Structured programming for interactive rv-systems

Cezara Drăgoi and Gheorghe Ștefănescu

Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, Bucharest, Romania 010014
Email: {cdragoi,gheorghe}@funinf.cs.unibuc.ro

Abstract. A model, a core programming language, a specification and several analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced by Stefanescu in 2004 using register machines and space-time duality [9]:
(i) The model consists of rv-systems, it includes register machines, it is space-time invariant, it is compositional, it may describe computations extending in both time and space, and it is applicable to open, interactive systems. (ii) The programming language uses novel techniques for its syntax and semantics to support computation in the space paradigm: it uses rv-programs whose syntax and operational semantics are based on finite interactive systems and running scenarios. (iii) The specification of the rv-systems uses relations between the input registers and voices and their output counterparts.

The above rv-programs resemble flowcharts and assembly languages: one freely uses go-to statements, but now in a much extended setting where both temporal and spatial labels are used. In the present paper we show how structured programming techniques used for the classical sequential programs may be adapted to obtain structured rv-programs. Particular emphasis is on developing a structural spatial programming discipline, which provides a more structured interaction between objects and it is of great help in simplifying the construction and the analysis of interactive programs.

1 Introduction

A model, a core programming language, a specification and several analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been introduced by Stefanescu in 2004 using register machines and space-time duality [9]: (i) The model consists of rv-systems, it includes register machines, it is space-time invariant, it is compositional, it may describe computations extending in both time and space, and it is applicable to open, interactive systems. (ii) The programming language uses novel techniques for its syntax and semantics to support computation in the space paradigm: it uses rv-programs whose syntax and operational semantics are based on finite interactive systems and running scenarios. (iii) The specification of the rv-systems
uses relations between the input registers and voices and their output counterparts.

The above rv-programs resemble flowcharts and assembly languages: one freely uses go-to statements, but now in a much extended setting where both temporal and spatial labels are used. In the present paper we show how structured programming techniques used for the classical sequential programs may be adapted to obtain structured rv-programs. Particular emphasis is on developing a structural spatial programming discipline, which provides a more structured interaction between objects and it is of great help in simplifying the construction and the analysis of interactive programs.

The paper starts with a brief presentation of the main concepts needed to understand structured rv-programs and their semantics: rv-programs and their scenarios, spatio-temporal specifications using registers and voices. Then, the structured programming for rv-programs is developed and illustrated on specific examples.

## 2 Interactive programs with registers and voices

In this section we briefly describe rv-programs (interactive programs with registers and voices); see [9] for more details.

**Grids and scenarios.** A grid is a rectangular two-dimensional area filled in with letters of a given alphabet. An example of grid is presented in Fig. 1. In our standard interpretation the columns correspond to processes, the top-to-bottom order describing their progress in time. The left-to-right order corresponds to process interaction in a nonblocking message passing discipline: a process sends a message to the right, then resumes its execution.

A scenario is a grid enriched with data around each letter. The data may be given in an abstract form as Fig. 1(a) and (b), or in a more detailed form as in Fig. 1(c).

![Grid and scenarios](image)

**Finite interactive systems.** A finite interactive system (shortly FIS) is a finite hyper-graph with two types of vertices and one type of (hyper) edges:
(1) the first type of vertices is for states, labeled by numbers/lower case letters;
(2) the second type of vertices is for classes, labeled with capital letters; (3) the edges/transition are labeled by letters denoting atoms of the grids; each transition has two incoming arrows (one from a class, the other from a state), and two outgoing arrows (one to a class, the other to a state). Some classes/states may be initial (indicated by small incoming arrows) or final (indicated by double circles); see, e.g., [9]. An example is shown below.

For the parsing procedure, given a Fis $F$ and a grid $w$, insert initial states/classes at the north/west border of $w$ and parse the grid completing the scenario according to the Fis transitions; if the grid is fully parsed and the south/east border contains final states/classes, the grid $w$ is recognized by $F$. The language of $F$ is the set of its recognized grids. A Fis $F_1$ and a parsing for $abbcabcc$ are shown here.

\[ F_1 = \]

\[ \begin{array}{cccccccc}
\text{A} & & & & & & & \\
\text{B} & & & & & & & \\
\text{C} & & & & & & & \\
\text{D} & & & & & & & \\
\end{array} \]

\[ \begin{array}{cccccccc}
\text{1} & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\text{2} & 2 & 1 & 2 & 1 & 1 & 1 & 2 \\
\text{3} & & & & & & & \\
\text{4} & & & & & & & \\
\end{array} \]

\[ \begin{array}{cccccccc}
\text{A} & \text{a} & \text{b} & \text{A} & \text{b} & \text{B} & \text{a} & \text{b} \\
\text{A} & \text{a} & \text{b} & \text{b} & \text{A} & \text{b} & \text{B} & \text{a} \\
\text{A} & \text{a} & \text{b} & \text{b} & \text{A} & \text{a} & \text{b} & \text{B} \\
\text{A} & \text{a} & \text{b} & \text{b} & \text{A} & \text{a} & \text{b} & \text{B} \\
\end{array} \]

\[ \text{Spatio-temporal specifications} \]

For the spatial data, we use the common data structures and their natural representations in memory.

For the temporal data we use the stream structure: a stream is a sequence of data ordered in time and is denoted as $a_0\ldots a_1\ldots$, where $a_0, a_1\ldots$ are its data at time $0, 1\ldots$, respectively. A stream may be seen as the result of observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle.

A voice is defined as the time-dual of a register: A voice is a temporal structure that holds a natural number. It can be used (“heard”) at various locations. At each location it displays a particular, possible different, value.

Voices may be implemented on top of a stream in a similar way registers are implemented on top of a Turing tape, for instance specifying their starting address and their length. Most of usual data structures have natural temporal representations. Examples: $\text{tBool}$ (timed booleans), $\text{tInt}$ (timed integers), etc.

The notation $\otimes^*$ is used for the spatial product, while $\otimes$ for the temporal product (mathematically, they are just the Cartesian product); $N^\otimes^k$ denotes $N^\otimes\ldots\otimes N$ ($k$ terms) and $N^{-\otimes^k}$ denotes $N^{-\otimes\ldots\otimes N}$ ($k$ terms).

A spatio-temporal specification $S : (m, p) \rightarrow (n, q)$ is a relation $S \subseteq (N^{-m} \times N^\otimes^p) \times (N^{-n} \times N^\otimes^q)$, where $m$ (resp. $p$) is the number of input voices (resp. registers) and $n$ (resp. $q$) is the number of output voices (resp. registers). On elements, it is defined as a relation between concrete tuples, written as $v \mid r \mapsto v'$, where $v, v'$ (resp. $r, r'$) are tuples of voices (resp. registers).

Examples. The constants $c_0 = \emptyset$, $c_1 = \{0\}$, $c_2 = \{0 \mid x\} \mapsto \langle \mid x \rangle : x \in \mathbb{N}$; $c_3 = \{0 \mid x\} \mapsto \langle \mid x \rangle : x \in \mathbb{N}$; $c_4 = \{0 \mid x\} \mapsto \langle \mid x \rangle : x \in \mathbb{N}$; $c_5 = \{0 \mid x\} \mapsto \langle \mid x \rangle : x \in \mathbb{N}$; $c_6 = \{0 \mid x\} \mapsto \langle \mid x \rangle : x \in \mathbb{N}$. The constants $c_3$ (with various types for $x$, not only $\mathbb{N}$) are called speakers, while $c_4$ recorders.
Specifications may be composed horizontally and vertically, as long as their types agree; e.g., for two specifications $S_1 : (m_1, p_1) \rightarrow (n_1, q_1)$ and $S_2 : (m_2, p_2) \rightarrow (n_2, q_2)$ the horizontal composition $S_1 \triangleright S_2$ is defined only if $n_1 = m_2$ and the type of $S_1 \triangleright S_2$ is $(m_1, p_1 + p_2) \rightarrow (n_2, q_1 + q_2)$.

**Interactive programs with registers and voices** An rv-system (interactive system with registers and voices) is a FIS enriched with: (i) registers associated to its states and voices associated to its classes; and (ii) appropriate spatio-temporal transformations for actions.

We study programmable rv-systems specified using rv-programs. An example of rv-program is presented in Fig. 2. A computation is described by a scenario like in a FIS, but with concrete data around each action, see Fig. 1(c). Other examples of rv-programs and associated running scenarios may be found in [9].

**Syntax:** The syntax is based on the syntax used in imperative programming languages. The basic block is a module. To explain the syntax, let us focus on the first module of the program **Perfect** in Fig. 2. It has a name X and 4 areas: (1) In the top-left part we have a pair of labels (A, 1) which specifies the interaction and control coordinates where this module has to be applied. (2) The top-right part declares the spatial input variables. (3) The bottom-left part declares the temporal input variables. (4) The body of a module is its bottom-right part, including type declarations and C-like code. The exit from the module is realized by a goto statement, a statement like goto [B, 3] indicates that: (i) the data of the spatial variables in the current module will be used in a next module with control state 3; (ii) the data of the temporal variables in the current module will be used for the interaction interface of a new module with interaction label B.

**Operational semantics:** The operational semantics is given in terms of scenarios. Scenarios are built up with the following procedure, described using the scenario in Fig. 1(c) (for the rv-program **Perfect**):

**Fig. 2.** The rv-program **Perfect** (for perfect numbers)
Steps for building scenarios:

(1) Each cell of the associated grid has as label a module name \( X, Y, Z, \ldots \).

(2) An area around a cell may have an additional information as \( x=2 \) - that means, in that area \( x \) is updated to be 2.

(3) The full information on the current state of a process is obtained going vertically up and collecting the last updated values of the spatial variables. For instance, the bottom \( V \) in the second column has \( y=4 \).

(4) The full information on temporal variables in a current place is obtained collecting their last updated values going horizontally on left. For instance, \( W \) in the second row has \( tx=2 \).

(5) The first column has an input class and particular tuples of values for its temporal variables; the first row has an input state and particular tuples of values for its spatial variables.

(6) The computation in a cell \( \alpha \) is done as follows: (i) Take a module \( \beta \) of the program bearing the class label of the left neighboring area of \( \alpha \) and the state label of the top neighboring area of \( \alpha \). (ii) Follow the code in \( \beta \) using the spatial and the temporal variables of \( \alpha \) with their current values. (iii) If the local execution of \( \beta \) is finished with a \texttt{goto} \( [\Gamma, \gamma] \) statement, then the label of the right neighboring area of \( \alpha \) is set to \( \Gamma \) and the label of the bottom neighboring area of \( \alpha \) is set to \( \gamma \). (iv) Insert the values of the temporal variables updated by \( \beta \) in the right neighboring area of \( \alpha \) and the values of the spatial variables updated by \( \beta \) in the bottom neighboring area of \( \alpha \).

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

The scenario in Fig. 1(c) is a complete scenario for the rv-program Perfect.

Space-Time Duality. Space-time duality interchanges information in space and information in time, e.g., registers and voices. Then, it is naturally lifted to grids, scenarios, fis-es, spatio-temporal specifications, rv-systems, and rv-programs, which are all space-time invariant. The space-time operator \( \land \) is defined by:

- On grids: transpose the grid; replace each letter by a dual letter;
- On fis-es: interchange states and classes; replace each letter by a dual letter;
- On scenarios: apply \( \land \) to the underlaying grid; around each letter interchange input registers with input voices and output registers with output voices;
- On rv-programs, in each module: interchange class and state labels; interchange temporal and spatial data; switch top-right and bottom-left corners; (notice that, except for label and variable type change, no more modifications are needed in the body of a module).

Then, for any rv-program \( R \), its space-time dual \( R^\land \) is an rv-program and \( (R^\land)^\land = R \). Moreover, space-time respects operational semantics and input-output denotation.
3 The syntax of structured rv-programs

Extending structured programming to rv-systems is not so easy. The overall goal is to “liberate rv-programming of go-to statements”, hence to use no temporal or spatial labels, at all. A main source of complications is to match the typing of the modules at their composing interfaces. For usual structured sequential programs the state space is unique, hence there are no such typing problems.

Syntax:

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

The type of a program \(P\), denoted by \(P : (w(P), n(P)) \rightarrow (e(P), s(P))\) indicates the types at its west, north, east, and south border. On each side, the type is a sequence of tuples, denoted as \((x_1, \ldots, x_m); (y_1, \ldots, y_n); \ldots; (z_1, \ldots, z_p)\).

The starting blocks for the construction of structured rv-programs are called modules. The syntax of a module is given as follows:

\[
\text{module} ::= \{(\text{read spatial variables}:)(\text{listen temporal variables}:)(\text{code}:)
\quad (\text{write spatial variables})(\text{speak temporal variables})\}
\]

where the \text{read} (\text{listen}) instruction reads the spatial (temporal) input and the \text{write} (\text{speak}) instruction returns the spatial (temporal) output. The \text{code} consists in instructions that are similar to the C code.

In the following the instruction that reads the input or returns the output will be omitted by taking into consideration the following conventions: any variable not declared in the code will be considered an input variable and any variable that appears in the code will be considered an output variable.

The operations of structured rv-programming are described below.

**Composition:** Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types agree. They can also be composed diagonally by mixing the horizontal and vertical composition.

— For two programs \(P_i : (w_i, n_i) \rightarrow (e_i, s_i), i = 1, 2\), the horizontal composition \(P_1 # P_2\) is well defined only if \(e_1 = w_2\); the type of the composite is \((w_1, n_1 # n_2) \rightarrow (e_2, s_1 # s_2)\).

— Similarly, the vertical composition \(P_1 ; P_2\) is defined only if \(s_1 = n_2\); the type of the composite is \((w_1; w_2, n_1) \rightarrow (e_1; e_2, s_2)\).

— The diagonal composition \(P_1$P_2\) is a derived operation - it connects the east border of \(P_1\) to the west border of \(P_2\) and the south border of \(P_1\) to the north border of \(P_2\); it is defined only if \(e_1 = w_2\) and \(s_1 = s_2\); the type of the composite is \((w_1, n_1) \rightarrow (e_2, s_2)\).

If: For the “if” operation, given two programs with the same type \(P, Q : (w, n) \rightarrow (e, s)\), a new program \(\text{if}\ (c)\ \text{then}\ P\ \text{else}\ Q : (w, n) \rightarrow (e, s)\) is constructed, for a condition \(c\) involving both, the temporal variables in \(w\) and the spatial variables in \(n\).

While: There are various possible extensions of the while statement.

— For a program with a dummy interaction interface \(P : (0, n) \rightarrow (0, n)\), a temporal while is defined as a natural extension of the usual “while” statement,
The syntax has been defined above. In this section we describe the semantics, by a translation in rv-systems. The operational semantics (i.e., the set of running scenarios) and the denotational semantics (i.e., the input-output relation) of a structured rv-program $P$ are obtained from its associated rv-program $\text{Tr}(P)$.

### 4.1 Simple interfaces

The semantics of a module $M$ is the semantics of the rv-program that consists in only one basic block, denoted $\text{Tr}(M)$.

The input registers (voices) of $\text{Tr}(M)$ are the variables that appear in the read (listen) instruction and the output registers (voices) of $\text{Tr}(M)$ are the variables that appear in the speak (write) instruction. The pair of control labels of $\text{Tr}(M)$ consists in the input class (the set of input voices) and in the input state (the set of input registers). The code executed by $\text{Tr}(M)$ is the code from $M$ enriched with \text{go to} statements for each possible termination. The \text{go to} statement transfers the control using a pair of labels, one for the output class (denoting the set of output voices) and another one for the output state (denoting the set of output registers). The initial state and the initial class of the rv-program are the control labels of $\text{Tr}(M)$.

A module $M$ has a specification: $M : (m, p) \to (n, q)$, where $m$ ($n$) is the number of input (output) registers and $p$ ($q$) is the number of input (output) voices.

**Example 1.** Next we will consider a program formed of only one module (that has only one possible ending) and on it’s right the rv-program semantically equivalent that has the initial class is $A$ (represents the set of input voices) and the initial state is $1$ (represents the set of input registers). The final class is $D$.
Horizontal composition. The rv-program \( \text{Tr}(X_1 X_2) \) corresponding to \( X_1 X_2 \) is obtained taking the translations \( \text{Tr}(X_1), \text{Tr}(X_2) \) of its arguments and identifying the input class of \( \text{Tr}(X_2) \) to the output class of \( \text{Tr}(X_1) \). The initial/final states of \( \text{Tr}(X_1) \) and \( \text{Tr}(X_2) \) are identified, respectively; moreover, the type of the resulting state is the union of the types of the former states. The identities for the horizontal composition are \( \text{Id} \# : (m,0) \rightarrow (m,0) \), i.e., modules that have no spatial input and the temporal input is left unchanged.

An example:

\[
\{ x,y,z,w : \text{sInt}; x=\text{random}(0,200); y=\text{random}(0,200); \\
            z=\text{random}(200,400); w=\text{random}(200,400); \text{write } x,y,z,w; \} \#
\{ x1 : \text{sInt}; x1=\text{random}(0,400); \text{write } x1; \} \#
\{ y1 : \text{sInt}; y1=\text{random}(0,400); \text{write } y1; \} \#
\]

The 2nd horizontal composition \# is using simple types \( 0|x_1 \rightarrow 0|x_1 \) and \( 0|x_2 \rightarrow 0|x_2 \); the top \# shows that the full type of this piece of code is \( 0|(x,y,z,w);x1:x2) \rightarrow 0|(x,y,z,w);x1:x2) \) (to make the identification easier, we wrote the variables in the fields, not their real types \( \text{sInt} \)).

Vertical composition. Vertical composition is similar - actually, it may be obtained using space-time duality. The rv-program \( \text{Tr}(X_1;X_2) \) corresponding to \( X_1;X_2 \) is obtained from the translations \( \text{Tr}(X_1), \text{Tr}(X_2) \) of its arguments by the identification of the input state of \( \text{Tr}(X_2) \) to the output state of \( \text{Tr}(X_1) \). Moreover, the initial/final states of \( \text{Tr}(X_1), \text{Tr}(X_2) \) are identified, respectively. The identities for the horizontal composition are \( \text{Id}^\#_n : (0,m) \rightarrow (0,m) \), i.e., modules that have no temporal input and the spatial input is left unchanged.

An example:

\[
\{ x\text{:Int}; t:t\text{Int}; \text{t=x}; \text{t} \}
\{ x=x/2; \} \{ \text{t=x}; \text{t} \}
\]

The type is \( 0|\{x\} \rightarrow t|\{t|x\} \), where again each variable should be replaced by its type.

Diagonal composition. The translated rv-program \( \text{Tr}(X_1 SX_2) \) consists of nine parts: the translated rv-programs of the arguments \( \text{Tr}(X_1), \text{Tr}(X_2) \), two recorders \( R_1, R_2 \), two speakers \( S_1, S_2 \), an identity \( \text{Id} \), and two empty modules \( \Lambda_1, \Lambda_2 \). The initial state and class of the program are those of \( \text{Tr}(X_1) \), while the final ones are those of \( \text{Tr}(X_2) \).

In order to preserve the temporal and the spatial output of \( \text{Tr}(X_1) \) and pass them as input to \( \text{Tr}(X_2) \), the temporal output of \( \text{Tr}(X_1) \) is recorded using \( R_1 \) (the values of the temporal variables are assigned to spatial variables) and it is read

\[
\text{sx:spaceInput ; tx:tempInput } (A,1) \text{ cod } M; \text{write } sx; \text{speak } tx; \}
goto(D,2);
\]

\[
\text{Horizontal composition. The rv-program } \text{Tr}(X_1 X_2) \text{ corresponding to } X_1 X_2 \text{ is obtained taking the translations } \text{Tr}(X_1), \text{Tr}(X_2) \text{ of its arguments and identifying the input class of } \text{Tr}(X_2) \text{ to the output class of } \text{Tr}(X_1). \text{The initial/final states of } \text{Tr}(X_1) \text{ and } \text{Tr}(X_2) \text{ are identified, respectively; moreover, the type of the resulting state is the union of the types of the former states. The identities for the horizontal composition are } \text{Id} \# : (m,0) \rightarrow (m,0), \text{i.e., modules that have no spatial input and the temporal input is left unchanged.}
\]

An example:

\[
\{ x,y,z,w : \text{sInt}; x=\text{random}(0,200); y=\text{random}(0,200); \\
            z=\text{random}(200,400); w=\text{random}(200,400); \text{write } x,y,z,w; \} \#
\{ x1 : \text{sInt}; x1=\text{random}(0,400); \text{write } x1; \} \#
\{ y1 : \text{sInt}; y1=\text{random}(0,400); \text{write } y1; \} \#
\]

The 2nd horizontal composition \# is using simple types \( 0|x_1 \rightarrow 0|x_1 \) and \( 0|x_2 \rightarrow 0|x_2 \); the top \# shows that the full type of this piece of code is \( 0|(x,y,z,w);x1:x2) \rightarrow 0|(x,y,z,w);x1:x2) \) (to make the identification easier, we wrote the variables in the fields, not their real types \( \text{sInt} \)).

\[
\text{Vertical composition. Vertical composition is similar - actually, it may be obtained using space-time duality. The rv-program } \text{Tr}(X_1;X_2) \text{ corresponding to } X_1;X_2 \text{ is obtained from the translations } \text{Tr}(X_1), \text{Tr}(X_2) \text{ of its arguments by the identification of the input state of } \text{Tr}(X_2) \text{ to the output state of } \text{Tr}(X_1). \text{Moreover, the initial/final states of } \text{Tr}(X_1), \text{Tr}(X_2) \text{ are identified, respectively. The identities for the horizontal composition are } \text{Id}^\#_n : (0,m) \rightarrow (0,m), \text{i.e., modules that have no temporal input and the spatial input is left unchanged.}
\]

An example:

\[
\{ x\text{:Int}; t:t\text{Int}; \text{t=x}; \text{t} \}
\{ x=x/2; \} \{ \text{t=x}; \text{t} \}
\]

The type is \( 0|\{x\} \rightarrow t|\{t|x\} \), where again each variable should be replaced by its type.

\[
\text{Diagonal composition. The translated rv-program } \text{Tr}(X_1 SX_2) \text{ consists of nine parts: the translated rv-programs of the arguments } \text{Tr}(X_1), \text{Tr}(X_2), \text{two recorders } R_1, R_2, \text{two speakers } S_1, S_2, \text{an identity } \text{Id}, \text{and two empty modules } \Lambda_1, \Lambda_2. \text{The initial state and class of the program are those of } \text{Tr}(X_1), \text{while the final ones are those of } \text{Tr}(X_2). \text{In order to preserve the temporal and the spatial output of } \text{Tr}(X_1) \text{ and pass them as input to } \text{Tr}(X_2), \text{the temporal output of } \text{Tr}(X_1) \text{ is recorded using } R_1 \text{ (the values of the temporal variables are assigned to spatial variables) and it is read}
\]

\[
\text{sx:spaceInput ; tx:tempInput } (A,1) \text{ cod } M; \text{write } sx; \text{speak } tx; \}
goto(D,2);
\]
Fig. 3. The diagonally composition of two programs \( X \) and \( Y \).

using \( S_1 \) (the values of the spatial variables are assigned to temporal variables). In a similar manner, the spatial output of \( \text{Tr}(X_1) \) is read using \( S_2 \) and it is recorded using \( R_2 \). We use two different recorders \( R_1, R_2 \) and two different speakers \( S_1, S_2 \) because these blocks may have different types. We must also use an appropriate identity \( \text{Id} \) to make the connection between the registers and speakers above. The control labels of these blocks allow the building of the scenarios of \( \text{Tr}(X_1), \text{Tr}(X_2) \) according to the following formula, where \( |P| \) denotes the scenarios of an rv-program \( P \) (the formula is illustrated in Fig. 3):

\[
|\text{Tr}(X_1)| \perp |\text{Tr}(X_2)| = \{ |\text{Tr}(X_1)| \triangleright R_1 \triangleright A_1 \} \cdot \{ S_2 \triangleright \text{Id} \triangleright R_2 \} \cdot \{ A_2 \triangleright S_1 \triangleright |\text{Tr}(X_2)| \}.
\]

The identities for this composition are the modules \( \text{Id}_p^q : (m, p) \to (m, p) \) whose temporal (spatial) output is the same with its temporal (spatial) input.

4.2 General interfaces

The above definition is easily extended to the case of general interfaces on the interacting interfaces \((x_1, \ldots, x_n); (y_1, \ldots, y_n); \ldots; (z_1, \ldots, z_p)\). One may use a label, say \( A \), for this interface and access its components by the notation \( A.1, A.2, \ldots \) (for \((x_1, \ldots, x_m),(y_1, \ldots, y_n), \ldots\)). Below we explain the procedure in the case of simple modules.

We consider two programs \( P_1 : (m_1, p_1) \to (n_1, q_1) \), \( P_2 : (m_2, p_2) \to (n_2, q_2) \) with the following decomposition on modules:

\[
P_1 = \{ X_1^1 \# X_1^2 \# \ldots \# X_1^n \}; \{ X_2^1 \# X_2^2 \# \ldots \# X_2^n \}; \ldots; \{ X_m^1 \# X_m^2 \# \ldots \# X_m^n \},
\]

\[
P_2 = \{ Y_1^1 \# Y_1^2 \# \ldots \# Y_1^n \}; \{ Y_2^1 \# Y_2^2 \# \ldots \# Y_2^n \}; \ldots; \{ Y_m^1 \# Y_m^2 \# \ldots \# Y_m^n \}.
\]

The horizontal composition of \( P_1 \) and \( P_2 \) is defined only if \( n_1 = m_2 \) and all the premises for the horizontal composition of modules \( X_i^j \) and \( Y_i^j \) are fulfilled for all \( 1 \leq j \leq n \). The semantics of the program \( P_1 \# P_2 \) is the semantics of the rv-program that consists in the basic blocks \( \text{Tr}(X_i^j) \) and \( \text{Tr}(Y_i^j) \) for all \( 1 \leq i \leq m \), \( 1 \leq j \leq n \). The control labels of these blocks are chosen such that when building the scenarios from the semantics of the rv-program, they may be vertically and horizontally composed as follows:

\[
\{ |\text{Tr}(X_1^1)| \triangleright |\text{Tr}(X_2^1)| \triangleright \ldots \triangleright |\text{Tr}(X_i^1)| \triangleright |\text{Tr}(X_1^2)| \triangleright |\text{Tr}(X_2^2)| \triangleright \ldots \triangleright |\text{Tr}(X_i^2)| \} \cdot \{ |\text{Tr}(X_1^n)| \triangleright |\text{Tr}(X_2^n)| \triangleright \ldots \triangleright |\text{Tr}(X_i^n)| \triangleright |\text{Tr}(X_1^1)| \triangleright |\text{Tr}(X_2^1)| \triangleright \ldots \triangleright |\text{Tr}(X_i^1)| \}
\]

\[
\ldots
\]

\[
\{ |\text{Tr}(Y_1^1)| \triangleright |\text{Tr}(Y_2^1)| \triangleright \ldots \triangleright |\text{Tr}(Y_i^1)| \triangleright |\text{Tr}(Y_1^2)| \triangleright |\text{Tr}(Y_2^2)| \triangleright \ldots \triangleright |\text{Tr}(Y_i^2)| \} \cdot \{ |\text{Tr}(Y_1^n)| \triangleright |\text{Tr}(Y_2^n)| \triangleright \ldots \triangleright |\text{Tr}(Y_i^n)| \triangleright |\text{Tr}(Y_1^1)| \triangleright |\text{Tr}(Y_2^1)| \triangleright \ldots \triangleright |\text{Tr}(Y_i^1)| \}
\]

The vertical composition of \( P_1 \) and \( P_2 \) is defined only if \( q_1 = p_2 \) and all the premises for the vertical composition of modules \( X_i^j \) and \( Y_i^j \) are fulfilled, for all
The semantics of the program $P_1; P_2$ is obtained applying space-time duality, i.e., using the formula $((P_1^\vee)^\#(P_2^\vee))^\vee$.

The diagonal composition of $P_1$ and $P_2$ is defined only if $q_1 = p_2$, $n_1 = m_2$, and all the premises for the vertical composition of modules $X_i^p$ and $Y_i^q$ for all $1 \leq i \leq m$ and for the horizontal composition of modules $X_i^p$ and $Y_i^q$ for all $1 \leq j \leq n$ are fulfilled. The semantics of the program $P_1; P_2$ is obtained using a similar procedure as above (see also Fig.3, but now the constants (speakers, recorders, identities) are extended from simple to arbitrary interfaces.

4.3 Colecting data form various components

In order to collect the information from some specific states or classes scattered among various components we define three more types of constants, transformed recorders denoted $TR$, transformed speakers denoted $TS$ and transformed recording-speakers denoted $TRS$. The constant $TR : (m, p) \rightarrow (m + p, p)$ has the semantics $\{ (x \mid y) \mapsto (x, y) \mid y : x \in \mathbb{N}^p, y \in \mathbb{N}^p \}$; next, $TS : (m, p) \rightarrow (m, p + m)$ has the semantics $\{ (x \mid y) \mapsto (x \mid (y, x)) : x \in \mathbb{N}^m, y \in \mathbb{N}^p \}$; finally, $TRS : (m, p) \rightarrow (p, m)$ has the semantics $\{ (x \mid y) \mapsto (y \mid x) : x \in \mathbb{N}^m, y \in \mathbb{N}^p \}$.

Next we will present how we collect the information from a temporal interface formed of $n$ tuples $a_1; \ldots; a_n$. The information of each tuple is recorded using a transformed recorder and then is spoken to become a temporal input of the block that needs the collected information. Also the temporal interface is recreated using a recorder and a speaker for each tuple. Next, to collect the spatial information of $m$ tuples we use $m$ transformed speakers that read the spatial information and then a recorder registers it and pass it as spatial input to the block that needs the collected information. Notice that the spatial interface is recreated using a speaker and a recorder for each tuple in the interface. The situation when a block needs both a temporal interface and a spatial interface to be collected follows naturally combining the situations previously presented.

5 If–then–else

Syntactically, an “if” program is an expression $if(C)\{P_1\} else \{P_2\}$, where $P_1$, $P_2$ are two programs of the same type, and $C$ is an expression build up with the variables from the temporal and spatial inputs using the usual arithmetical and logical operators.

In this section we define the semantics of the “if then else” instruction. In the case we have a simple interface, the semantics is trivial: $Tr(if(C)\{P_1\} else \{P_2\})$ consists of the $rv$-programs $Tr(P_1)$, $Tr(P_2)$ and an additional block that test $C$ and passes the control to the input of the corresponding program. The semantics in the case of general interfaces is somehow complicated because we have to use specific constants\(^4\) to collect the input data from various (sub)interfaces and to test the condition before the start of the corresponding component $P_1$ or $P_2$.

\(^4\) I.e., transformed recorders, transformed speakers, and transformed recording-speakers.
Let $P = \text{if}(c) \text{ then } P_1 \text{ else } P_2$. It’s semantics contains a piece of rv-program used to collect input data. The block used to test the collected input data is denoted by $P_1;P_2$. It’s state and class types are those of $P_1$ (equal to $P_2$). This block passes the control as described above.

Example 2. Next, we will give an example of a program that tests if a point, whose coordinates are randomly generated, is in a rectangular area. The up-left point of the rectangular has coordinates $(x, y)$ and the down-right point has coordinates $(z, w)$. The first process randomly generates the coordinates $x$, $y$, $z$ and $w$, while the second and the third process generate the abscissa and the ordinate of the random point, respectively. If the point is in the rectangular area a message appears, else new values are generated.

$$\{x,y,z,w : \text{sInt};
\{x=\text{random}(0,200); y=\text{random}(0,200);
\quad z=\text{random}(200,400); w=\text{random}(200,400); \text{write } x,y,z,w;\}#
\{x1:\text{sInt}; x1=\text{random}(0,400); \text{write } x1;\}#
\{y1:\text{sInt}; y1=\text{random}(0,400); \text{write } y1;\}#
\\
\text{if } ((z<x1<x) \&\& (y<y1<w)) \text{ then }
\{\text{write}(''\text{in the rectangular is the point}'');\}#
\{\text{read } x1; \text{write}(x1);\}#\{\text{read } y1; \text{write}(y1);\}#
\text{else } \{\text{write}(''\text{the point wasn’t in the rectangular}'');\}#
\{\text{read } x1; x1=\text{random}(0,400); \text{write } x1;\}#
\{\text{read } y1; y1=\text{random}(0,400); \text{write } y1;\}#
\}$$

The test condition spans over three processes (obtained by horizontal composition) that have the coordinates of the rectangular area and of the point. For further use, denote by $P_1, P_2, P_3$ the parts of this program such that the full program is $P_1; \text{if}(C)\{P_2\} \text{else}\{P_3\}$.

An rv-program semantically equivalent with $P_1$ is:

| $X_1^1$: | $x,y,z,t : \text{sInt}$ | $X_2^1$: | $x1 : \text{sInt}$ | $X_3^1$: | $y1 : \text{sInt}$ |
|---|---|---|---|---|
| $(A1,3)$ | $\text{write}(''\text{in the}''$); | $(B,4)$ | $\text{write} (x1)$; | $(C,5)$ | $\text{write} (y1)$; |
| | $\text{write}(''\text{rectangular}''$); | | $\text{goto}(C,3)$; | | $\text{goto}(D,3)$; |
| | $\text{write}(''\text{is the point}''$); | | $\text{goto}(B,4)$; | | |

An rv-program semantically equivalent with $P_2$ is:

| $Y_1^1$: | $x,y,z,t : \text{sInt}$ | $Y_2^1$: | $x1 : \text{sInt}$ | $Y_3^1$: | $y1 : \text{sInt}$ |
|---|---|---|---|---|
| $(A2,3)$ | $\text{write}(''\text{in the}''$); | $(F,4)$ | $x1=\text{random}(0,400)$; | $(G,5)$ | $y1=\text{random}(0,400)$; |
| | $\text{write}(''\text{rectangular}''$); | | $\text{goto}(G,6.4)$; | | $\text{goto}(D,6.5)$; |
| | $\text{write}(''\text{is the point}''$); | | $\text{goto}(B,6.3)$; | | |
Finally, the block $X_1^1Y_1^1$ is defined as follows:

$$X_1^1Y_1^1: \begin{cases}
(A, 6) & x, y, z, w, x_1, y_1: \text{sInt} \\
\text{if}((z < x_1 < x) \&\& (y < y_1 < w)) \text{ goto}(A1, 6) \text{ else goto}(A2, 6);
\end{cases}$$

where $A$, $A1$ and $A2$ have the same type.

An rv-program semantically equivalent with with the full program $P$ has the initial state (class) 1 ($A$), the final state (class) 6 ($D$) and it consists in the the blocks from the semantics of $P_1$ and $P_2$ and the block $X_1^1Y_1^1$ there are described below. The constants used to collect the information for the evaluation of the condition $c$ are described in the scenario from Fig.4.

$$U:: (A, 1) \quad V:: (B, 8) \quad W:: (C, 9)$$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$x=$ random(0,200);</td>
<td>$x_1:$sInt;</td>
<td>$y_1:$sInt;</td>
</tr>
<tr>
<td>$y=$ random(0,200);</td>
<td>$x_1=$ random(0,200);</td>
<td>$y_1=$ random(0,200);</td>
</tr>
<tr>
<td>$z=$ random(0,200);</td>
<td>goto(C, 2.4);</td>
<td>goto(D, 2.5);</td>
</tr>
<tr>
<td>$w=$ random(0,200);</td>
<td>goto(B, 2.3);</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4. A scenario for $P$ (when the condition is true and $P_1$ is executed).

6 Spatio-temporal structured programs

Spatio-temporal structured programming uses three types of while instructions: the temporal while, while_t, which corresponds to the while from the classical structured programming and it is used for iteration on columns; the spatial while, while_s, which is used for iteration on rows; and the spatio-temporal while, while_st, which iterates on both time and space.

6.1 Temporal while

The significance of while_t(c){P} is as long as condition C is satisfied, P is executed. Between iterations only the spatial variables are transmitted and there is
no correspondence between the temporal output of one iteration and the temporal input of the next one. The semantics of $\text{while}_t(c)\{P\}$ is the rv-program which consists of 3 components: the rv-program associated to $P$, the preprocessing rv-program used to collect the spatial input data, and a block that tests the condition and passes the control either to $P$, or stops the execution passing the control to an exit block. A simpler version is the case when the condition apply to the first tuple of the north interface; in such a case the the preprocessing rv-program used to collect the input data is not needed.

Example 3. Next, we will give an example of a structured rv-program that computes if a given number is perfect or not. The input $x$ of the program is the number to be checked, and the output, $z$ is 1 if $x$ is perfect or 0 if not.

In: $x$; Out: $z$;

```
x,y,z : sInt; t : tInt;
{
  {t=x; x=x/2;}#
  {y=t;}#
  {z=t;}#
}
while $t(x>0)$
{
  {t=x; x=x-1;}#
  {if((y%t==0){t=0};)}#
  {z=z-t;}#
}

Q =
{
  {read x; t=x; x=x/2; speak t; write x;}#
  {listen t; y=t; speak t; write y;}#
  {listen t; z=t; write z;}#
}
and $S = \text{while}_t(x>0){R}$, where $R =$
{
  {read x; t=x; x=x-1; speak t; write x;}#
  {read y; listen t; if(!(y%t==0){t=0};}#
  {speak t; write y;}#
  {read z; listen t; z=z-t; write z;}#
}
```

Fig. 5. A structured rv-program for perfect numbers (a); in (b), the fully specified version of (a)

6.2 Spatial while

Using space-time duality we define the spatial while as being the dual instruction of the temporal while. The meaning of $\text{while}_s(C)\{P\}$ is as long as condition $C$ is satisfied, the program $P$ is executed. Between iterations only the temporal variables are transmitted and there is no correspondence between the spatial output of one iteration and the spatial input of the next one. The semantic of this statement is obtained using space-time duality. Notice that the recorders and the speakers are dual constants. The same holds for the transformed speakers and the transformed recorders.

6.3 Spatio-temporal while

The significance of $\text{while}_s\text{p}(C)\{P\}$ is as long as condition $C$ is satisfied, the program $P$ is executed. This type of while is different from the previous ones in the sense that the temporal (spatial) output of one iteration is the temporal (spatial) input of the next iteration.

The semantics of $\text{while}_s\text{p}(C)\{P\}$ is the rv-program which consists of 3 components: the rv-program associated to $P$, the preprocessing rv-program used to
collect the spatial and temporal input data, and a block that tests the condition and passes the control either to \( P \), or stops the execution passing the control to an exit block.

Again, a simpler version is the case when the condition uses the variables of the first tuple of the north interface and the first tuple of the west interface. In such a case there is no need to collect the temporal and spatial input of the program or the temporal and spatial output of an iteration. Hence, the program \( P \) is repeatedly diagonally composed with himself as long as condition \( C \) is true.

**Example 4.** Next we will present an example of a structured rv-program for an algorithm that tests the termination of a ring of processes.

\[
\begin{align*}
\{ & \text{Init}_0 \# \text{while}(\text{t}_\text{id} < \text{t}_n) \text{Init} \}$\{ & \text{Start} \# \text{while}(\text{t}_\text{id} < \text{t}_n) \text{C} \}$\{ & \text{while} \text{sp}(!((\text{tlist}[n+1]==0) \&\& (\text{tlist}[n]==1))) \}$\{ & \text{C}_0 \# \text{while}(\text{t}_\text{id} < \text{t}_n) \text{C} \}$
\end{align*}
\]

The module \( \text{Init}_0 \) creates the first process in the ring and \( \text{Init} \) creates the rest of the processes (their number is \( t_n \)). The module \( \text{Start} \) creates a token and colors it white if the first process has terminated or else it colors it black. Module \( \text{C} \) contains the code executed by each process. If a processes has the token (\( \text{tlist}[n+1] \) denotes the process that has the token) and has terminated, it passes the token to the next process (if it had reactivated other processes it colors the token black). The code inside module \( \text{C}_0 \) is executed by the first process. If it has received a white token the execution stops (this is the condition of the \( \text{while} \text{sp} \)).

7 Extensions: repeat, forall

The **repeat** statement is just a programming construct to denote Kleene closure of a composition, hence we have three types of **repeat** statements: **repeat_t**, **repeat_s**, **repeat_st** for vertical, horizontal, and diagonal compositions. It is a nondeterministic operation and should be avoided if we want to have deterministic programs (deadlock may appear).

The **forall** statement is somehow similar, but the number of repetitions is determined be the number of inputs and/or outputs into/from this module. An “in” version, safe for a horizontal composition with a module on left \( X \overset{\text{forall \_in \_t}}{\leadsto} P \), is to repeat \( P \) as long as inputs are feed-ed into the component of \( P \). An “out” version, is to repeat \( P \) as long as its output may be accepted by the next component - this is safe for a composition on right \( \text{forall \_out \_t}(P) \overset{\text{in}}{\leadsto} X \).

**Example 5.** A new structured rv-programs for perfect numbers is described below. This second version corresponds to the column decomposition of the running scenarios. It uses the **forall** statement.

\[
\begin{align*}
\text{In: } & x; \text{Out: } z; x,y,z \ : \text{Int}; t \ : t\text{Int} \\
\{ & t=x; \text{ speak } t; \ x=x/2; \ \text{ while}\_t(x>0)\{t=x; \text{ speak } t; \ x=x-1;\}\}# \\
\{ & \text{listen } t; \ y=t; \ \text{ speak } t; \ \text{forall}\_\text{in}\_t\{\text{listen } t; \ \text{if}((y%t==0))\{t=0; \text{ speak } t;\}\}\}# \\
\{ & \text{listen } t; \ z=t; \ \text{forall}\_\text{in}\_t\{\text{listen } t; \ z=z-t;\}\}#
\end{align*}
\]

The program is deterministic, as the processes with **forall** \( P_2, P_3 \) are completely determined by the first process in a structure \( P_1 \# P_2 \# P_3 \).
8 Conclusions

We have shown how structured programming techniques used for the classical sequential programs may be adapted to obtain structured rv-programs. Particular emphasis is on developing a structural spatial programming discipline, which provides a more structured interaction between objects and it is of great help in simplifying the construction and the analysis of interactive programs. The spatio-temporal while is interesting to express object oriented programs. The correctness of classical structured sequential programs may be adapted to prove the correctness of structured interactive rv-programs [5]. The interaction of the temporal while and the spatial while with the environment complicates a lot their structures. As future work we plan on develop the basic statements presented in this paper.

References

9 Appendix

Example 6. We provide a scenario for the program \( P = \text{if}(c) \text{then } P_1 \text{ else } P_2 \) when \( c \) is true and \( P_1 \) is executed:

Example 7. To provide a better intuition for the while we will give, in the left of Figure 7, a scenario from the semantics of “while \( t(c) \{P\} \)”, where \( P \) has a decomposition of \( 2 \times 2 \) modules and \( c \) becomes false after the first iteration. In the right there is an example in the case the input collection is not necessary.

Example 8. Figure 8 presents a sub-scenario for the program \( P' = \text{while}\_st(c) \{P\} \) when \( P \) is a program that has a decomposition of \( 2 \times 2 \) modules.

Fig. 6. A scenario for \( P = \text{if}(c) \text{then } P_1 \text{ else } P_2 \) when \( c \) is true and \( P_1 \) is executed.
Fig. 7. A sub-scenario for while_t(c){P}

Fig. 8. A sub-scenarios for while_st(c){P}