

## ALGEBRA OF NETWORKS

*Modeling simple networks, as well as complex interactive systems.*

G. STEFANESCU

*Department of Computer Science*

*National University of Singapore*

gheorghe@comp.nus.edu.sg

**Abstract.** The first part of the paper contains an overview of *Network Algebra* (NA) book [35]. The second part introduces *finite interactive systems* as an abstract mathematical model of agents' behaviour and their interaction.

**Network Algebra:** This book is devoted to an algebraic study of networks and their behaviour. The kernel of the involved algebraic structures is BNA (Basic Network Algebra) structure. These axioms are sound and complete for (abstract) networks modulo graph isomorphism. Branching constants are added to the BNA signature, together with the corresponding weak axioms leading to an axiomatization for networks with branching constants, modulo graph isomorphism. Strong and enzymatic axioms for branching constants are used to depart from graph isomorphism models. Using them, Elgot theory may be introduced, as a sound and complete axiomatization for input behaviour (regular trees). Extensions to the input-output behaviour of deterministic networks (Park theories) and of nondeterministic networks (Kleene theories) are included.

These general results on abstract networks are then specialized on certain particular cases including flowchart schemes, finite automata, process algebra, data-flow networks, or Petri nets.

Finally, a brief presentation of the Mixed Network Algebra models is included in NA book. These models are obtained mixing network algebra models for control, space, and time. A few results are presented, including a challenging space-time duality thesis. Many interesting questions risen by this model are open.

**Finite interactive systems:** The second half of the paper introduces *finite interactive systems*, an abstract mathematical model of agents' behaviour and their interaction. Finite interactive systems may be seen as a version of the Mixed Network Algebra where the mixture is made at the Kleene theory level. Its aim is the fill in the gap between behaviour and class diagrams used in modern (UML) system design [4].

## 1. An overview of Network Algebra approach

### 1.1. KEY FEATURES

The term network is used in a broad sense here as consisting of a collection of interconnecting cells.

#### 1.1.1. Models

Two radically different interpretations of this enlarged notion of network are particularly relevant. First, virtual networks are obtained using the Cantorian interpretation in which at most one cell is active at a given time. With this interpretation, Network Algebra (NA) covers the classical models of control, including finite automata or flowchart schemes. In a second Cartesian interpretation, each cell is always active. This implies models for reactive and concurrent systems such as Petri nets or data-flow networks may be covered as well.

These two interpretations may be mixed. A sketch of the resulting much more complicated Mixed Network Algebra (MixNA) models is included, as well.

#### 1.1.2. Algebraic structures

The results are presented in the unified framework of the calculus of flownomials. This is an abstract calculus very similar to the classical calculus of polynomials.

The kernel structure is the *BNA* (“Basic Network Algebra”). After their introduction in the context of control flowcharts setting [33], the BNA axioms were rediscovered in various fields ranging from circuit theory to action calculi, from data-flow networks to knot theory (traced monoidal categories), from process graphs to functional programming.

In general, the involved algebraic structures are BNAs enriched with branching constants and additional specific axioms. The branching constants are specified by  $xy$  with  $x \in \{a, b, c, d\}$  and  $y \in \{\alpha, \beta, \gamma, \delta\}$ ; the axioms are divided into three groups: weak, strong, and enzymatic axioms.

As already said, an important feature of the NA approach is the uniform presentation of many apparently unrelated models which have appeared in the literature. A few such structures are described below.

- (1) *BNA* (case  $a\alpha$ -weak,  $a\alpha$ -strong,  $a\alpha$ -enzymatic): any transformation which preserves the graph-isomorphism relation is valid (no duplication or removal of wires or cells is permitted);
- (2) *xy-symocats with feedback* (case  $xy$ -weak,  $a\alpha$ -strong,  $a\alpha$ -enzymatic): in these cases duplication or removal is permitted for wires, but not for network cells;

- (3) *Elgot theory* (case  $b\delta$ -weak,  $a\delta$ -strong,  $a\delta$ -enzymatic): in this setting, networks may be completely unfolded to get “regular” trees; Elgot theory axioms capture the transformation rules that are valid for these trees;
- (4) *Kleene theory* (case  $d\delta$ -weak,  $d\delta$ -strong,  $d\delta$ -enzymatic): in this setting, the input-output sequences modeling network behaviour define “regular” languages; Kleene theory captures all the identities which are valid for regular languages (flownomial and regular expressions are equivalent in this context);
- (5) *Park theory* (case  $b\delta$ -weak,  $b\delta$ -strong,  $b\delta$ -enzymatic): this is a setting in-between Elgot and Kleene theories; it deals with input-output behaviour, but in the deterministic case;
- (6) *Căzănescu–Ungureanu theory* (case  $b\delta$ -weak,  $a\delta$ -strong,  $a\alpha$ -enzymatic): this structure is a BNA over a coalgebraic theory; Floyd–Hoare logic for program correctness may be developed in this setting;
- (7) *Conway theory* (case  $d\delta$ -weak,  $d\delta$ -strong,  $a\alpha$ -enzymatic): this structure is a BNA over a matrix theory; this setting suffices for proving Kleene theorems;
- (8) *Milner theory* (case  $d\delta$ -weak,  $a\delta$ -strong,  $a\delta$ -enzymatic): this setting may be seen as a lifting with weak constants of the Elgot theory to the nondeterministic case; this setting is used to build-up process algebra.

This ability of the calculus of flownomials to cover many particular algebras may be a good motivation for the reader to follow this, maybe too abstract, calculus.

### 1.1.3. *Results*

The main results presented in the book include: axiomatizations for isomorphic networks without or with branching constants; axiomatizations for several classes of relations; axiomatizations for regular trees or regular languages; the Universality Theorems; the Structural Theorems; correctness of Floyd–Hoare logic; the duality between flowcharts and circuits; correctness of ACP (Algebra of Communicating Processes) with respect to bisimulation semantics; axiomatization for input–output behaviour of deterministic data-flow networks; a Kleene-like theorem for Petri net languages; a construction of the free distributive category; the space-time duality principle.

## 1.2. A BRIEF INTRODUCTION TO NETWORK ALGEBRA

In this introduction we briefly present the *calculus of flownomials*, a unified algebraic theory for various types of networks. This approach is the result of an effort to integrate various algebraic approaches used in computer science.

We may agree: a variety of models is desirable. Then, one may use the most appropriate one for the task. But the models should not be completely unrelated. If so, there will be no transfer of information, results, techniques, etc. with the undesired result of duplication of effort and lost of time.

As a source of inspiration, one may look at the marvelous world of polynomials. There is a lot of freedom to accommodate various models, but still within a clear and unique (or coherent) algebraic framework.

Flownomials were designed with the explicit intention to inherit some of the qualities of polynomials. The motivation for their introduction comes from particular concrete models of computation, but the result is an abstract mathematical model that may be used whenever the underlying rules are valid. (A striking example is the coincidence of the kernel BNA structure of the calculus of flownomials [33] with the later introduced traced monoidal categories occurring in the study of quantum groups [21].)

Flownomials may be used as a mechanism for translating facts from one field to another, or, furthermore, may transform themselves into their own field from where translation to particular models is straightforward.

### 1.2.1. Regular expressions

A basic calculus for sequential computation is provided by Kleene's *calculus of regular expressions*, see, e.g., [23], [11]. It has a simple syntax and a simple semantics. The regular expressions over a set of atomic symbols  $X$ , denoted as  $\text{RegExp}(X)$ , are given by the following grammar

$$E ::= E + E \mid E \cdot E \mid E^* \mid 0 \mid 1 \mid x (x \in X)$$

In the standard semantics regular expressions are interpreted as languages, i.e., sets of strings. The operations are interpreted as "union," "catenation," and "iterated catenation," respectively, while the constants are interpreted as "empty set" and "empty string," respectively.

This calculus is a basic calculus for sequential computation and has a broad range of applications in the design and analysis of circuits and programming languages, study of automata and grammars, analysis of flowchart schemes, semantics of programming languages, etc.

However, the calculus has a strong, but less visible, restriction. In order to handle complex objects like automata, grammars, circuits, etc. one has to add one more operation to the above syntax, namely a *matrix building operator*: if  $a_{ij}, i \in [m], j \in [n]$  ( $[k]$  denotes the set  $\{1, 2, \dots, k\}$ ) are expressions, then the matrix  $(a_{ij})$  is an (extended) expression. Such an expression is then used to specify the behaviour of a complex object with  $m$  inputs and  $n$  outputs.

By changing the point of view from “elements” to such “complex objects,” one may see that the above construction is based on the following restriction:

*Matrix theory rules: reducing  $m$ -to- $n$  to 1-to-1 networks*

- Let  $f : m \rightarrow n$  denote the behaviour of a network  $F$  with  $m$  input ports and  $n$  output ports.
- For  $i \in [m], j \in [n]$ , let  $f_{ij}$  denote the behaviour of the network obtained by restricting  $F$  to its  $i$ -th input and  $j$ -th output only.
- By the matrix theory rules,  $f$  is uniquely determined by these  $f_{ij}$ .

This is a very strong restriction indeed and it makes the calculus unsuited for many computation models. In particular, as is well-known, in such a framework one may rather freely copy or discard network cells, a questionable property in certain cases.

### 1.2.2. Iteration theories

An algebraic setting in between regular expressions and the calculus of flownomials is provided by an algebraic theory modeling flowchart schemes or data-flow networks. In such a case a weaker hypothesis is used.<sup>1</sup>

*Coalgebraic theory rules: from  $m$ -to- $n$  to 1-to- $n$  networks*

- Let  $f : m \rightarrow n$  denote the behaviour of a network  $F$  with  $m$  input ports and  $n$  output ports.
- For  $i \in [m]$ , let  $f_i$  denote the behaviour of the network obtained from  $F$  by considering only its  $i$ -th input port (all the output ports are preserved).
- By coalgebraic theory rules,  $f$  is uniquely determined by these  $f_i$ .

The resulting algebraic calculus is more general than the calculus of regular expressions and it is useful to study certain semantic models, see, e.g., [12], [3], [26]. However, it is still restrictive and no simple syntactic model similar to Kleene’s model of regular expressions exists.

### 1.2.3. Flownomials

The *calculus of flownomials* may be seen as an extension of the calculus of regular expressions to the case of many-input/many-output atoms. Its main ingredients were presented in a series of papers, including [31], [32], [33], [9], [10]; see [35] for more on this.

<sup>1</sup> In the given form the hypothesis fits with the semantic models of flowchart schemes. The case of data-flow models is dual, i.e., a many-output network is specified as a tuple of single-output ones.

*Symocat rules: atomic m-to-n networks*

Using symocats (shorthand for symmetric strict monoidal categories), no reductions of a network behaviour  $f : m \rightarrow n$  to certain parts is a priori possible. The network is atomic; its behaviour cannot be decomposed into simpler parts.

The calculus of flownomials is an abstract calculus for networks (= labeled directed hyper-graphs) and their behaviours. It starts with two families of doubly-ranked elements:

- a family of variables  $X = \{X(a, b)\}_{a, b \in M}$ ; these variables denote “black boxes,” i.e., atomic elements with two types of connecting ports: input ports (described by  $a$ ) and output ports (described by  $b$ ).
- a connection structure  $T = \{T(a, b)\}_{a, b \in M}$ ; the elements in  $T$  model known processes between the input/output interfaces; e.g., its elements may model the flow of messages that are transmitted via the connecting channels between the input/output pins; this is the “known” (or “interpreted”) part of the model; the flownomial operations are supposed to be already defined on this part.

$M = (M, \star, \epsilon)$  is a monoid modeling network interfaces; if  $M$  is a free monoid, then an interface may be considered as a string of atomic ports/pins. We also use the functions  $i : \dots \rightarrow M$  and  $o : \dots \rightarrow M$  that give the corresponding interfaces for inputs and outputs, respectively.

Flownomial expressions over  $X$  and  $T$  are specified by:<sup>2</sup>

$$E ::= E \star E \mid E \cdot E \mid E \uparrow^c \mid \mathbb{1}_a \mid {}^a X^b \mid \wedge_k^a \mid \vee_a^k \mid x(\in X) \mid f(\in T)$$

where  $a, b, c \in M$  and  $k \in \mathbb{N}$ . Let  $\text{FlowExp}[X, T]$  denote their set.

The collection of *flownomial expressions* is obtained starting with the elements in  $X$  and  $T$ , certain constants (to be described below), and applying three operations:

*juxtaposition “ $\star$ ”, composition “ $\cdot$ ”, and feedback “ $\uparrow$ ”*

Juxtaposition  $f \star g$  is always defined and  $i(f \star g) = i(f) \star i(g)$  and  $o(f \star g) = o(f) \star o(g)$ . The composite  $f \cdot g$  is defined only in the case  $o(f) = i(g)$  and, in such a case,  $i(f \cdot g) = i(f)$  and  $o(f \cdot g) = o(g)$ . The feedback  $f \uparrow^c$  is defined only in the case  $i(f) = a \star c$  and  $o(f) = b \star c$ , for some  $a, b \in M$  and, in such a case,  $i(f \uparrow^c) = a$  and  $o(f \uparrow^c) = b$ . (Technically, a condition

<sup>2</sup> Notice the simple way regular expressions fit in this setting: this is the particular instance where  $M = (\mathbb{N}, +, 0)$  and  $0 = \wedge_0^1 \cdot \vee_1^0$ ,  $1 := \mathbb{1}_1 (= \wedge_1^1 = \vee_1^1)$ ,  $f + g := \wedge_2^1 \cdot (f \star g) \cdot \vee_1^2$ ,  $f \cdot g := f \cdot g$ ,  $f^* := [\vee_1^2 \cdot (1 + f) \cdot \wedge_2^1] \uparrow^1$ .

$a \star c = a' \star c \Rightarrow a = a'$  is necessary for the feedback case; it is valid when  $M$  is a free monoid.)

The *acyclic case* refers to the case without feedback.

The support theory for connections  $T$  should “contain” certain finite binary relations. The main classes used below are: bijections  $\mathbb{B}i$ , functions  $\mathbb{F}n$ , partial functions  $\mathbb{P}fn$  and relations  $\mathbb{R}el$ . At the abstract level, this means that we should have some constants generating these relations. The collection of constants used for this purpose is:

*identity*  $l_a$ , (*block*) *transposition*  ${}^aX^b$ , (*block*) *ramification*  $\wedge_k^a$ ,  
and (*block*) *identification*  $\vee_a^k$ , where  $a, b \in M$  and  $k \in \mathbb{N}$

The types of constants are as follows:  $i(l_a) = o(l_a) = a$ ;  $i({}^aX^b) = a \star b$ ,  $o({}^aX^b) = b \star a$ ;  $i(\wedge_k^a) = a$ ,  $o(\wedge_k^a) = a \star \dots \star a$  ( $k$  times);  $i(\vee_a^k) = a \star \dots \star a$  ( $k$  times),  $o(\vee_a^k) = a$ .

Each of the above classes of relations (and 13 other classes) may be generated using certain sub-collections of these constants only. To specify them one has to *restrict* the using of the branching constants  $\wedge_m$  and  $\vee^n$  as follows:

*Restrictions on branching degrees*

- Restriction  $x = a$  means the constants of the type  $\wedge_m^a$  always have  $m = 1$ ; restriction  $x = b$  means  $m \leq 1$ ;  $x = c$ , means  $m \geq 1$ ; and  $x = d$  means no restriction (arbitrary  $m$ ).
- Restriction  $y$  is defined in a similar manner, using  $n$  of the constants  $\vee_a^n$  and the corresponding Greek letters.

The notation  $xy$ , where  $x \in \{a, b, c, d\}$  and  $y \in \{\alpha, \beta, \gamma, \delta\}$ , refers to the class of relations generated with branching constants satisfying restriction  $xy$ . E.g.,  $a\alpha$  refers to bijections,  $a\delta$  to functions,  $b\delta$  to partial functions,  $d\delta$  to relations, etc.

For flownomials we may single out certain abstract algebraic structures to play the role the ring structure is playing for polynomials. They are extensions of a basic algebraic structure, called *BNA* (*Basic Network Algebra*), or *a $\alpha$ -flow*, which uses  $l$  and  $X$  in its definition only (no branching constants). The critical axioms used to define these extensions are:

(1) *commutation of the branching constants to the other elements*

$$\begin{aligned} f \cdot \wedge_k^b &= \wedge_k^a \cdot (kf) \\ \vee_a^k \cdot f &= (kf) \cdot \vee_b^k \end{aligned}$$

where  $f : a \rightarrow b$  and  $kf = f \star \dots \star f$ ,  $k$ -times.

This axiom scheme is used in two ways:

- (1a) In the *weak* variant it is used only for certain ground terms  $f$ 's.<sup>3</sup>  
 (1b) In the *strong* variant it applies to arbitrary  $f$ 's.

(2) *enzymatic axiom for the looping operation*

$$f \cdot (l_b \star z) = (l_a \star z) \cdot g \quad \text{implies} \quad f \uparrow^c = g \uparrow^d$$

where  $z : c \rightarrow d$  is an abstract relation and  $f : a \star c \rightarrow b \star c$ ,  $g : a \star d \rightarrow b \star d$  are arbitrary elements.

Roughly speaking (see Subsection 1.2.4 for more precise definitions):

- an *xy-symocat* (shorthand for *xy-symmetric strict monoidal category*) is the acyclic structure defined using the weak commutation for all *xy*-terms;
- in a *strong xy-symocat*, the strong commutation axioms hold for all *xy*-terms;
- finally, an *xy-flow* is: (1) a BNA over an *xy-symocat*, which is (2) a strong *xy-symocat*, and (3) obeys the enzymatic axiom whenever  $z$  is an abstract *xy*-relation.

A large number of algebraic structures may be obtained by choosing the branching constants and certain weak, strong or enzymatic axioms for them. This freedom gives flexibility to the calculus which is well-suited to model various kind of networks and equivalences.

*Classification using weak/strong/enzymatic scheme*

The main algebraic structures used in NA book are defined as

$$\text{BNA} + x_1y_1\text{-weak} + x_2y_2\text{-strong} + x_3y_3\text{-enzymatic}$$

The first restriction  $x_1y_1$  shows what branching constants may be used, the second  $x_2y_2$  which branching constants strongly commute with arbitrary arrows, and the third  $x_3y_3$  for which branching constants the enzymatic rule may be applied. (Usually  $x_1y_1 \geq x_2y_2 \geq x_3y_3$ , where “ $\geq$ ” means more constants, hence weaker restriction.)

Polynomials may be seen as classes of equivalent polynomial expressions. In a similar manner, *flownomials* are defined as classes of equivalent flownomial expressions under an appropriate relation of equivalence. (However, we have various classes of flownomials, for various restrictions *xy*.) Moreover, in both calculi there are two equivalent ways to introduce such equivalences:

<sup>3</sup> A *ground term*, or an *abstract relation*, is a term written with the flownomial operations and certain constants in  $l, X, \wedge_k, \vee^k$ . A *ground xy-term*, or an *abstract xy-relation*, is a term which may be written using only constants fulfilling restriction *xy*.

(1) by using the rules of the appropriate algebra, or (2) by using normal form representations.

For the first way, a standard equivalence  $\sim_{xy}$  on flownomial expressions may be introduced using  $xy$ -flow rules.

The second way to define such an equivalence is to use *normal form flownomial expressions*. Such an expression over  $X$  and  $T$  is of the following type:

$$((l_a \star x_1 \star \dots \star x_k) \cdot f) \uparrow^{i(x_1) \star \dots \star i(x_k)}$$

with  $f : a \star o(x_1) \star \dots \star o(x_k) \rightarrow b \star i(x_1) \star \dots \star i(x_k)$  in  $T$  and  $x_1, \dots, x_k$  in  $X$ . Then, two normal form flownomial expressions  $F = ((l_a \star x_1 \star \dots \star x_m) \cdot f) \uparrow^{i(x_1) \star \dots \star i(x_m)}$  and  $G = ((l_a \star y_1 \star \dots \star y_n) \cdot g) \uparrow^{i(y_1) \star \dots \star i(y_n)}$  are *similar* via a relation  $r : m \rightarrow n$  iff

- (i)  $(i, j) \in r \Rightarrow x_i = y_j$ ;
- (ii)  $f \cdot (l_b \star i(r)) = (l_a \star o(r)) \cdot g$ .

Here  $i(r)$  represents the “block” extension of  $r$  to inputs (e.g., for a free monoid  $M$ , if  $r$  relates two variables  $x_i$  and  $y_j$ , then  $i(r)$  relates the 1st input of  $x_i$  to the 1st input of  $y_j$ , the 2nd to the 2nd, and so on); similarly  $o(r)$  denotes the block extension of  $r$  to outputs.

Simulation is reflexive, transitive, and compatible with the flownomial operations, but not symmetric. The generated congruence, denoted  $\overset{xy}{\iff}$ , is the equivalence relation generated by simulation via  $xy$ -relations.

It turns out that  $\overset{xy}{\iff}$  coincides with  $\sim_{xy}$  in the basic cases studied in NA book:  $a\alpha$ ,  $a\delta$ ,  $b\delta$ , or  $d\delta$  (sometimes the proofs require certain additional conditions). Generally speaking,  $\sim_{xy}$  is coarser.

A basic model is  $[T, X]_{xy}$ , i.e., the algebra of normal form flownomial expressions modulo  $\overset{xy}{\iff}$  equivalence. Another one is  $\text{FlowExp}[X, T]_{xy}$  consisting of classes of  $\sim_{xy}$ -equivalent normal form flownomials over  $X$  and  $T$ . The above fact shows that these models are isomorphic in the basic cases  $a\alpha$ ,  $a\delta$ ,  $b\delta$ , and  $d\delta$ .

#### 1.2.4. Basic algebraic structures

A few basic algebraic structures used in the calculus of flownomials are described below, with a brief description of the basic features of these models. The axioms are listed in Table I. We start with the graph-isomorphism models.

**BNA Case** ( $a\alpha$ -weak,  $a\alpha$ -strong,  $a\alpha$ -enzymatic): axioms I-II

This is the basic algebraic structure we are working with. In this setting, one may represent cyclic networks. The cells in  $X$  may have multiple input or output pins. The signature allows us to use only bijective relations. More sophisticated connections may be used, but only

TABLE I. The standard axioms for the calculus of flownomials

## I. Axioms for symocats

- B1  $f \star (g \star h) = (f \star g) \star h$   
 B2  $l_\epsilon \star f = f = f \star l_\epsilon$   
 B3  $f \cdot (g \cdot h) = (f \cdot g) \cdot h$   
 B4  $l_a \cdot f = f = f \cdot l_b$   
 B5  $(f \star f') \cdot (g \star g') = f \cdot g \star f' \cdot g'$   
 B6  $l_a \star l_b = l_{a \star b}$   
 B7  ${}^a X^b \cdot {}^b X^a = l_{a \star b}$   
 B8  ${}^a X^{b \star c} = ({}^a X^b \star l_c) \cdot (l_b \star {}^a X^c)$   
 B9  $(f \star g) \cdot {}^c X^d = {}^a X^b \cdot (g \star f)$   
 for  $f : a \rightarrow c$ ,  $g : b \rightarrow d$

## II. Axioms for feedback

- R1  $f \cdot (g \uparrow^c) \cdot h = ((f \star l_c) \cdot g \cdot (h \star l_c)) \uparrow^c$   
 R2  $f \star g \uparrow^c = (f \star g) \uparrow^c$   
 R3  $(f \cdot (l_b \star g)) \uparrow^c = ((l_a \star g) \cdot f) \uparrow^d$   
 for  $f : a \star c \rightarrow b \star d$ ,  $g : d \rightarrow c$   
 R4  $f \uparrow^\epsilon = f$   
 R5  $(f \uparrow^b) \uparrow^a = f \uparrow^{a \star b}$   
 R6  $l_a \uparrow^a = l_\epsilon$   
 R7  ${}^a X^a \uparrow^a = l_a$

## III. Axioms for (angelic) branching constants, without feedback

- A1a  $\wedge^a \cdot (\wedge^a \star l_a) = \wedge^a \cdot (l_a \star \wedge^a)$   
 A1b  $\wedge^a \cdot (\perp^a \star l_a) = l_a$   
 A1c  $(\vee_a \star l_a) \cdot \vee_a = (l_a \star \vee_a) \cdot \vee_a$   
 A1d  $(\top_a \star l_a) \cdot \vee_a = l_a$   
 A2a  $\wedge^a \cdot {}^a X^a = \wedge^a$   
 A2b  ${}^a X^a \cdot \vee_a = \vee_a$   
 A3a  $\top_a \cdot \perp^a = l_\epsilon$   
 A3b  $\top_a \cdot \wedge^a = \top_a \star \top_a$   
 A3c  $\vee_a \cdot \perp^a = \perp^a \star \perp^a$   
 A3d  $\vee_a \cdot \wedge^a =$   
 $(\wedge^a \star \wedge^a) \cdot (l_a \star {}^a X^a \star l_a) \cdot (\vee_a \star \vee_a)$   
 A4  $\wedge^a \cdot \vee_a = l_a$

## IV. Feedback on (angelic) branching constants

- R8  $\wedge^a \uparrow^a = \top_a$   
 R9  $\vee_a \uparrow^a = \perp^a$   
 R10  $[(l_a \star \wedge^a) \cdot ({}^a X^a \star l_a) \cdot (l_a \star \vee_a)] \uparrow^a = l_a$

- A5a  $\perp^{a \star b} = \perp^a \star \perp^b$   
 A5b  $\wedge^{a \star b} = (\wedge^a \star \wedge^b) \cdot (l_a \star {}^a X^b \star l_b)$   
 A5c  $\top_{a \star b} = \top_a \star \top_b$   
 A5d  $\vee_{a \star b} = (l_a \star {}^b X^a \star l_b) \cdot (\vee_a \star \vee_b)$   
 [A5e  $\perp^\epsilon = l_\epsilon$   
 A5f  $\wedge^\epsilon = l_\epsilon$   
 A5g  $\top_\epsilon = l_\epsilon$   
 A5h  $\vee_\epsilon = l_\epsilon]$

## V. The strong axioms

- Sa  $f \cdot \perp^b = \perp^a$   
 Sb  $f \cdot \wedge^b = \wedge^a \cdot (f \star f)$   
 Sc  $\top_a \cdot f = \top_b$   
 Sd  $\vee_a \cdot f = (f \star f) \cdot \vee_b$

VI. The enzymatic rule ( $E$  is a class of finite, abstract relations)

ENZ $_E$ : if for  $f : a \star c \rightarrow b \star c$  and  $g : a \star d \rightarrow b \star d$  there exists  $y : c \rightarrow d$  in  $E$  such that  $f \cdot (l_b \star y) = (l_a \star y) \cdot g$ , then  $f \uparrow^c = g \uparrow^d$

within the connection theory  $T$ . Any transformation which preserves the graph-isomorphism relation is valid. This is a linear-like setting: no duplication or removal of network cells is permitted.

**$xy$ -symocats with feedback** Case ( $xy$ -weak,  $a\alpha$ -strong,  $a\alpha$ -enzymatic):  
I-II and III-IV(using  $xy$  constants)

Only 9 cases are allowed for  $xy$ , namely those with  $x = b$ , or  $y = \beta$ , or  $xy \in \{a\alpha, d\delta\}$ . In these cases branching constants are added to the syntactic part of the calculus. They come with appropriate algebraic rules, which were designed in such a way: (1) to be strong enough to allow usual relations with their (angelic) operations to be mapped in these structures, and (2) to preserve the graph-isomorphism setting. Hence, duplication or removing is permitted for connections/wires, but not for network cells.

Together with BNA, the next 3 cases enter into the category of  $xy$ -flows. In these cases, all three conditions (weak, strong, enzymatic) are applied for the same class of relations, with a slight exception for the weak case, where actually the closure of the restriction to feedback has to be used.

**Elgot theory** Case ( $b\delta$ -weak,  $a\delta$ -strong,  $a\delta$ -enzymatic):

I-II, III-IV(using  $\top, \vee, \perp$ ), V(Sc-d), VI (for  $E = \mathbb{F}n$ )

In this case, by the  $a\delta$ -strong rules one can safely restrict to single-input networks, variables, etc. As a byproduct, such networks may be completely unfolded to get “regular” trees. Elgot theory axioms capture the transformation rules that are valid for these trees. Rather different networks may produce the same unfolding tree, but the enzymatic rule (together with the other ones) suffices to identify them.

**Kleene theory** Case ( $d\delta$ -weak,  $d\delta$ -strong,  $d\delta$ -enzymatic): I-V, VI(IRel)

Kleene theory does the same as Elgot theory, but for input–output computation sequences (paths) for nondeterministic networks. In this case, due to the  $d\delta$ -strong rules, one can restrict to single-input/single-output networks, cells, etc. For these networks input–output sequences may be defined, producing the associated regular languages. Kleene theory captures all the identities which are valid in this model. (Actually, flownomials and regular expressions are equivalent in this context.) Again, the enzymatic rule proves to be of crucial importance in the proof of axioms’ completeness.

**Park theory** Case ( $b\delta$ -weak,  $b\delta$ -strong,  $b\delta$ -enzymatic):

I-II, III-IV( $\top, \vee, \perp$ ), V(Sa,Sc-d), VI(IPfn)

This is a setting in-between Elgot and Kleene theories. It deals with

input–output behaviour, but in the deterministic case. Hence matrices cannot be used here. While this case has not (yet!) the widely recognized value of the two cases above, it has its own beauty. For instance, the Structural Theorem modeling deterministic minimization is an interesting and nontrivial extension of the one for Elgot theories, but still the proof is clean, with a rather natural additional “identification-free” hypothesis.

Three more cases deserve some special attention. In these structures, there is a mismatch between the restrictions used for the weak, strong, and enzymatic rules.

**Căzănescu–Ungureanu theory** Case ( $b\delta$ -weak,  $a\delta$ -strong,  $a\alpha$ -enzymatic):

I-II, III-IV( $\top, \vee, \perp$ ), V(Sc-d)

Căzănescu–Ungureanu theory is a BNA over a coalgebraic theory. Floyd–Hoare logic for program correctness may be developed in this setting.

**Conway theory** Case ( $d\delta$ -weak,  $d\delta$ -strong,  $a\alpha$ -enzymatic): I-V

This case is similar to the above one, so a Conway theory is a BNA over a matrix theory. This setting suffices for proving Kleene theorems.

**Milner theory** Case ( $d\delta$ -weak,  $a\delta$ -strong,  $a\delta$ -enzymatic):

I-IV, V(Sc-d), VI(IFn)

This may be seen either as a lifting (with weak constants) of the Elgot theory to the nondeterministic case, or as a relaxation of Kleene theory, preserving the strong and enzymatic rules just for functions. This setting is used to build-up process algebra.

### 1.2.5. *Basic results*

The main results on the calculus of flownomials presented in NA book are:

**Expressiveness results:** Various classes of networks are represented by flownomial expressions over appropriate connecting theories. E.g.: (a) nondeterministic networks are represented by flownomials over  $\mathbb{I}Rel$ ; deterministic networks are represented by flownomials over  $\mathbb{I}Fn$ , etc. (b) networks with value passing channels are represented by flownomials over relational semantic models  $AddRel(D)/MultRel(D)$ ; (c) higher order flownomials are represented by flownomials over other flownomials.

**Axiomatization results:** Correct (i.e., sound and complete) axiomatizations for the classes of equivalent networks with respect to various natural equivalence relations are presented. E.g., BNA axioms are

correct for graph isomorphism;  $xy$ -symocat with feedback axioms are correct for graph isomorphism with various constants; Elgot theory axioms are correct for deterministic input behaviour; Milner theory axioms are correct for nondeterministic input behaviour (bisimilar networks); Park theory axioms are correct for deterministic IO behaviour; Kleene theory axioms are correct for nondeterministic IO behaviour.

**Uniformity:** All the axiomatization results above are proved in an abstract setting, namely in the case the connecting wires are represented by morphisms in an appropriate abstract theory. Hence we have uniform proofs that may be used, for example, to the case of connections made by simple wiring relations, or by message passing channels, or by other flownomials. On the other hand, the proofs are ordered within a natural framework: from the simplest case of BNAs, to the most complicated case of Kleene theories.

**Universality:** In such an abstract setting the correctness problem consists of the preservation of the algebraic structure when one passes from connections to classes of equivalent networks. This problem is solved by theorems of the following type (in cases  $b\delta$  and  $d\delta$  the proofs are done under some mild additional conditions for the connecting theory  $T$ ): If  $T$  is an  $xy$ -flow, then  $[X, T]_{xy}$  is  $xy$ -flow; moreover,  $[X, T]_{xy}$  forms the  $xy$ -flow freely generated by adding  $X$  to  $T$ .

If one replaces “flownomials” by “polynomials” and “ $xy$ -flow” by “ring,” then one gets the classical universal property for polynomials. In particular, this result shows that one may use standard algebraic refinement methods.

**Special networks:** The calculus of flownomials is general enough to cover various types of networks, including flowchart schemes, automata, data-flow networks, Petri nets. (A process algebra version, re-cast in the timed-NA framework, may, perhaps, be included as well; in the current approach, process algebra has no “identities.”)

**Modularity:** The approach is variable-free with respect to interface references. As a byproduct a high degree of modularity is achieved. One may freely shift pieces of networks from one part to another.

### 1.2.6. *Mixed calculi*

Appropriate instances of the NA models may be found for control (flowcharts), space (data-flows) and time (processes). A very promising line for current research is to mix such models in order to have a unique calculus

for all these features of the computing systems. This may lead towards an algebraic calculus for (concurrent, object-oriented) software systems. A few results are presented, including a challenging space-time duality thesis. Many interesting questions risen by this model are open.

The second part of this paper contains a detailed introduction to finite interactive system, which may be seen as a version of the Mixed Network Algebra where the mixture is made at the Kleene theory level.

## 2. Finite interactive systems

### 2.1. INTRODUCTION

This section gives a brief introduction to *finite interactive systems*, an abstract mathematical model of agents' behaviour and their interaction.

The agents we are talking about are those specified in standard *concurrent object-oriented languages (COOP)*. The key point is the observation that *planar words may be seen as interaction running patterns* for programs written in these COOP languages much in the same way as usual words/paths represent running paths of classical sequential programs. Then, finite interactive systems may be seen as a kind of two-dimensional finite automata melting together a state transforming automaton with a behaviour transforming one (this latter transformation is responsible for modeling the interaction of the threads generated by the first automaton).

### 2.2. PLANAR WORDS AND REGULAR EXPRESSIONS

#### 2.2.1. Planar words

A *planar word* is a *rectangular* two-dimensional area filled in with atomic letters in a given alphabet  $V$ . Each letter in  $V$  is to be seen as a two-dimensional atom having its own *northern*, *southern*, *western* and *eastern* border. For the beginning we do not consider any typing mechanism, hence all atoms' borders are similar, denoted by 1; this border information is naturally extended to general planar words specifying the number of atoms laying on that border (rectangle's dimensions). For a planar word  $p$ , we let  $n(p)$ ,  $s(p)$ ,  $w(p)$ , and  $e(p)$  denote the dimension of its northern, southern, western, and eastern border, respectively.

The term *picture* is often used as a substitute for two-dimensional or planar word, especially in the context of *picture languages*. Useful surveys on two-dimensional languages may be found in [20], [15], or [24].

It is known that the following are equivalent for a planar language  $L$  (called *recognizable two-dimensional language*):

1.  $L$  is recognized by a *on-line tessellation automaton*;
2.  $L$  is defined by a *simple regular expression with intersection and adding homomorphisms*;
3.  $L$  is defined by a *local lattice language plus homomorphisms (tile system)*;
4.  $L$  is defined by an *existential monadic second order formula*.

The proof of this and many other informations may be found in [16], [15], [25], etc.<sup>4</sup>

However, the situation is more complex in this two-dimensional case. For instance, deterministic, nondeterministic, and alternating (4-way) finite automata have distinct increasing power, and the class of languages recognized by the last one is incomparable with the above class of recognizable two-dimensional language; see [22].

### 2.2.2. *Simple regular expressions*

We present here a simple extension of classical regular expressions [23, 11] to cope with planar words.

*Simple two-dimensional regular expressions.* The *signature* of simple two-dimensional regular expressions consists of two collections of usual regular operators, sharing the additive part. Specifically, it is

$$+, 0, \cdot, *, \uparrow, \triangleright, \dagger, -$$

where  $(+, 0, \cdot, *, \uparrow)$  is a usual Kleene signature to be used for the vertical dimension, while  $(+, 0, \triangleright, \dagger, -)$  is another Kleene signature to be used now for the horizontal dimension. Notice that, “0”, “ $\uparrow$ ” and “ $-$ ” are zero-ary operators (constants), “ $*$ ” and “ $\dagger$ ” are unary, while the remaining ones are binary.

Our default rule is that if no dimension is specified, that the vertical one is considered. Hence, usually written Kleene operators refer to the vertical axis: “ $\cdot$ ” is *(vertical) composition*, “ $*$ ” is *iterated (vertical) composition or (vertical) star* and “ $\uparrow$ ” is *(vertical) identity*. Similarly, “ $\triangleright$ ” is *horizontal composition*, “ $\dagger$ ” is *iterated horizontal composition or horizontal star* and “ $-$ ” is *horizontal identity*.

*Simple two-dimensional regular expressions* over an alphabet  $V$  (consisting of two-dimensional letters/atoms) are obtained starting from atoms

---

<sup>4</sup> Notice that regular expressions and tile systems do not directly cover the class of recognizable languages, but reach this using homomorphisms. This may be seen as an inconvenient of these mechanisms. On the other hand, this problem does not occur for on-line tessellation automata or for our interactive systems, as they are closed to homomorphisms.

and iteratively applying the operations in the described signature. Formally,

$$E ::= a \mid 0 \mid E + E \mid E \cdot E \mid E^* \mid | \mid E \triangleright E \mid E^\dagger \mid -$$

Their set is denoted by  $2\text{RegExp}(V)$ .

*From expressions to planar words.* To each expression in  $2\text{RegExp}(V)$  one may associate a language of planar words over  $V$ . To this end, we first describe how composition operations act on planar words and give the meaning of the identity operators.

If  $v, w$  are planar words, then their horizontal composition is defined only if  $e(v) = w(w)$  and the result  $v \triangleright w$  is the word obtained putting together  $v$  on the left and  $w$  on the right. Their vertical composition is defined only if  $s(v) = n(w)$  and  $v \cdot w$  is the word obtained putting  $v$  on top of  $w$ .

For each natural number  $k$  one may associate two “empty” planar words: the vertical identity  $\epsilon_k$  having  $w(\epsilon_k) = e(\epsilon_k) = 0$  and  $n(\epsilon_k) = s(\epsilon_k) = k$  and the horizontal identity  $\lambda_k$  with  $w(\lambda_k) = e(\lambda_k) = k$  and  $n(\lambda_k) = s(\lambda_k) = 0$ .

Now, the interpretation

$$| \mid : 2\text{RegExp}(V) \rightarrow \text{LangPlanarWords}(V)$$

from expressions to languages of planar words is defined by:

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

Before giving some examples, we define a useful flattening operator mapping languages of planar words to languages of usual, one-dimensional words.

*The flattening operator.* The flattening operator

$$\flat : \text{LangPlanarWords}(V) \rightarrow \text{LangWords}(V)$$

maps sets of planar words to sets of strings representing their topological sorting. In more details it is defined as follows:

- Each planar word may be considered as an acyclic directed graph drawing: (1) horizontal edges from each letter to its right neighbor, if this exists, and (2) vertical edges from each letter to its bottom neighbor, if this exists.

- Being acyclic, the starting graph associated to a planar word and all of its subgraphs have *minimal atoms/vertices*, i.e., vertices without incoming arrows.
- The topological sorting procedure selects a minimal vertex, then deletes it and its outgoing edges and repeats this as long as possible. This way a sequence (i.e., a usual one-dimensional word) containing the atoms of the planar word is obtained.
- Varying the minimal vertex selection in the topological sorting procedure in all possible ways one gets a set of words that is the value of the flattening operator applied to the planar word.
- Finally, to define  $b$  on a language  $L$ , take the union of  $b(w)$ , for  $w \in L$ .

To have an example, let us start with a planar word  $\begin{array}{c} abcd \\ efg h \end{array}$ ; there is

only one minimal element  $a$  and after its deletion we get  $\begin{array}{c} bcd \\ efg h \end{array}$ ; now there are two minimal elements  $b$  and  $e$  to choose from and suppose we choose  $b$ ; what remains is  $\begin{array}{c} cd \\ efg h \end{array}$ ; next, from the minimal elements  $c$  and  $e$ , we choose  $e$  and get  $\begin{array}{c} cd \\ fgh \end{array}$ ; and so on; finally a usual word, say  $abecfgdh$ , is obtained. Actually,

$$b\left(\begin{array}{c} abcd \\ efg h \end{array}\right) = \{abcdefgh, abcedfgh, abcefdgh, abcefgdh, abecd fgh, \\ abecfdgh, abecfgdh, abefcdgh, abefcgdh, aebcdfgh, \\ aebcf dgh, aebcf gdh, aebfcdgh, aebfcdgh\}$$

It is one of our main claims that this

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

And one of our main hopes is that

*the lifting of the verification techniques from usual words/paths to the planar version may avoid this problem*

*Convention:* we use terminal type letters  $a, b, \dots$  in planar words; they should be identified with the corresponding italic version, used elsewhere.

*Examples.* 1. The expression  $(a \cdot d^* \cdot g) \triangleright (b \cdot e^* \cdot h)^\dagger \triangleright (c \cdot f^* \cdot i)$  represents the language of planar words

$$\begin{array}{ccccccc} a & b & \dots & b & c & & \\ d & e & \dots & e & f & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & & \\ \cdot & \cdot & \cdot & \cdot & \cdot & & \\ d & e & \dots & e & f & & \\ g & h & \dots & h & i & & \end{array}$$

may be represented by the expression  $(a \triangleright b^\dagger \triangleright c) \cdot (d \triangleright e^\dagger \triangleright f)^* \cdot (g \triangleright h^\dagger \triangleright i)$ .

2. The language associated to  $a^\dagger \cdot b^\dagger$  consists of planar words  $\begin{array}{c} aa \dots a \\ bb \dots b \end{array}$ . Its flattened version  $\flat(a^\dagger \cdot b^\dagger) = \{w \in \{a, b\}^* : |w|_a = |w|_b \ \& \ \forall w = w'w'' : |w'|_a \geq |w'|_b\}$  is context-free, but not regular ( $|w|_a$  denotes the number of the occurrences of  $a$  in  $w$ ).

The slightly extended expression  $a^\dagger \cdot b^\dagger \cdot c^\dagger$  represents the language of planar words  $\begin{array}{c} aa \dots a \\ bb \dots b \\ cc \dots c \end{array}$ . Notice that its flattened version is  $\flat(a^\dagger \cdot b^\dagger \cdot c^\dagger) = \{w \in$

$\{a, b, c\}^* : |w|_a = |w|_b = |w|_c \ \& \ \forall w = w'w'' : |w'|_a \geq |w'|_b \geq |w'|_c\}$ . This latter language is even not context-free.

3. Notice that  $(a + b)^{\star\dagger} = (a + b)^\dagger{}^\star$ . This shows that vertical and horizontal stars may commute, providing simple atoms are involved.

On the other hand,  $\begin{array}{c} aa \\ bb \end{array} \in (a + b \triangleright b)^\dagger{}^\star \setminus (a + b \triangleright b)^{\star\dagger}$ , showing that, in general, the stars do not commute.

4. Finally, notice that  $a^\dagger \cdot b^\dagger \neq a^\star \triangleright b^\star$ , but  $\flat(a^\dagger \cdot b^\dagger) = \flat(a^\star \triangleright b^\star)$ . This example shows that the flattening mechanism may lose some information when passing from planar to usual languages.

### 2.2.3. Extended regular expressions

*More powerful iteration* The regular expressions previously described are simple and natural, but they have a number of shortcomings. For instance, their generative power is quite limited, comparing with finite interactive systems. In this subsection we roughly describe a few possible extensions based on a more powerful iteration mechanism.

A first possibility is to use an alternating horizontal/vertical concatenation within the iteration process. The separate vertical and horizontal iteration operators are not strong enough to represent the languages  $L_{sq}$  consisting of squares of  $a$ 's. But, starting with  $a$  and iterating a horizontal concatenation on the right with a vertical string  $a^\star$  and then a vertical concatenation on the bottom with a horizontal string  $a^\dagger \triangleright a$  one precisely gets the language of squares of  $a$ 's. In other words,  $L_{sq}$  is the least solution of the equation  $X = a + (X \triangleright a^\star) \cdot (a^\dagger \triangleright a)$ . (This type of iteration is studied in [27].)

Another example is given by the language  $L_{spir}$  of spiral words

x	2aa	2aaaa	...
	2x1	22aa1	
	bb1	22x11	
		2bb11	
		bbbb1	

A corresponding equation is  $X = x + (2 \triangleright a^\dagger) \cdot (2^* \triangleright X \triangleright 1^*) \cdot (b^\dagger \triangleright 1)$ . While in the above example only concatenation at the right and the bottom parts were used within the iteration process, in the present case concatenation on all sites were used within the iteration process.

*Space-time duality operator* Space-time duality operator “ $\$$ ” maps a planar word to its symmetric over the main diagonal.

The signature of Kleene algebra with space-time duality is:

$$(a, +, 0, \cdot, ^*, |, \$)$$

With space-time duality operator the horizontal operators are defined in terms of the vertical ones by:

$$\begin{aligned} v \triangleright w &= (v^\$ \cdot w^\$)^\$ \\ v^\dagger &= (v^{\$*})^\$ \\ - &= |^\$ \end{aligned}$$

Using a parallel with boolean algebra case, the former regular expressions are somehow similar to lattice expressions, while with this space-time duality operator they become more similar to boolean expressions.

*More constants* One may depart from the restriction to have only rectangular planar words by allowing some new constants, i.e., “ $\top$ ”, “ $\perp$ ”, “ $\dashv$ ”, or “ $\vdash$ ”. Their meaning may be obvious: they allow to initiate atoms from nothing (“ $\top$ ” and “ $\vdash$ ”) or to block their continuation (“ $\perp$ ” and “ $\dashv$ ”). One may be even more flexible and add a two-dimensional identity (crossing) “ $\oplus$ ”, or even constants “ $\neg$ ”, “ $\lrcorner$ ”, “ $\llcorner$ ”, or “ $\lrcorner$ ”. These latter constants may be used to switch data from time to space and vice-versa.

### 2.3. FINITE INTERACTIVE SYSTEMS

A finite interactive system may be seen as a kind of two-dimensional automaton mixing together a state-transforming machinery with a machinery used for the interaction of different threads of the first automaton.

While this way of presentation may be somehow useful, it is also misleading.

The first point is that there are not two different automata to be combined, but these two views are melted together to give this new concept of finite interactive systems.

A next critics is on using a term as ‘automaton’ for this device. While there may be some other interpretations, an automaton is generally considered as a state-transforming device. A state is a temporal section of a

running system: it gives the information related to the current values of the involved variables at a given temporal point. This information is used to compose the systems, to get more complex behaviours out of simpler ones. Then, using a term as ‘two-dimensional automaton’ is more or less as considering automata, but with a more complicate, say two-dimensional, state structure.

The situation with finite interactive systems is by far different: one (currently, the vertical) dimension is used to model this state transformation, but the other dimension is considered to be a behaviour transforming device. Orthogonal to the previous case of states, a behaviour gives a spatial section of the running system: it consider information on all the actions a spatially located agent has made during its evolution. This information is used at the temporal interfaces by which the agents interact, to get more complicate intelligent systems out of simpler ones. An agent is considered as a job-transforming device: given a job-request at its input temporal interface, the agent acts, transforms it, and passes the transformed job-result at its output temporal interface.

One feature which is implicitly present in this view is that the systems we are talking about are open: to run such a system one has to give an appropriate initial state, but also an initial job-request, actually a place where the starting user interacts with the system. The result of the running is a pair consisting of a final state of the system and a job-result (or a stream of messages) the system passes to an output user.

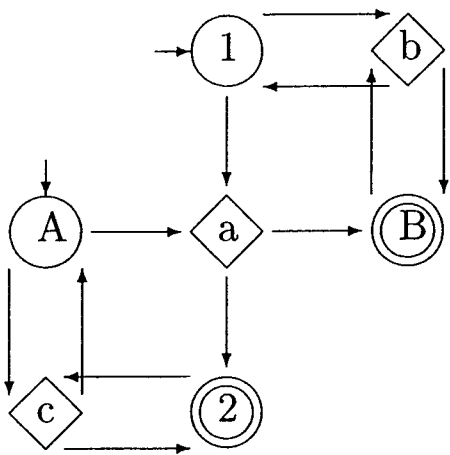
### 2.3.1. *Finite interactive systems*

After the above general considerations let us give a precise definition of finite interactive systems and explain it on a concrete example.

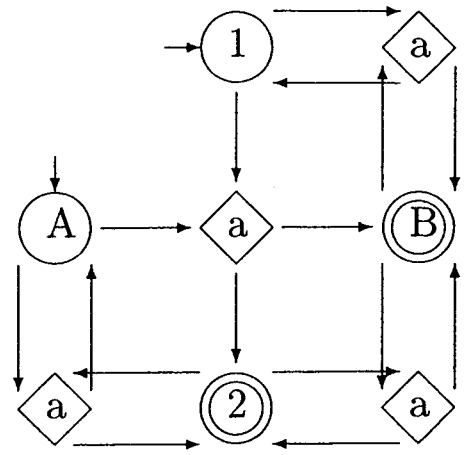
*Finite interactive systems.* A *finite interactive system* is a finite hypergraph with two types of vertices and one type of (hyper) edges:

- the first type of vertices is distinguished using a labeling with numbers (or lower case letters); such a vertex denotes a *state* (memory of variables);
- the second type of vertices is distinguished using a labeling with capital letters; such a vertex is considered to be a *class* (of job requests);
- the actions/transitions are labeled by letters denoting atoms of planar words and are such that: (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.

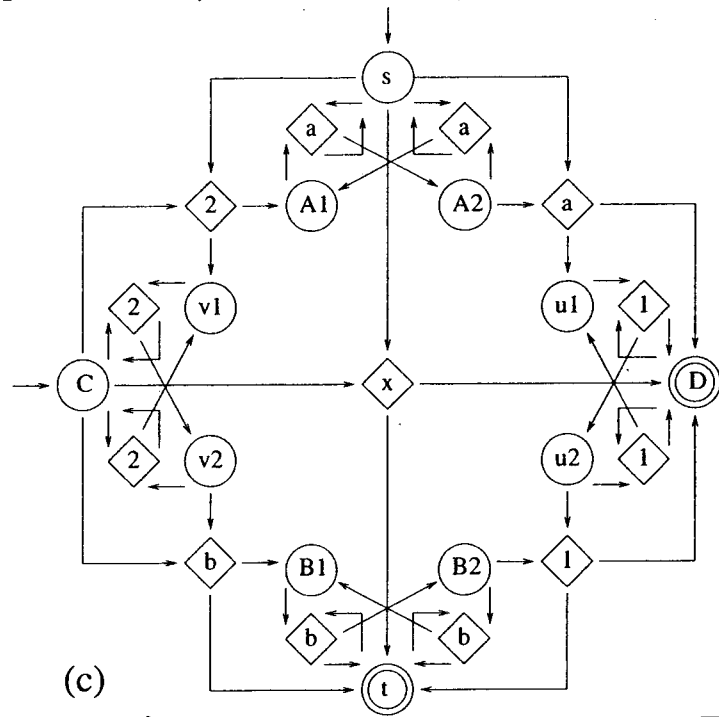
Some classes/states may be *initial* (in the graphical representation this situation is specified by using small incoming arrows) or *final* (in this case the double circle representation is used).



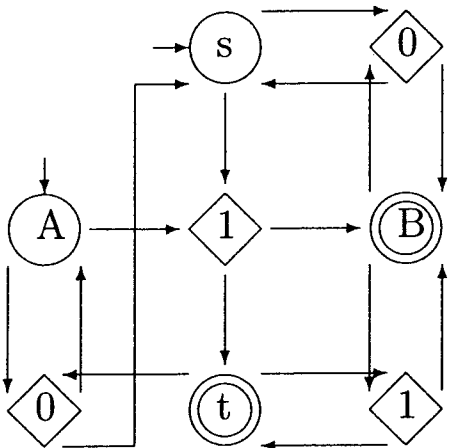
(a) *Sq* (for square words)



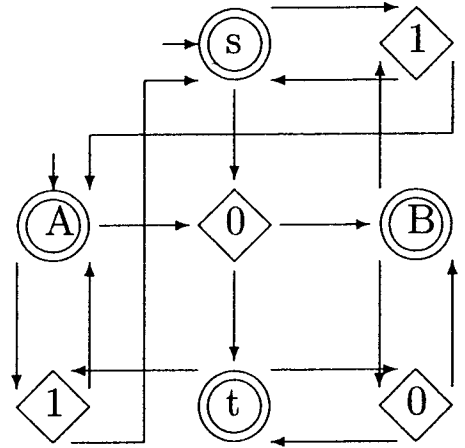
(b) *Sq2* (for square words)



(c)



(d) *Pow* (for  $2^n - 1$ )



(e) *Pas* (for Pascal triangle)

Figure 1. Examples of finite interactive systems

An example of a finite interactive system is given in Fig. 1(a). This finite interactive system has four vertices: two classes  $A$  and  $B$  and two states 1 and 2; moreover,  $A$  is an initial class,  $B$  is a final class, 1 is an initial state, and 2 is a final state. It also has three transitions labeled by  $a$ ,  $b$ , and  $c$ .

Sometimes it may be more convenient to have a simple non-graphical representation of an interactive system. One possibility is to represent a finite interactive system by its transitions and to specify which states/classes are initial or final. (In this 'cruz' representation of transitions the incoming vertices are those placed at north and west, while the outgoing ones are those from east and south.) For the given example, the finite interactive system is represented by:

—  $\begin{array}{|c|c|c|} \hline & 1 & \\ \hline A & a & B \\ \hline & 2 & \\ \hline \end{array}$ ,  $\begin{array}{|c|c|c|} \hline & 2 & \\ \hline A & c & A \\ \hline & 2 & \\ \hline \end{array}$ , and  $\begin{array}{|c|c|c|} \hline & 1 & \\ \hline B & b & B \\ \hline & 1 & \\ \hline \end{array}$  and the information that

—  $A, 1$  are initial states/classes and  $B, 2$  are final.

*Running patterns, accepting criteria.* Finite interactive systems may be used to *recognize* planar words. The procedure is the following:

(1) Suppose we are given: a finite interactive system  $S$ ; a planar word  $w$  with  $m$  lines and  $n$  columns; a horizontal string with initial states  $b_n = s_1 \dots s_n$ ; and a vertical string with initial classes  $b_w = C_1 \dots C_m$ . Then:

- Insert the states  $s_1, \dots, s_n$  at the northern border of  $w$  (from left to right) and the classes  $C_1, \dots, C_m$  at the western border of  $w$  (from top to bottom).
- Parse the planar word  $w$  selecting minimal<sup>5</sup> still unprocessed atoms.
- For each such minimal unprocessed atom  $a$ , suppose  $s$  is the state inserted at its northern border and  $C$  is the class inserted at its western border. Then, choose a transition  $\begin{array}{|c|c|c|} \hline & s & \\ \hline C & a & C' \\ \hline & s' & \\ \hline \end{array}$  from  $S$  (if any), insert the state  $s'$  at the southern border of  $a$  and the class  $C'$  at its eastern border, and consider this atom to be already processed.
- Repeat the above two steps as long as possible.

If finally all the atoms of  $w$  were processed, then look at the eastern border  $b_e$  and the southern border  $b_s$  of the result. If they contain only final classes and final states, respectively, then consider this to be a *successful running* for  $w$  with respect to the border conditions  $b_n, b_w, b_e, b_s$ .

(2) Given a finite interactive system  $S$  and four regular languages  $B_n, B_w, B_e,$  and  $B_s$  for the border conditions, a planar word  $w$  is *recognized* by

<sup>5</sup> The notion of minimal atoms we are using here is the one described when the flattening operator has been introduced.

$S$  with respect to the border conditions  $B_n, B_w, B_e, B_s$  if there exists border strings  $b_n \in B_n, b_w \in B_w, b_e \in B_e, b_s \in B_s$  and a successful running for  $w$  with respect to the border conditions  $b_n, b_w, b_e, b_s$ ; let us denote by  $L(S; B_n, B_w, B_e, B_s)$  their set.

(3) Finally, a language of planar words  $L$  is *recognized* by  $S$  if there are some regular languages  $B_n, B_w, B_e, B_s$  for the border conditions such that  $L = L(S; B_n, B_w, B_e, B_s)$ .

(4) We simply denote the associated language by  $L = L(S)$  when the border conditions are superfluous, i.e., the border languages are the corresponding full languages and actually no border condition is imposed.

*Example.* A concrete example may be useful. Let us look at the finite interactive system  $Sq$  in Fig. 1(a) and suppose that the border conditions are irrelevant, i.e.,  $B_n, B_w, B_e$  and  $B_s$  are the full languages  $\{1\}^*, \{A\}^*, \{B\}^*$  and  $\{2\}^*$ , respectively. A running is:

abb	1 1 1	1 1 1	1 1 1	1 1 1	...	1 1 1
cab	Aa b b	AaBb b	AaBbBb	AaBbBb	...	AaBbBbB
cca	Ac a b	Ac a b	Ac a b	AcAa b	...	AcAaBbB
	Ac c a	Ac c a	Ac c a	Ac c a	...	AcAcAaB
	2	2	2	2	...	2 2 2
$w$	$w_0$	$w_1$	$w_2$	$w_3$	...	$w_9$

The above sequence starts with a planar word  $w$ . (1) The word obtaining bordering  $w$  with initial states/classes is  $w_0$ . (2) At this stage all atoms are unprocessed, but only one is minimal, i.e., the  $a$  in the left-upper corner.

In  $S$  there is only one transition of the type  $\begin{array}{|c|c|c|} \hline & 1 & \\ \hline A & a & \\ \hline & & \\ \hline \end{array}$ , i.e.,  $\begin{array}{|c|c|c|} \hline & 1 & \\ \hline A & a & B \\ \hline & & 2 \\ \hline \end{array}$ . After

its application one gets  $w_1$ . (3) Now there are two minimal unprocessed elements:  $b$  (position (1,2) in  $w$ ) and  $c$  (position (2,1) in  $w$ ). Suppose we

choose  $b$ . Again, there is only one appropriate transition  $\begin{array}{|c|c|c|} \hline & 1 & \\ \hline B & b & B \\ \hline & & 1 \\ \hline \end{array}$ . After its

application one gets  $w_2$ . (4) Next, the minimal unprocessed elements are  $b$  (position (1,3) in  $w$ ) and  $c$  (position (2,1) in  $w$ ). Suppose we choose  $c$  and

the corresponding unique transition  $\begin{array}{|c|c|c|} \hline & 2 & \\ \hline A & c & A \\ \hline & & 2 \\ \hline \end{array}$ . Then we get  $w_3$ . (5) We can

continue and finally an accepting running is obtained, leading to  $w_9$ .

One may easily see that the recognized language  $L(S)$  consists of *square words with a's on the main diagonal, the top-right half filled in with b's and the bottom-left half filled in with c's.*

Finally, one may look at some border conditions. There is only one initial/final state/class, hence the border languages are regular languages over a one letter alphabet. To have an example, if  $B_n = (111)^*$ ,  $B_w = (AAAA)^*$ ,  $B_e = B^*$  and  $B_s = (22)^*$ , then  $L(S; B_n, B_w, B_e, B_s)$  is the sub-language of the above  $L(S)$  consisting of those squares whose dimension  $k$  is a multiple of 12.

*State projection and class projection.* Some familiar nondeterministic finite automata (nfa's) may be obtained from finite interactive systems neglecting one dimension.

The automaton obtained from a finite interactive system  $S$  neglecting the class transforming part is called the *state projection nfa* associated to  $S$  and it is denoted by  $s(S)$ . In UML this is known as the behaviour/statechart diagram.

Similarly, the *class projection nfa*, denoted  $c(S)$ , is the nfa obtained neglecting the state transforming part of  $S$ . UML use a similar class/interaction diagram name for this view of the system.

The class projection nfa  $c(S)$  may be used to define *macro-step transitions* in the original finite interactive system  $S$ . Dually, the state projection nfa  $s(S)$  may be used to define *agent behaviours* (seen as job class transformers) in the original finite interactive system.

Let us return to the finite interactive system of Fig. 1(a). Its class projection nfa  $c(S)$  is given by the transitions  $A \xrightarrow{a} B$ ,  $A \xrightarrow{c} A$ ,  $B \xrightarrow{b} B$  with  $A$  initial and  $B$  final. The state projection nfa  $s(S)$  is defined by transitions  $1 \xrightarrow{a} 2$ ,  $2 \xrightarrow{c} 2$ ,  $1 \xrightarrow{b} 1$  with 1 initial and 2 final.

This particular finite interactive system has an interesting *intersection property*:

$$L(S) = L(s(S)) \cap L(c(S))$$

showing that, in some sense, the interaction between the state and the class transformations of  $S$  is not so strong.<sup>6</sup>

---

<sup>6</sup> To avoid some confusions, we emphasize here that the usual languages associated to the projection nfa's are extended to the planar case. For instance, a planar word  $w$  is in  $L(s(S))$  if every column in  $w$  is recognized by  $s(S)$  in the usual sense. Similarly for  $L(c(S))$ , but now the lines are taken into account.

### 2.3.2. Examples

1. *Square words.* Let us denote by  $L_{sq}$  the language consisting of square words of  $a$ 's, namely

$a$	$aa$	$aaa$	$aaaa$	$\dots$
	$aa$	$aaa$	$aaaa$	
		$aaa$	$aaaa$	
			$aaaa$	

An example of a finite interactive system recognizing  $L_{sq}$  is  $Sq$  illustrated in Fig. 1(a), where  $b$  and  $c$  are to be replaced by  $a$ .<sup>7</sup> Two examples of running patterns for  $Sq$  are shown below (the first one is successful; the second one is unsuccessful, as the southern border contains an occurrence of the non-final state 1):

$1$	$1$	$1$	$1$	$1$	$1$
$AaBaBaB$	$AaBaBaB$	$AaBaBaB$	$AaBaBaB$	$AaBaBaB$	$AaBaBaB$
$2$	$1$	$1$	$1$	$1$	$1$
$AaAaBaB$	$AaAaBaB$	$AaAaBaB$	$AaAaBaB$	$AaAaBaB$	$AaAaBaB$
$2$	$2$	$1$	$1$	$1$	$1$
$AaAaAaB$	$AaAaAaB$	$AaAaAaB$	$AaAaAaB$	$AaAaAaB$	$AaAaAaB$
$2$	$2$	$2$	$2$	$2$	$1$

### 2. Exponential function.

Another finite interactive system is  $Pow$ , drawn in Fig. 1(d). It recognizes the language

$1$	$10$	$100$	$\dots$
	$01$	$010$	
	$11$	$110$	
		$001$	
		$101$	
		$011$	
		$111$	

(a running)

$s$	$s$	$s$	$s$	$s$	$s$
$A$	$1$	$B$	$0$	$B$	$0$
$t$	$s$	$t$	$s$	$t$	$s$
$A$	$0$	$A$	$1$	$B$	$0$
$s$	$t$	$s$	$t$	$s$	$t$
$A$	$1$	$B$	$1$	$B$	$0$
$t$	$t$	$s$	$s$	$t$	$s$
$A$	$0$	$A$	$0$	$A$	$1$
$s$	$s$	$t$	$t$	$s$	$t$
$A$	$1$	$B$	$0$	$B$	$1$
$t$	$s$	$t$	$s$	$t$	$s$
$A$	$0$	$A$	$1$	$B$	$1$
$s$	$t$	$t$	$t$	$s$	$t$
$A$	$1$	$B$	$1$	$B$	$1$
$t$	$t$	$t$	$t$	$t$	$t$

### 3. Pascal triangle.

The new finite interactive system  $Pas$  in Fig. 1(e) recognizes the set of words over  $\{0, 1\}$  whose northern and western borders have alternating sequences of 0 and 1 (starting with 0) and, along the secondary diagonals, it satisfies the recurrence rule of the Pascal triangle, modulo 2 (that is, the value in a cell  $(i, j)$  is the sum of the values of the cells  $(i-1, j)$  and  $(i, j-1)$ , modulo 2).

(a running)

$s$	$s$	$s$	$s$	$s$	$s$	$s$	$s$
$A$	$0$	$B$	$1$	$A$	$0$	$B$	$1$
$t$	$s$	$t$	$s$	$t$	$s$	$t$	$s$
$A$	$1$	$A$	$0$	$B$	$0$	$B$	$1$
$s$	$t$	$t$	$s$	$s$	$t$	$s$	$t$
$A$	$0$	$B$	$0$	$B$	$1$	$A$	$0$
$t$	$t$	$t$	$s$	$t$	$t$	$t$	$t$
$A$	$1$	$A$	$1$	$A$	$1$	$A$	$0$
$s$	$s$	$s$	$t$	$t$	$t$	$t$	$t$
$A$	$0$	$B$	$1$	$A$	$0$	$B$	$0$
$t$	$s$	$t$	$t$	$t$	$t$	$t$	$t$
$A$	$1$	$A$	$0$	$B$	$0$	$B$	$0$
$s$	$t$	$t$	$t$	$t$	$t$	$t$	$t$

In this case, one can see that the projection automata (on states and classes, respectively) are reset automata having all states/classes final,

<sup>7</sup> The same language is recognized by the finite interactive system  $Sq2$ , drawn in the same figure.

hence they recognize the corresponding full languages. Their intersection is now the full language of two-dimensional words, very far from the language recognized by *Pas*. This may be interpreted as a strong interaction between the state transformation and the class transformation within *Pas*. It also shows that the ‘intersection’ property is not generally valid for the full class of finite interactive systems.

4. *Spiral words*. The finite interactive system drawn in Fig. 1(c) recognizes the language of ‘spiral’ words

x	2aa	2aaaa	...
	2x1	22aa1	
	bb1	22x11	
		2bb11	
		bbbb1	

### 2.3.3. Finite interaction system languages

*Local Lattice Languages* (LLL’s, for short), are specified using a finite set of finite (rectangular) words that are allowed as sub-words of the recognized planar words. If one combines this with (letter-to-letter) homomorphisms, then the power of the mechanism does not change by restricting to  $2 \times 2$  pieces, or even to horizontal and vertical domino-s (i.e.,  $1 \times 2$  and  $2 \times 1$  pieces). These are also called *tile systems*.

It is easy to show that (1) one may simulate a LLL by a finite interactive system and (2) the class of languages recognized by finite interactive systems is closed under letter-to-letter homomorphisms. From this simple observations it follows that the class of languages recognized by finite interactive systems contains the class of recognizable planar languages. The converse inclusion is also valid: (1) one may see that the runnings of a finite interaction system  $A$ , completed with a special letter for the blank space, are locally testable, hence form a LLL; then (2) using an homomorphism one may abstract from the state/class/blank information around the letters, to go to  $L(A)$ , hence  $L(A)$  is a recognizable picture language.

To conclude, finite interactive systems give a different representation for recognizable picture languages, adding more arguments that this class of two-dimensional languages is a robust one.

From this pleasant but unexpected connection one inherits many results known for picture languages. One important fact is the undecidability of the emptiness problem for finite interactive system.<sup>8</sup>

<sup>8</sup> This follows from the known fact that if one restrict himself to the top line of the pictures in a recognizable picture language, then one gets precisely context-sensitive (string) languages.

## 2.4. COOP PSEUDO-CODE PROGRAMS

The finite interactive system in Fig. 1(a) may be used to describe, in pseudo-code, an algorithm for solving the  $8$ -queen problem. The problem is to place queens on a chess table such that no queen may attack another queen. For convenience, we replace the transition labels from  $a, b$ , and  $c$  to  $av$  (advance and place queen),  $r$  (report on a successful attempt) and  $c$  (curry on the placement information), respectively.

The interpretation is as follows (from this interpretation one also gets the information regarding the data structures used on the atoms' borders):<sup>9</sup>

- $r$ : if ( $w = ok$  and ( $n = ok$  or  $n = new$ )),  
 then ( $s = ok$  and  $e = ok$ ),  
 otherwise ( $s = no$  and  $e = no$ );
- $c$ : if ( $n = pos_k$  and  $w = \langle pos_1, \dots, pos_{k-1} \rangle$ ),  
 then ( $s = pos_k$  and  $e = \langle pos_1, \dots, pos_{k-1}, pos_k \rangle$ );
- $av$ : if ( $w = \langle pos_1, \dots, pos_k \rangle$  and ( $n = ok$  or  $n = new$ )  
 and  $free(pos_1, \dots, pos_k) \neq \emptyset$ ),  
 then (select  $p_{k+1} \in free(pos_1, \dots, pos_k)$  and make  
 $s = pos_{k+1}$  and  $e = ok$ ),  
 otherwise ( $s = random$  and  $e = no$ );

Two running patterns are given below

		*	
*			
			*
	*		

$\langle \rangle$	$\overset{new}{\boxed{av}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$
	2		OK		OK		OK	
$\langle \rangle$	$\boxed{c}$	$\langle 2 \rangle$	$\overset{new}{\boxed{av}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$
	2		4		OK		OK	
$\langle \rangle$	$\boxed{c}$	$\langle 2 \rangle$	$\boxed{c}$	$\langle 2, 4 \rangle$	$\overset{new}{\boxed{av}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$
	2		4		1		OK	
$\langle \rangle$	$\boxed{c}$	$\langle 2 \rangle$	$\boxed{c}$	$\langle 2, 4 \rangle$	$\boxed{c}$	$\langle 2, 4, 1 \rangle$	$\overset{new}{\boxed{av}}$	$OK$
	2		4		1		3	

and

*			
	*		

$\langle \rangle$	$\overset{new}{\boxed{av}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$
	1		OK		OK		OK	
$\langle \rangle$	$\boxed{c}$	$\langle 1 \rangle$	$\overset{new}{\boxed{av}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$	$\overset{new}{\boxed{r}}$	$OK$
	1		3		OK		OK	
$\langle \rangle$	$\boxed{c}$	$\langle 1 \rangle$	$\boxed{c}$	$\langle 1, 3 \rangle$	$\overset{new}{\boxed{av}}$	$NO$	$\overset{new}{\boxed{r}}$	$NO$
	1		3		?		NO	
$\langle \rangle$	$\boxed{c}$	$\langle 1 \rangle$	$\boxed{c}$	$\langle 1, 3 \rangle$	$\boxed{c}$	$\langle 1, 3, ? \rangle$	$\overset{new}{\boxed{av}}$	$NO$
	1		3		?		??	

<sup>9</sup> Notice that in order to have defined the  $free$  predicate, one need to know the chess table dimension.

## 2.5. CLOSING COMMENTS

The concept of finite interactive systems presented here has evolved from a study of Mixed Network Algebras - see, e.g., [34], [18], [17], [36], or the final part of the recent book [35].<sup>10</sup>

Our finite interactive systems may be seen as an extension of existing models like finite transition systems [1], Petri nets [14], data-flow networks [5], or asynchronous automata [37], in the sense that a potentially unbound number of processes may interact. In the mentioned models, while the number of processes may be unbounded, their interaction is bounded by the transitions' breadth. This seems to be a key feature for the ability to pass from concurrent to concurrent, object-oriented agents.

Our model has some similarities with the tile model studied by Montanari and his Pisa group (see, e.g., [6], [7], or [13]), but the precise relationship still has to be investigated. We also expect that a logic similar to the spatial logic of [8] may be associated to finite interactive systems and use for the verification of concurrent, object-oriented systems.

*Acknowledgment* I acknowledge with thanks many useful discussions I had on various mixed network algebra models (including the present finite interaction system model) with various people including M. Broy, C. Dima, R. Grosu, Y. Kawahara, U. Montanari, D. Lucanu, S. Merz, R. Soriccut, A. Stefanescu, B. Warinschi, and M. Wirsing.

## References

1. Arnold, A. (1994). *Finite transition systems*. Prentice-Hall International.
2. Baeten, J.C.M. and Weijland, W.P. (1990). *Process Algebra*. Cambridge University Press.
3. Bloom, S.L. and Esik, Z. (1993). *Iteration Theories: The Equational Logic of Iterative Processes*. Springer-Verlag.
4. Booch, G., Rumbaugh, J., and Jacobson, I. (1999) *The Unified Modeling Language User Guide*. Addison Wesley.
5. Broy, M. and Stefanescu, G. (2001). The algebra of stream processing functions. *Theoretical Computer Science*, 258, 95–125.
6. Bruni, R. (1999). Tile Logic for Synchronized Rewriting of Concurrent Systems. PhD thesis, Dipartimento di Informatica, Universita di Pisa. Report TD-1/99.
7. Bruni, R. and Montanari, U. (1997). Zero-safe nets, or transition synchronization made simple. *Electronic Notes in Theoretical Computer Science*, vol. 7(20 pages).

---

<sup>10</sup> In [36] a preliminary more complicated version of the current model was presented. It uses two-dimensional queues associated with states/classes in order to model more complicated running scenarios for concurrent object-oriented systems. The present model of finite interactive systems is simpler, but still raises many difficult mathematical problems.

8. Cardeli, L. and Gordon, A. (2000). Anytime, anywhere: modal logics for mobile ambients. In: *POPL-2000, Symposium on Principles of Programming Languages*, Boston, 2000. ACM Press.
9. Cazanescu, V.E. and Stefanescu, G. (1990). Towards a new algebraic foundation of flowchart scheme theory. *Fundamenta Informaticae*, 13, 171–210.
10. Cazanescu, V.E. and Stefanescu, G. (1994). Feedback, iteration and repetition. In Gh. Păun, editor, *Mathematical aspects of natural and formal languages*, pages 43–62. World Scientific, Singapore.
11. Conway, J.H. (1971). *Regular Algebra and Finite Machines*. Chapman and Hall.
12. Elgot, C.C. (1975). Monadic computation and iterative algebraic theories. In H.E. Rose and J.C. Sheperdson, editors, *Proceedings of Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 175–230. North-Holland.
13. Gadducci, F. and Montanari, U. (1996). The tile model. In *Papers dedicated to R. Milner festschrift*. The MIT Press, Cambridge, to appear, 1999. Also: Technical Report TR-96-27, Department of Computer Science, University of Pisa.
14. Garg, V. and Ragnath, M.T. (1992). Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Science*, 96, 285–304.
15. Giammarresi, D. and Restivo A. (1997). Two-dimensional languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages. Vol. 3: Beyond words*, pages 215–265. Springer-Verlag.
16. Giammarresi, D., Restivo, A., Seibert, S., and Thomas W. (1996). Monadic second order logic over rectangular pictures and recognizability by tiling systems. *Information and Computation*, 125, 32–45.
17. Grosu, R., Lucanu, D., and Stefanescu G. (2000). Mixed relations as enriched semiringal categories. *Journal of Universal Computer Science*, 6(1), 112–129.
18. Grosu, R., Stefanescu, G., and Broy, M. (1998). Visual formalism revised. In *Proceeding of the CSD'98 (International Conference on Application of Concurrency to System Design, March 23-26, 1998, Fukushima, Japan)*, pages 41–51. IEEE Computer Society Press.
19. Hennessy, M. (1988). *Algebraic theory of processes*. Foundations of Computing. The MIT Press, Cambridge.
20. Inoue, K. and Takanami, I. (1991). A survey of two-dimensional automata theory. *Information Sciences*, 55, 99–121.
21. Joyal, A., Street, R., and Verity, D. (1996). Traced monoidal categories. *Proceedings of the Cambridge Philosophical Society*, 119, 447–468.
22. Kari, J. and Moore, C. (2000). New results on alternating and non-deterministic two-dimensional finite-state automata. In: *Proceedings STACS 2001*, LNCS 2010, 396–406, Springer-Verlag.
23. Kleene, S.C. (1956). Representation of events in nerve nets and finite automata. In: C.E. Shannon and J. McCarthy, eds., *Automata Studies, Annals of Mathematical Studies*, vol. 34, 3–41. Princeton University Press.
24. Lindgren, K., Moore, C., and Nordahl, M. (1998). Complexity of two-dimensional patterns. *Journal of Statistical Physics* 91, 909–951.
25. Latteux, M. and Simplot, D. (1997). Recognizable picture languages and domino tiling. *Theoretical Computer Science* 178, 275–283.
26. Manes, E.G. and Arbib, M.A. (1986). *Algebraic approaches to program semantics*. Springer-Verlag.

27. Matz, O. (1997). *Regular expressions and context-free grammars for picture languages*. In: *Proceedings STACS'97*, LNCS 1200, 283–294, Springer-Verlag.
28. Meseguer, J. and Montanari, U. (1990). Petri nets are monoids. *Information and Computation*, 88, 105–155.
29. Milner R. (1989). *Communication and concurrency*. Prentice-Hall International.
30. Petri, C.A. (1962). *Kommunikation mit Automaten*. PhD thesis, Institute für Instrumentelle Mathematik, Bonn, Germany.
31. Stefanescu, G. (1987). On flowchart theories: Part I. The deterministic case. *Journal of Computer and System Sciences*, 35, 163–191.
32. Stefanescu, G. (1987). On flowchart theories: Part II. The nondeterministic case. *Theoretical Computer Science*, 52, 307–340.
33. Stefanescu, G. (1986). Feedback theories (a calculus for isomorphism classes of flowchart schemes). Preprint Series in Mathematics No. 24, National Institute for Scientific and Technical Creation, Bucharest. Also in: *Revue Roumaine de Mathematiques Pures et Applique*, 35, 73–79, 1990.
34. Stefanescu, G. (1998). Reaction and control I. Mixing additive and multiplicative network algebras. *Logic Journal of the IGPL*, 6(2), 349–368.
35. Stefanescu, G. (2000). *Network algebra*. Springer-Verlag.
36. Stefanescu, G. (2001). Kleene algebras of two dimensional words: A model for interactive systems. *Dagstuhl Seminar on "Applications of Kleene algebras"*, Seminar 01081.
37. Zielonka, W. (1987). Notes on finite asynchronous automata. *Theoretical Informatics and Applications*, 21, 99–135.