A New Representation of Two-Dimensional Patterns and Applications to Interactive Programming

I.T. Banu-Demergian, C.I. Paduraru, G. Stefanescu

A new representation of two-dimensional patterns and applications to interactive programming

I.T. Banu-Demergian[1], C.I. Paduraru[1], and G. Stefanescu[1]

Department of Computer Science, University of Bucharest, Romania
th_iulia84@yahoo.com, ciprian.paduraru2009@gmail.com,
and gheorghe.stefanescu@fmi.unibuc.ro

Abstract. Regular expressions and the associated regular algebra provide a rich formalism for specifying and analysing sequential models of computation. For parallel computation, extensions to handle two-dimensional patterns are often required. In this paper we present a new type of regular expressions for two-dimensional patterns based on contours and their composition. Targeted applications comes from the area of modelling, specification, analysis and verification of structured interactive programs via the associated scenario semantics.

Keywords: regular expressions, two-dimensional patterns, contours, structured interactive programming, formal methods

1 Introduction

Regular expressions and the associated regular algebra provide a rich formalism for specifying and analysing sequential models of computation. They were originally introduced by Kleene [17] in connection with neural networks and finite automata - Kleene theorem states that finite automata and regular expressions are equivalent (i.e., they specify the same language). In the meantime, regular expressions became a core formalism for many other models used in computer science. In particular, they provide the backbone of a rich algebraic theory of automata, see, e.g. [28, 11, 20, 18, 6, 19, 8].

For parallel computation, enrichment of the sequential models with mechanisms for modelling process interaction are needed. We only mention a Kleene theorem to Petri nets [27, 13]: Petri nets and a class of concurrent regular expressions are equivalent. The result is based on the following procedure: (1) decompose the behaviour to have separate components where each transaction has no more than one input and one output place; (2) decompose the behaviour of a component to have an independent run for each initial token; (3) use the classical Kleene theorem for these sequential runs; (4) use synchronization and renaming to force the composition of these separate projected runs to behave as a run of the initial overall system. However, as it was often noticed (see, e.g., [18]), renaming has bad algebraic properties and should be avoided.

A natural semantics for parallel computation is provided by a kind of two-dimensional patterns/languages - for instance, messages sequence charts or scenarios fall into this category. A robust class of "regular two dimensional languages" has been identified in 1990's [14, 22]; it may be specified by many equivalent formalisms, in particular by a 2-dimensional version of regular expressions 2RE, including intersection and renaming.

In this paper we present a new type of regular expressions for two-dimensional patterns n2RE based on contours and their composition. It avoids the use of intersection and renaming, being closer in spirit with classical 1-dimensional regular expressions. In this approach, the magic way of getting the intended language by renaming and intersection is replaced by a steady work of tiling shapes to build up the words step by step.

Our targeted applications comes from the area of modelling, specification, analysis and verification of structured interactive programs via the associated scenario semantics. Interactive computation [15] is becoming more and more important in the recent years, in particular due to the advance of multicore computation. We use a model rv-IS [35] based on space-time duality. In particular, finite interactive systems [34] are the space-time invariant extension of finite automata in this context. Agapia programming [12] is a core interactive programming language based on this model. We use the n2RE expressions to present a relational semantics for Agapia programs; it may be seen as an extension to two dimensions of the classical relational semantics of sequential computing models [23, 24].

The paper is organized as follows. Section 2 presents a known approach using two sets of regular algebra operators, intersection, and renaming. Section 3 presents the new approach based on contours and Section 4 shows an application for getting a relational semantics for structured interactive programs. Related and future works and references conclude the paper.

2   A known approach

2.1   Finite interactive systems (FIS's) and regular expressions (2RE's)

Definition 1. A *finite interactive system (FIS)* [34, 35] is defined by

- two types of nodes: *states* (denoted by numbers $1, 2, ...$) and *classes* (denoted by capital letters $A, B, ...$);
- *transactions*: $(A, 1) - a \rightarrow (B, 2)$, where a is a letter of the considered alphabet and $A, B, 1, 2$ are as above;
- specification of the *initial/final* states and classes. □

A useful *cross/tile representation* may be used; is is based on showing the transitions and stating which states and classes are initial/final. An example is

$S1$: $\boxed{\begin{array}{c}1\\ A\,a\,B\\ 2\end{array}}$ $\boxed{\begin{array}{c}2\\ A\,c\,A\\ 2\end{array}}$ $\boxed{\begin{array}{c}1\\ B\,b\,B\\ 1\end{array}}$ with $1, A$ initial and $2, B$ final.

rectangular word   accepting $S1$ scenario   general word   accepting $S1$ scenario
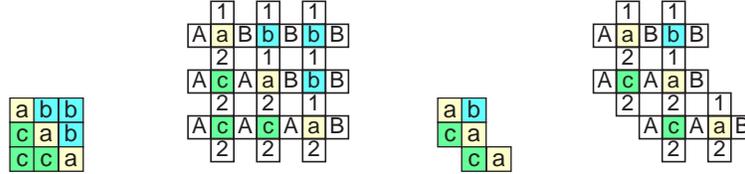
Fig. 1. FIS recognizing procedure

FIS recognizing procedure. The *FIS recognizing procedure* is via accepted scenarios. A *scenario* alternates class/state information and letters according to the FIS transitions. It is an *accepting scenario* if the northern border has initial states, the western border has initial classes, the eastern border has final classes, and the southern border has final states.

Graphically, a scenario may be easily obtained using the crosses representing the transitions and identifying the matching classes or states of the neighbouring cells. In Fig. 1 we show a few examples of scenarios for the FIS $S1$ above. Notice that the recognizing procedure may be applied to non-rectangular words, as well.

Definition 2. First, *simple 2-dimensional regular expressions* (simple 2RE's) are defined by two sets of regular operators (one for the vertical, the other for the horizontal direction) which share the additive part. Formally, they use:

1. the *additive operators*: O (for *empty set*) and + (for *union*);
2. the *vertical composition operators*: I_v (*vertical identity*), ;_v (*vertical composition*) and *_v (*iterated vertical composition*); our preferred textual notation is: |, ; and *;
3. the *horizontal composition operators*: I_h (*horizontal identity*), ;_h (*horizontal composition*) and *_h (*iterated horizontal composition*); our preferred textual notation is: -, > and ^.

Next, *2-dimensional regular expressions (2RE's)* are obtained adding intersection and renaming to simple 2RE's. Formally, they use the following additional operators

1. *intersection*: our preferred textual notation is /\;
2. *renaming* via a letter-to-letter homomorphism rho:V->V' (V and V' are the old and the new vocabulary, respectively). □

Examples. A few examples of 2RE expressions and typical specified words are presented in Fig. 2. They are related to the following expression

    E = (b* ; a ; c*)^ /\ (c^ > a > b^)*.

In Fig. 2.(1)-(4), typical words generated by the following expressions are presented: b* ; a ; c*; (b* ; a ; c*)^; c^ > a > b^; and (c^ > a > b^)*. It can be proved that the intersection (b* ; a ; c*)^ /\ (c^ > a > b^)* has only square words with a on the diagonal, b on the top right area and c on

the bottom left area, see Fig. 2.(5). The first part of the expression constrains the column patterns, while the second the rows. Intersection does the magic action of selecting only the square 2-dimensional words.
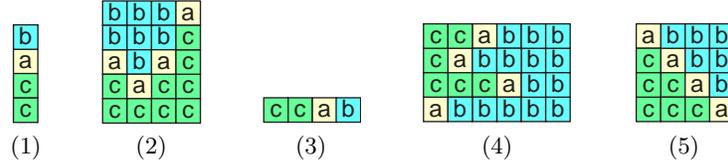

**Fig. 2.** A 2RE expression for the FIS S1

Theorem 1. *(connecting 2RE's to FIS's [14, 34, 35, 29]) The languages represented by finite interactive systems (FIS's) and those specified by 2-dimensional regular expressions (2RE's) are the same.*

Proof: (Sketch): As usual, more complicate is the passing from FIS's to 2RE's. It is done in two steps:

- for a FIS S with transitions having distinct letters the procedure is:
  - take a usual regular expression Es for the state-projected nondeterministic finite automaton (NFA) of S and another one Ec for the class-projected NFA of S (these NFA's are obtained from S ignoring one dimension)
  - an expression for S is (Es)^ /\ (Ec)*
- for an arbitrary FIS proceed as follows: first rename the transitions with new letters to apply the above step; then, apply the previous result; finally, use the renaming operator to rename back the new letters with their original version in the resulting expression.

For the other part, one can use the result in [29] showing that FIS's are equivalent with tile systems, then the relation between 2RE's and tile systems. □

Example: Let us consider the FIS S2 defined by

, where 1,A are initial and 2,B are final.
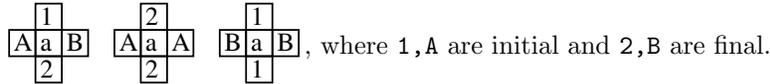
The procedure is the following:

1. rename the 2-nd a as c and the 3-rd a as b to get different letters for transitions; actually, this way we get the FIS S1 above;
2. get a 2RE for this new FIS; using the projected NFA's such an expression is
   E1 = (b* ; a ; c*)^ /\ (c^ > a > b^)*
3. rename back to get an expression for S2
   E2 = rho [(b* ; a ; c*)^ /\ (c^ > a > b^)*]
   with rho mapping a,b,c into a,a,a, respectively.

Problems. There are a few problems with this approach, the main critics being the following:

– Intersection is a nonintuitive operator: Indeed, it is difficult to grasp what you get by intersecting two or more languages.

– The formalism is not robust under renaming: As an example, notice that the expression E3 = (a* ; a ; a*)^ /\ (a^ > a > a^)*, obtained by syntactically renaming `a,b,c` as `a` into the expression E above, represents all rectangular words of `a`'s, not only the square ones as one expects.

– Renaming is yet a still more nonintuitive operator: It's like writing in Chinese and getting an English text using a letter-to-letter morphism, losing most of the information.

The solution we propose to the above problems is the following:

– Construct a formalism for handling words of arbitrary shapes in the 2-dimensional plane;

– Introduce a powerful set of composition operators for these shapes (extending vertical/horizontal compositions and their iterated versions)

In other word, **the magic way of getting the intended language by renaming and intersection is to be replaced by a steady work of tiling shapes to get the words step by step**.

3   A new approach

3.1   General 2-dimensional words

A *(pointed) contour* is a closed line, with a chosen starting point, on a rectangular grid $\mathbb{Z} \times \mathbb{Z}$ that divide the space into two disjoint regions: the internal area (which is required to be finite) and the external area. It will be represented using a sequence of letters from the set $\{u,d,l,r\}$ (u stands for "up", d for "down", l for "left", and r for "right") and a placement (x, y) of the starting point. For simplicity, by default one can consider the starting point to be $(0, 0)$.

A few examples of contours are shown in Fig. 3. A representation for C1 is `l_1 u_1 l_1 d_1 l_1 u_2 r_3 d_2`. The numbers after the letters are used to count repetitions. The contour starts at the chosen (black dot) point and travel clockwise. The interior area of a contour is the dashed (yellow) one. As the contour is surrounded clockwise, the area on the right is internal, while the one on the left is external. By changing the starting point, the representation is shifted circularly; for instance, C2 is represented by `l_1 d_1 l_1 u_2 r_3 d_2 l_1 u_1`. Two slightly more complicate contours are shown in C3 and C4 in Fig. 3. As the last example shows, one can have contours with distinct disjoint components in its internal area, connected via lines travelled forth and back in the representation. The lines travelling into the internal areas are also called *tunnels*, while those sitting in the external areas *bridges*. As one can see, the tunnels and the bridges have a lot of freedom regarding both their forms and their
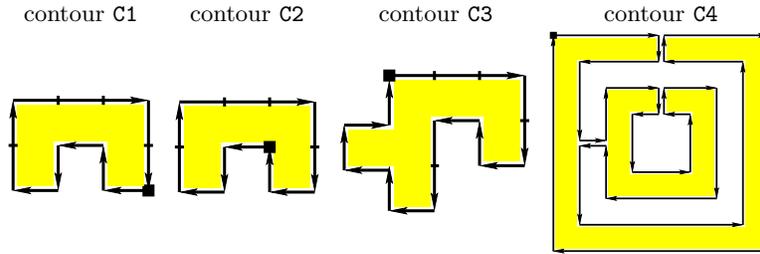
Fig. 3. Contours

physical placement; for instance, they may have branches going nowhere. From the 2-dimensional words point of view, all these representations of contours are equivalent.

The formal definition of a contour requires quite a lot of preparation and it is presented below. A *closed line* `C` is a string over the set $\{\texttt{u},\texttt{d},\texttt{l},\texttt{r}\}$, obeying the conditions `no(C,u) = no(C,d)` and `no(C,r) = no(C,l)`, where `no(C,x)` denotes the number of occurrences of `x` in `C`.

For a point $p = (x, y) \in \mathbb{Z}^2$, p_C^k specifies the information that C passes exactly k time through p; notice that $k \geq 0$. A vertical line segment $((x, y), (x, y + 1))$ is specified by its middle point $l = (x, y + 0.5)$; the notation l_C^k specifies that the difference between the "up" and the "down" times C passes through this segment is k; notice that $k \in \mathbb{Z}$ may be both positive or negative. Similarly, for an horizontal line segment $((x, y), (x + 1, y))$, denoted $l = (x + 0.5, y)$, the notation l_C^k says that k is the difference between the "right" and the "left" times C passes through l. Finally, a unit cell with the corners $\{(x, y), (x + 1, y), (x + 1, y + 1), (x, y + 1)\}$ is specified by its center point $c = (x + 0.5, y + 0.5)$.

For a cell $c = (x + 0.5, y + 0.5)$, the notation $c_{C,w}^k$ specifies how c is seen in C from a western perspective. Formally, let $z = max\{w \in \mathbb{Z} : w < x$ and $l = (w, y + 0.5)$ is such that l_C^k is true with $k \neq 0\}$; then $c_{C,w}^k$ is true if l_C^k is true for the line $l = (z, y + 0.5)$. In words, starting from the center of the cell and travelling horizontally towards the west there is a first line crossed and having a unequal up/down passings and, moreover, the difference between the "up" and the "down" passings along that line is k. The notations $c_{C,e}^k$, $c_{C,n}^k$, and $c_{C,s}^k$ are similarly introduced for the eastern, northern, and southern directions.

A cell is seen as *internal* if $c_{C,w}^k$ and $k > 0$. For the *external* property the condition is slightly different: either $c_{C,w}^k$ and $k < 0$ (i.e., going horizontally towards west, the there is a first line crossed with more down than up passings) or *there is no line crossed with unequal up/down passings*. This additional condition is needed to ensure the internal area of a valid contour is finite; for instance, `drul` is not a valid contour (see below the formal definition of valid contours).

With these notations, the correctness criteria for a string to represent a valid contour are the following: a closed line C represents a *valid contour* if:

– each cell is either internal from all directions, or external from all directions;
– for the internal cells, the conditions $c_{C,w}^k, c_{C,e}^k, c_{C,n}^k$, and $c_{C,s}^k$ are all satisfied with $k = 1$.
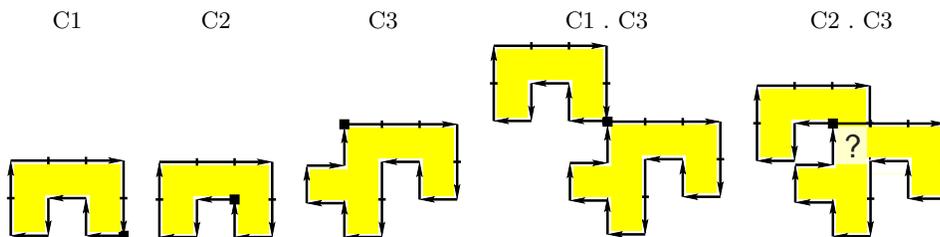
C1   C2   C3   C1 . C3   C2 . C3

Fig. 4. Contours composition

The last condition is needed to avoid overlapping by multiple surroundings of the same internal area; for instance `rdlurdlu` is not valid, while `rdlu` is.

It is possible to replace the conditions here with conditions on the string itself, rather than on the lattice cells. However, such an approach is less intuitive (it is based on forbidden string configurations) and the details are quite complex; see [3].

A *general 2-dimensional word* is specified by a contour and a filling of its internal area with letters from the given alphabet. In the following we will mostly ignore this additional information as most of the difficulties are posed by the handling of the contours/shapes and not by the contents of their internal areas.
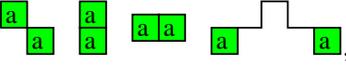
3.2   General composition

A *general composition* operator '.' on contours may be defined as follows: given two contours, get a new contour by putting them together and identifying their starting points (the black dots). This means, one has to travel along the first contour and when he arrives back to the starting point, to travel along the next. In the string representation of the contours, the operation actually is concatenation `C1 . C2 = C1 C2`.

Comments: The condition to have a definite composite is to have a valid non-overlapping contour after the concatenation of the representations of the given contours (for instance, a pointed contour cannot be composed with itself). In particular, this implies that there is no constraint on the contents of the internal areas of the contours, hence the operation is straightforwardly extended to general 2-dimensional words. This is a very powerful and general composition operator, indeed.

For a graphical example, C1 . C3 in Fig. 4 shows a composition leading to a valid contour, while C2 . C3 leads to a string representation which does not represent a valid contour (it has overlapping areas).

This composition is extended to two-dimensional words as follows. For two words `W1, W2`, consider arbitrary contours `C1, C2` representing them (having as internal areas the shapes of the words) and arbitrary positions as starting points of these contours. Then, `W1 . W2` consists of all words resulting from valid compositions of such contours and placing the letters of the words `W1, W2`

in the corresponding positions of the resulting composites. E.g., the composite `a . a` contains the words , etc.

3.3 Particular composition operators

The new type of 2-dimensional regular expressions, to be defined below, put constraints on the contact elements of the composed words. These constraints acts on the following three types of elements: *side borders*, *land corners* (turning points on the contour having 3 neighbouring cells outside the word and one neighbouring cell inside), and *golf corners* (turning points on the contour with at least 2 neighbouring cells inside and one neighbouring cell outside). An example is shown in Fig. 5(1). The resulting restricted composition operators extend the usual vertical and horizontal composition operators used on rectangular words.

Points of interest on the words borders. Let us use the following notation (their meaning is explained right after the listing):

- *side borders*: elements in `C1={w,e,n,s}`, where `w` stands for "west border", `e` for "east border", `n` for "north border", and `s` for "south border";
- *land corners*: elements in `C2={nw,ne,sw,se}`, where `nw` stands for "north-west land corner", `ne` for "north-east land corner", `sw` for "south-west land corner", and `se` for "south-east land corner";
- *golf corners*: elements in `C3={nw',ne',sw',se'}`, where `nw'` stands for "north-west golf corner", `ne` for "north-east golf corner", `sw` for "south-west golf corner", and `se` for "south-east golf corner".

A line $l = (x, y+0.5)$ is on the *east border* of a word f if the cell $c = (x-0.5, y+0.5)$ is in the internal area of f, while the cell $c = (x + 0.5, y + 0.5)$ is in the external area of f. For the other west, north, and south directions, the definition is similar. A point $p = (x,y) \in \mathbb{Z}^2$ is on the *south-east land corner border* of a word f if the cell $c = (x - 0.5, y + 0.5)$ is in the area of f, while the other 3 cells around are not in the area of f (they are in the external area of f). For the other 3 types of land corners the definition is similar. A point $p = (x, y) \in \mathbb{Z}^2$ is on the *south-east golf corner border* of a word f if the cell $c = (x - 0.5, y + 0.5)$ is not in the area of f (it is in the external area of f), while at least 2 of the other 3 cells around are in the area of f. For the other 3 types of golf corners the definition is similar.

Glueing combinations. The constraints on glueing the borders of the words in the composite word are independently put on one or more the following combinations (x, y):

- x and y are different and either they are both in $\{e, w\}$, or both in $\{s, n\}$, or both are land corners in $\{nw, ne, sw, se\}$, or both are combinations golf-land corners for the same directions.

Spelling out the resulting combinations we get the following lists:

- linking side borders: L1={(w,e),(e,w),(n,s),(s,n)};
- linking land corners: L2={(nw,ne),(nw,se),(nw,sw),(ne,nw),(ne,se),
  (ne,sw),(se,nw),(se,ne),(se,sw),(sw,nw),(sw,ne),(sw,se)};
- linking golf-land corners: L3={(nw',nw),(nw,nw'),(ne',ne),(ne,ne'),
  (se',se),(se,se'),(sw',sw),(sw,sw')}.

The set of all combinations in $L1 \cup L2 \cup L3$ is denoted by Connect.

Constricting formulas. On each of the above eligible glueing combination (x, y) we put a constrain given by a propositional logic formula[1] $F \in PL(\phi_1, \phi_2, \phi_3, \phi_4)$, i.e., a boolean formula built up starting with the following atomic formulas:

$$\phi_1(x, y) = \text{``}x < y\text{''}, \phi_2(x, y) = \text{``}x = y\text{''}, \phi_3(x, y) = \text{``}x > y\text{''}, \phi_4(x, y) = \text{``}x \# y\text{''}.$$

The meaning of the connectors is the following: "<" - left is included into the right; "=" - left is equal to the right; ">" - left includes the right; "$x \# y$" - left and right overlaps, but no one is included in the other.

For instance: $f(e = w)g$ means "restrict the general composition of f and g such that the east border of f is identified to the west border of g"; $f(e > w)g$ - the east border of f includes all the west border of g, but some east borders of f may still be not covered by west borders of g; etc.

We also use the notation

$\phi_0(x, y) = \text{``}x \text{ ! } y\text{''}$, where "!" means empty intersection.

Actually, this is a derived formula $\neg(\phi_1(x, y) \vee \phi_2(x, y) \vee \phi_3(x, y) \vee \phi_4(x, y))$.

Particular composition operators. We are now in a position to introduce the particular composition operators induced by the above constricting formulas.

Definition 3. (restricted compositions)
A *restriction formula* ϕ is a boolean combination in $PL(F_1, \ldots, F_n)$, where F_i are constricting formulas involving certain eligible glueing combinations $(x_i, y_i) \in$ *Connect*. A *restricted composition operation* $\_(F)\_$ is the restriction of the general composition to composite words satisfying F. A word $h \in f \cdot g$ belongs to $f (F) g$ if for all glueing combinations (x_i, y_i) occurring in F the contact of the x_i border of f and y_i border of g satisfies F_i. □

This interpretation shows the constricting formulas act on the involved glueing combinations, while for the glueing combinations (x_j, y_j) not occurring in the formula no constraints are imposed, at all. Other default conventions are possible, too; for instance stating that what is not specified should not touch.

Notice that the restricted composition operations are not always associative; e.g., ((a (s=n) a) (e>w) b) (e>w) c $\neq$ (a (s=n) a) (e>w) (b (e>w) c). When some parentheses are missing, we suppose a left-parentheses order applies, as in ((C1 op C2) op C3).

---

[1] $PL(Atom)$ denotes the set of propositional logic formulas built up with atomic formulas in *Atom*. For typing reasons, the boolean operations "not", "and", and "or" are denoted by "!", "&", and "|", respectively.
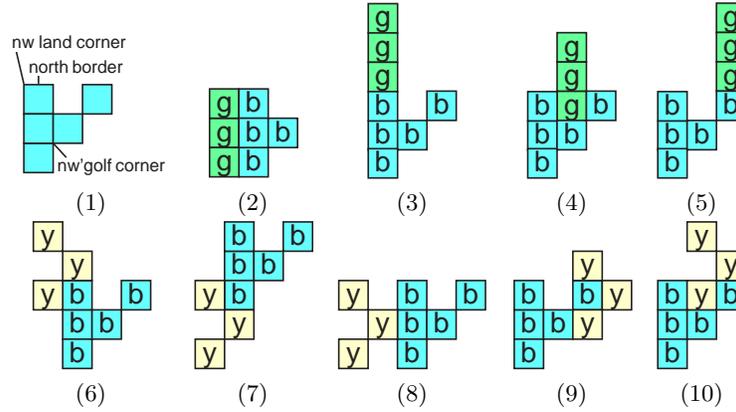
Fig. 5. Particular composition operators

Examples. A few examples are shown in Fig. 5. Let `g,` `b,` and `y` represent the green, blue, and yellow areas, respectively. Then, {(2)} (the word described in Fig. 5(2)) is the result of `g (e=w) b`. Similarly: `g (s<n) b` is the set of words {(3),(4),(5)}; `b (ne>ne') y` is {(9)}; `y (s#n & w#e) b` is {(9),(10)}. The words for `y (e#w) b` strictly include the set {(6),(7),(8),(10)}; one can use the expression `y (e#w & ne!nw & se!sw) b` to exclude two words from `y (e#w) b` and to get precisely the set {(6),(7),(8),(10)}.

Definition 4. (iterated composition operators) The *iterated composition operators* are denoted by $*(F)$, for a restriction formula F. □

Definition 5. The set of expressions obtained using the operators defined so far are denoted by n2RE's; they represent our *new type of regular expressions for two-dimensional patterns/words.* □

Examples, related to S1. The examples in Fig. 6 are related to $S1$, the original FIS we have considered in the beginning of the section. We first show the expressions, then include samples of typical words associated to these expressions. Combined with the constraint to have rectangular words, the final regular expression `Eabc` specifies the language of `S1`.

Observation. The formalism is robust, in particular it commutes with renaming: renaming letters either in the associated words or in the given expression leads to the same set of general 2-dimensional words.

4   A relational semantics for structured interactive programs

In this section we show how the introduced regular expressions can be used to get a relational semantics for structured interactive programs presented in the rv-IS
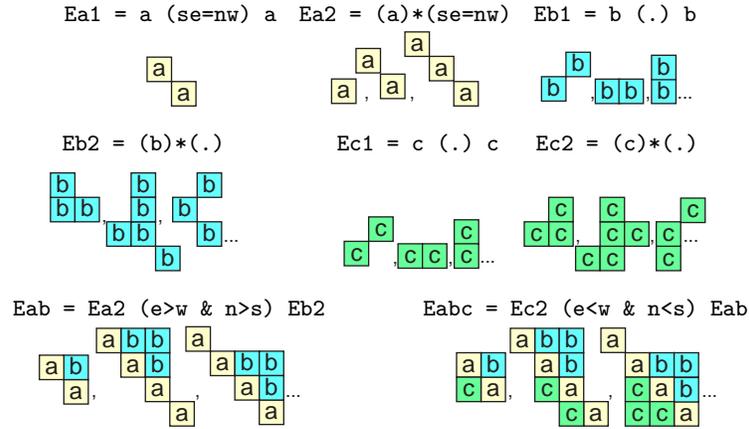
Ea1 = a (se=nw) a   Ea2 = (a)*(se=nw)   Eb1 = b (.) b

Eb2 = (b)*(.)   Ec1 = c (.) c   Ec2 = (c)*(.)

Eab = Ea2 (e>w & n>s) Eb2   Eabc = Ec2 (e<w & n<s) Eab

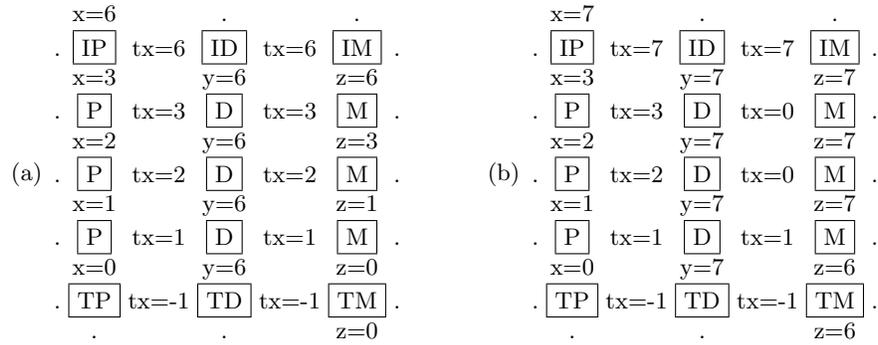**Fig. 6.** A n2RE expression for the FIS $S1$

Fig. 7. Two scenarios for computing perfect numbers

formalism [35, 12]. The operational semantics of structured programs is given in terms of scenarios. In Fig. 7(a) we illustrate an rv-IS scenario for deciding whether the number 6 is a perfect number (i.e., it is equal to the sum of its proper divisors); in (b) it is a scenario for testing if 7 is a perfect number.

In this representation, the actions to be performed are placed in the square cells. One example is the cell with the identifier P placed in the 2nd row and the 1st column in (a). On the top and the bottom of this cell there is the same state variable x with its concrete values 3 and 2. Actually, the effect of P on the memory state is to decrease x by 1. With respect to the interaction part, this cell has no variables for its left border (a fact specified by the '.' inserted there) and has a variable tx at its right border. The effect of P on the interaction part is to set in tx the input value 3 of x in order to be used in other columns/processes.

In Fig. 8 we present relational specifications for the cells used in Fig. 7. All these cells have functional behaviour, hence the corresponding relations may be

specified as partial functions in the following way:

```
Cell(west,north) = (east,south), if Condition.
```

4.1   Example - imperative programming style

We start with the following expression specifying scenarios checking if a number n is perfect

```
        [((IP (e=w) ID) (e=w) IM)
  (s=n) (((P (e=w) D) (e=w) M) *(s=n))]
  (s=n) ((IP (e=w) ID) (e=w) IM)
```

In this model we can imagine that we have three processes: one generates all the numbers in the set $\{n/2, \ldots, 1\}$ (with module P), one checks if a number is a divisor of n (module D) and the last one updates a variable z (module M). Modules IP, ID and IM are used for initializations and TP, TD and TM for termination. At the end of the program, if the variable z is 0, then the number n is perfect.

In order to show how we can construct a scenario using the expression above let us consider a concrete example for $n = 6$. The scenario for $n = 6$ is presented in Fig. 7 (a).

```
  IP((.),(x)) = ((tx'),(x')) with tx' = x and x' = x/2; defined if x ≥ 2
  ID((tx),(.)) = ((tx'),(y')) with tx' = tx and y' = tx
  IM((tx),(.)) = ((.),(z'))   with z' = tx
   P((.),(x)) = ((tx'),(x')) with tx' = x and x' = x-1; defined if x > 0
   D((tx),(y)) = ((tx'),(y')) with y' = y and
                                tx' = if(y%tx=0) then tx else 0
   M((tx),(z)) = ((.),(z'))   with z' = z-tx
  TP((.),(x)) = ((tx'),(.))  with tx' = -1; defined if x = 0
  TD((tx),(y)) = ((tx'),(.))  with tx' = -1
  TM((tx),(z)) = ((.),(z'))   with z' = z
```

Fig. 8. Relational semantic specifications for the cells used in Fig. 7

In the first line of the scenario we initialize the processes with the needed informations: module IP is reading the value $n = 6$ and provides the first process with $x = 3$ and declare a temporal variant of n, namely $tn = 6$, that will be used by modules ID and IM for the other initializations; modules ID and IM use the temporal variable tn for initializing the other two processes with the initial value of n, namely $y = 6$, $z = 6$, respectively.

In the next step, module P produces a temporal data $tx = 3$ (tx is equal with the data x of the first process) and decrease x. Module D verifies if tx is a divisor of y and, if no, it resets the value of tx to 0. Finally, module M decreases the value of z by tx. Notice that module M decreases the value of z only with the divisors of the initial variable n. We continue this steps until the variable x becomes 0.

A final line contains terminating modules that rearrange some interfaces, keeping only the relevant result z.

4.2 Dataflow and mixed imperative-dataflow programming styles.

The above model corresponds to the construction of scenarios by rows and it exhibits a (parallel) *imperative programming style*, illustrated in Fig. 9(a).
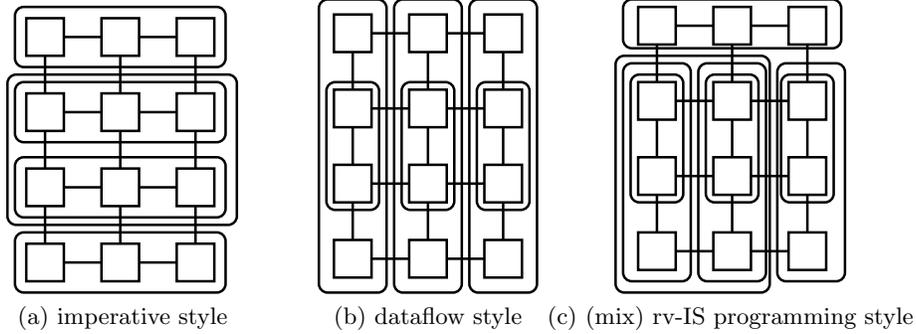

(a) imperative style   (b) dataflow style   (c) (mix) rv-IS programming style

Fig. 9. Programming strategies

The same computing scenarios may be generated in many other ways. Below is a model which constructs the same scenarios by columns, exhibiting a *dataflow computing style*, illustrated in Fig. 9(b):

```
          [((IP (s=n) (P *(s=n)) (s=n) TP)
(e=w) ((ID (s=n) (D *(s=n)) (s=n) TD)]
(e=w) ((IM (s=n) (M *(s=n)) (s=n) TM)
```

In this implementation, the processes corresponding to the 2nd and the 3rd columns act as "services": they receive initialization data (here the value of **n**), then a stream of data to act individually on each one according to the service function (for the 2nd process this function is the check for divisibility, while for the 3rd is subtraction), and finally a termination token (represented here by -1).

Finally, we present a last model, illustrated in Fig. 9(c), which *mixes the imperative and dataflow styles*

```
          ((IP (e=w) ID) (e=w) IM)
(s=n) [(((P (e=w) D) *(s=n)) (s=n) (TP (e=w) TD))
          (e=w) ((M *(s=n)) (s=n) TM)]
```

In this version the construction of the scenarios is as follows. It starts by constructing the 1st line of the scenarios. Then, the remaining parts of the first two columns are generating in the same way as with the initial model (that is, by an imperative style). Moreover, the same is done separately for the remaining part of the third column. Finally, these parts are composed horizontally (following a dataflow style).

5   Related and future works

Related work. Regular expressions are introduced in the seminal paper by Kleene [17] on the representation of events in neural nets and automata; it was published in the early 1950s. Kleene theorem (i.e., the equivalence between finite automata and appropriate regular expressions) was extended to cover other computing models of interest and is a basis for the development of algebraic theories for those models.

The algebraic theory of finite automata is based on semirings enriched with an axiomatic iteration operator; often the term Kleene algebra is used in this context. It was steadily developed since 1960s till now, including deep results as Krob's solution [19] for two deep conjectures of Conway [11]. We notice the interest in getting complete equational axiomatizations; see, e.g., [28, 18, 6, 19, 8]. A few books recording the results are [11] and [20].

When (matrices over) semirings are replaced by more general algebraic structures as symmetric (strict) monoidal categories, the iteration operators may have different expressive powers and axiomatisations. Trace monoidal categories [30, 9, 16, 33] are now recognized as a powerful formalism for iterative processes, with wider applications than Kleene algebras; in particular they apply to circuits, dataflow computation, quantum computation, etc. Translations between various formalisms using axiomatic iteration operators may be found in [31, 32, 7, 10, 33]. Axiomatic iteration operators are also present in process calculi; a few papers are [25, 5, 26].

Parallel computation often requires the enrichment of the sequential computation models with mechanisms for modelling process interaction. We mention three examples of extensions of Kleene theorem into such a context: Kleene theorems for tile systems [14], for Petri nets [27, 13], and for timed automata [1, 2]. In all these contexts, the Kleene theorem is based on the following procedure: (1) decompose/project the behaviour to have separate sequential runs; (2) use the classical Kleene theorem for these sequential runs; (3) use synchronization and renaming to force the composition of these separate projected runs to behave as the initial overall system. It was noticed (see, e.g., [18]) that renaming has bad algebraic properties and should be avoided.

The study of two dimensional languages has started in 1960's; see [14, 22]. In 1990's, a robust class of "regular two dimensional languages" has been identified; it may be specified either by tile systems, or by a type of cellular automata, or by a class of monadic second-order formulas, etc. Unfortunately, the class is quite complex - for instance, emptiness property is not decidable, see [21].

Interactive computation [15] is becoming more and more important in the recent years, in particular due to the advance of multicore computation. We use a model rv-IS [35] based on space-time duality. In particular, finite interactive systems [34] are the space-time invariant extension of finite automata in this context. A Kleene theorem for finite interactive systems follows directly from their equivalence with tile systems [29]. Agapia programming [12] is a core interactive programming language based on this model. The relational semantics described in the present paper for Agapia programs may be seen as an extension to 2

dimensions of the classical relational semantics of sequential computing models [23, 24].

Future work. There are many directions to continue the research presented in this paper. We are particularly interested to develop the theoretical basis of the model (e.g., to prove a Kleene theorem for finite interactive systems[2]; to look for an associated algebraic theory; etc.) and to provide a software tool for manipulating n2RE's. Among the possible applications we mention:

- the study of massively parallel, interactive OO-programs (semantics, specification, verification, etc.), in particular the programs written in the structured interactive programming language Agapia;
- applications to image processing, in particular learning n2RE as a image recognition procedure;
- modelling discrete physical or biological systems.

Acknowledgements. The research reported in this paper was partially supported by Deploy Project, an FP7 Integrated Project supported by European Commission (Grant No. 214158). We thank the anonymous reviewers for their suggestions for improving the presentation of the results.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
2. Asarin, E., Caspi, P., Maler, O.: Timed regular expressions. J. ACM 49, 172–206 (2002)
3. Banu-Demergian, I., Stefanescu, G.: On the contour representation of two-dimensional patterns (2013), draft
4. Banu-Demergian, I., Stefanescu, G.: Representation of scenarios in finite interactive systems (2013), draft, submitted
5. Bergstra, J., Bethke, I., Ponse, A.: Process algebra with iteration. The Computer Journal 60, 109–137 (1994)
6. Bloom, S., Esik, Z.: Equational axioms for regular sets. Mathematical Structures in Computer Science 3, 1–24 (1993)
7. Bloom, S., Esik, Z.: Iteration Theories: The Equational Logic of Iterative Processes. Springer-Verlag, Berlin (1993)
8. Bonsangue, M., Rutten, J., Silva, A.: A Kleene theorem for polynomial coalgebras. In: Proc. FSSCS'09. pp. 122–136. LNCS, Springer-Verlag (2009)
9. Cazanescu, V., Stefanescu, G.: Towards a new algebraic foundation of flowchart scheme theory. Fundamenta Informaticae 13, 171–210 (1990)
10. Cazanescu, V., Stefanescu, G.: Feedback, iteration and repetition. In: Păun, G. (ed.) Mathematical aspects of natural and formal languages, pp. 43–62. World Scientific, Singapore (1995)

---

[2] Recently, the first and the last authors presented a characterization theorem for FIS languages in [4]. Their result shows that a slightly extended class of n2RE expressions and a mechanism for solving recursive equations suffice to represent FIS languages.

11. Conway, J.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
12. Dragoi, C., Stefanescu, G.: Agapia v0. 1: A programming language for interactive systems and its typing system. Electronic Notes in Theoretical Computer Science 203(3), 69–94 (2008)
13. Garg, V., Ragunath, M.: Concurrent regular expressions and their relationship to Petri nets. Theoretical Computer Science 96, 285–304 (1992)
14. Giammarresi, D., Restivo, A.: Two-dimensional languages. In: Handbook of formal languages, pp. 215–267. Springer (1997)
15. Goldin, D., Smolka, S., Wegner, P.: Interactive computation: The new paradigm. Springer (2006)
16. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. In: Proceedings of the Cambridge Philosophical Society. vol. 119 (1996)
17. Kleene, S.: Representation of events in nerve nets and finite automata. Automata Studies (34), 3 (1956)
18. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: LICS'91. pp. 214–225. IEEE (1991)
19. Krob, D.: Complete systems of β-rational identities. Theoretical Computer Science 89, 207–343 (1991)
20. Kuich, W., Salomaa, A.: Semirings, automata and languages. Springer-Verlag, Berlin (1985)
21. Latteux, M., Simplot, D.: Context-sensitive string languages and recognizable picture languages. Information and Computation 138(2), 160–169 (1997)
22. Lindgren, K., Moore, C., Nordahl, M.: Complexity of two-dimensional patterns. Journal of statistical physics 91(5-6), 909–951 (1998)
23. Maddux, R.: Relation-algebraic semantics. Theoretical Computer Science 160, 1–85 (1996)
24. Manes, E., Arbib, M.: Algebraic approaches to program semantics. Springer-Verlag, Berlin (1986)
25. Milner, R.: Flowgraphs and flow algebras. Journal of the ACM (JACM) 26(4), 794–818 (1979)
26. Milner, R.: Action calculi V : Reflexive molecular forms (1994), draft, Department of Computer Science, University of Edinburgh
27. Petri, C.: Kommunikation mit automaten. Ph.D. thesis, Instituts fur Instrumentelle Mathematik, Bonn, Germany
28. Salomaa, A.: Two complete axiom systems for the algebra of regular events. Journal of the ACM (JACM) 13(1), 158–169 (1966)
29. Sofronia, A., Popa, A., Stefanescu, G.: Undecidability results for finite interactive systems. Romanian Journal of Information Science and Technology 12(2), 265–279 (2009), also: Arxiv, CoRR abs/1001.0143, 2010
30. Stefanescu, G.: Feedback Theories (A Calculus for Isomorphism Classes of Flowchart Schemes). No. 24 in Preprint Series in Mathematics, INCREST (1986), also in: *Revue Roumaine de Mathematiques Pures et Applique*, 35:73–79, 1990
31. Stefanescu, G.: On flowchart theories: Part I. The deterministic case. Journal of Computer and System Sciences 35(2), 163–191 (1987)
32. Stefanescu, G.: On flowchart theories: Part II. The nondeterministic case. Theoretical Computer Science 52(3), 307–340 (1987)
33. Stefanescu, G.: Network algebra. Springer Verlag (2000)
34. Stefanescu, G.: Algebra of networks: Modeling simple networks as well as complex interactive systems. In: Proof and System-Reliability, pp. 49–78. Springer (2002)
35. Stefanescu, G.: Interactive systems with registers and voices. Fundamenta Informaticae 73(1), 285–305 (2006)