be decomposed into two graphs, and the decomposition is unique, so the original bigraph may be recovered. Such a property is not true for finite interactive systems: they are more complex than the product of their projection automata.

[9]In fact, decomposable FISs and basic 2-dimensional regular expressions are equivalent.

[10]A Turing tape (or other memory organizations for standard data types used in imperative programming) exhibits a temporal dimension, too. By "spatial" we mean the memory has mainly a spatial extension and it is so organized that its cells may be reached in a constant amount of time. Then, a "temporal" data organization is a dual setting with a limited space, but large time extent.

[11]The approach does not break so sharp with the tradition as it may appear at a first sight. High-level temporal data structure are present in conventional progams, even to a very large extent: they are represented by control structures, the very heart of programming. Indeed, if one picks up a statement and record the time when it is activated, then one gets a temporal datum. But this form of temporal data appear in an impure form, as control structures mix pure control with spatial data. To conclude, our proposal here is to strip control structures of their interaction with spatial data and to present temporal data in a pure form.

[12]If there are more parameters, then one may use a coding to represent a tuple of numbers by a unique number, then apply the above convention.

[13]This kind of horizontal decomposition looks to be related to OO class manipulation, but the precise relationship still has to be investigated. It is a very powerful mechanism and it is supported in rv-programs by a kind of "programming in space" paradigm.

[14]If one sticks to have a functional specification, than star should be avoided and a more detailed and restricted functional specification has to be given.

[15]Turing test on natural vs. artificial intelligence.

[16]As we said, within the body of a module there is no difference between a spatial and a temporal variable. Consequentaly, the module body is a piece of code in a traditional imperative programming language and one may use classical techniques to find its input-output relation.

[17]For notational convenience, the range of indices for these arrays is denoted `1..n`, not `0..n-1` as in C.

[18]There is a different way to associate a FIS to an rv-program always resulting in a decomposable FIS: For a module X of an rv-program one may use different labels $X_1, X_2, \ldots$ for each "section" of the module leading to an exit via `goto` statement. Now, the resulting FIS is decomposable - a FIS transition corresponds to a unique module and a unique exit via `goto` statement, hence it has a unique name. However, this does not rule out the FISs role: the full FIS setting is conceptually simpler, more expressive, and more robust (e.g., the class of FISs is closed to homomorphisms, while the class of decomposable FISs is not closed).

[19]See the Appendix for a formalization of MSCs in the FISs setting.