Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 203 (2008) 69–94
www.elsevier.com/locate/entcs

AGAPIA v0.1: A Programming Language for Interactive Systems and Its Typing System

Cezara Dragoi[1,2] and Gheorghe Stefanescu[1]

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

Abstract

A model (consisting of *rv-systems*), a core programming language (for developing *rv-programs*), several specification and analysis techniques appropriate for modeling, programming and reasoning about interactive computing systems have been recently introduced by Stefanescu using register machines and space-time duality, see [47]. After that, Dragoi and Stefanescu have developed structured programming techniques for rv-systems and their verification, see, e.g., [14,15,16,17,18].

In the present paper a kernel programming language AGAPIA v0.1 for interactive systems is introduced. The language contains definitions for complex spatial and temporal data, arithmetic and boolean expressions, modules, and while-programming statements with their temporal, spatial, and spatio-temporal versions. In AGAPIA v0.1 one can write programs for open processes located at various sites and having their temporal windows of adequate reaction to the environment. The main technical part of the paper describes a typing system for AGAPIA v0.1 programs.

Keywords: interactive systems, typing systems, AGAPIA programming, rv-programs, registers and voices, distributed termination protocols

1 Introduction

Interactive computation has a long tradition and there are many successful approaches to deal with the intricate aspects of this type of computation, see [1,2,3,7,9,19,28,49,50], to mention just a few references from a very rich literature. However, a simple model for interactive computation extending the classical, popular imperative programming paradigm is still to be find.

In an attempt to reconcile interactive and imperative computation styles, a model based on register machines and space-time duality have been recently proposed by Stefanescu, see [47]. Based on this model, a low level programming lan-

[1] The authors acknowledge partial support from the Romanian Ministry of Education and Research (CEEX Program, Project 2-CEx06-11-97 and PNCDI-II Program 4, Project 11052/18.09.2007).

[2] *Current address:* LIAFA, Universite Paris Diderot - Paris 7, France

doi:10.1016/j.entcs.2008.04.087

guage for writing interactive programs with registers and voices (rv-programs) have been presented [47]. One of the key features of the model is the introduction of high-level temporal data structures. Actually, having high level temporal data on interaction interfaces is of crucial importance in getting a compositional model for interactive systems, a goal not always easy to achieve (recall the difficulties in getting a compositional semantics for data-flow networks, [4,5,6,8,29]).

In a couple of papers Dragoi and Stefanescu have developed structured programming techniques for rv-programs and for their verification, see [14,15,16,17,18]. Here, a kernel structured programming languages **AGAPIA v0.1** for interactive systems is introduced and its typing system is studied. [3]

A first goal of the present paper is to introduce AGAPIA v0.1 language. An example is `P` in Sec. 3, an AGAPIA v0.1 program implementing a termination detection protocol. Here, we briefly touch on the key features of the language, with explicit reference to `P`.

The starting basic blocks for developing AGAPIA programs are "modules" inherited from rv-programming. Such a module has two types of interfaces: one type is for spatial data, the other is for temporal data. A spatial interface is specified using registers, while a temporal interface is specified using voices, usually implemented on streams. Complex spatial and temporal data are built up on top of these primitive types. A module has explicit `read/write` and `listen/speak` statements for its spatial and temporal data, respectively. An example of module is `R` in program `P`.

The AGAPIA v0.1 structured programming operations extend the classical structured programming operations to this context. Composition has extensions to AGAPIA which exploit the multiple possibilities to compose blocks: (1) Vertical (or "temporal") composition via spatial interfaces; (2) Horizontal (or "spatial") composition via temporal interfaces; (3) Diagonal (or "spatio-temporal") composition, where both, the spatial and the temporal output data of a block become the spatial and the temporal input data of the next block, respectively. *Temporal, spatial*, and *diagonal compositions* are denoted by '`%`', '`#`', and '`$`', respectively. The iterated versions are introduced using the *temporal*, the *spatial*, and the *diagonal while* statements, denoted by `while_t`, `while_s`, and `while_st`, respectively. Occurrences of most of these statements may be found in program `P`.

While not subject of the present paper, notice that AGAPIA v0.1 language has a natural scenario-based operational semantics, as well as a compositional relational semantics based on spatio-temporal specifications (see [17,18] or Subsec. 2.3 for brief presentations).

A second goal of the paper is to study the typing system of AGAPIA v0.1. Our particular interest is to use the results for the design of AGAPIA v0.1 compilers. While it is the programmer duty to ensure the correctness of his/her program, our type checking procedure help him/her by checking the types of the programs and

[3] The current low level version number reflects the restricted format of the language, particularly a restricted form of mixing structured rv-programming statements and module construction. (See [37] for the new AGAPIA v0.2 version, including stronger modularization techniques.)

returning a four-level answer: (1) `ok` (from the typing point of view the program is correct); (2) `war0` (at each composing interface the sets of types of the composing programs are equal, but not reduced to a singleton, hence a running time miss-typing is possible); (3) `war1` (at each composing interface the sets of types of the composing programs have nonempty intersection, hence there is a chance to have a well running program); and (4) `err` (at a composing interface the sets of types of the composing programs have an empty intersection, hence each running using that piece of code fails).

The paper is organized as follows. It starts with a brief presentation of rv-systems, including: scenarios, operations on scenarios, finite interactive systems, rv-systems and rv-programs. Then, structured rv-programs are introduced and their scenario-based operational semantics is presented. Next, the syntax of AGAPIA v0.1 is presented and an implementation of a termination detection protocol is developed. A more technical section follows, describing the typing systems for AGAPIA v0.1 programs. A few examples and final comments conclude the paper.

2 Preliminaries

2.1 Scenarios

In this subsection we briefly present temporal data, spatio-temporal specifications, grids, scenarios, and operations on scenarios.

Spatio-temporal specifications

To handle spatial data, common data structures and their natural representations in memory are used. For the temporal data, we use streams: a *stream* is a sequence of data ordered in time and is denoted as $a_0{}^\frown a_1{}^\frown \ldots$, where $a_0, a_1, \ldots$ are its data at time $0, 1, \ldots$, respectively. Typically, a stream results by observing the data transmitted along a channel: it exhibits a datum (corresponding to the channel type) at each clock cycle. (See [7,8] for more on streams, temporal specifications, and their algebraic representations.)

A *voice* is defined as the time-dual of a register: *A voice is a temporal data structure that holds a natural number. It can be used ("heard") at various locations. At each location it displays a particular 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 time and their length. Most of usual data structures have natural temporal representations. Examples include timed booleans, timed integers, timed arrays of timed integers, timed linked lists, etc.

When restricted to registers and voices only, a *spatio-temporal specification* $S : (m, p) \to (n, q)$ is a relation $S \subseteq (\mathbb{N}^m \times \mathbb{N}^p) \times (\mathbb{N}^n \times \mathbb{N}^q)$, where m (resp. p) is the number of input voices (resp. registers) and n (resp. q) is the number of output voices (resp. registers). It may be defined as a relation between tuples, written as $\langle v \mid r \rangle \mapsto \langle v' \mid r' \rangle$, where v, v' (resp. r, r') are tuples of voices (resp. registers).

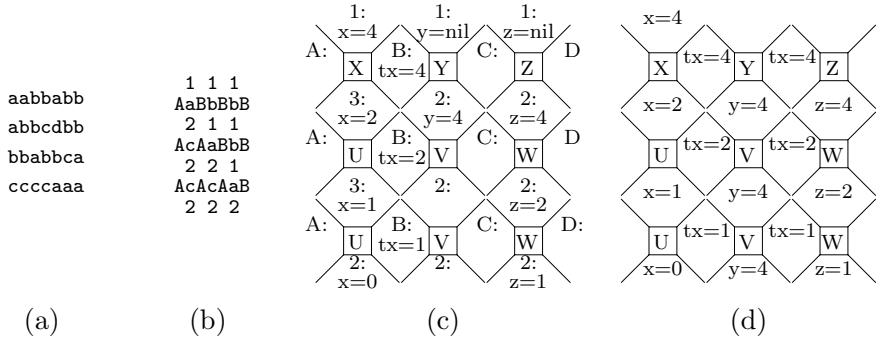


Fig. 1. A grid (a), an abstract scenario (b), and concrete scenarios (c,d).

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

Grids and scenarios

A *grid* is a *rectangular* two-dimensional area filled in with letters of a given alphabet. An example of a grid is presented in Fig. 1(a). 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 it resumes its execution.

A *scenario* is a grid enriched with data around each letter. The data may have various interpretation: they either represent control/interaction information, or current data of the variables, or both. Fig. 1(b) illustrates the first case, Fig. 1(c) the last case, and Fig. 1(d) the middle case. Notice that the scenario from Fig. 1(d) is similar to that in (c), but the control/interaction labels A,B,C,1,2,3 are omitted. The scenarios of a rv-program look like in (c), while those of structured rv-programs as in (d) - there are no control/interaction labels in the latter case.

The type of a scenario interface is represented as $t_1; t_2; \ldots; t_k$, where each t_k is a tuple of simple types used in the scenario cells. [4] An empty tuple is also written 0 or *nil* and can be freely inserted to or omitted form such descriptions. The type of a scenario f is specified by the notation $f : \langle w|n\rangle \to \langle e|s\rangle$. For the example in Fig. 1(d), the type is $\langle nil; nil; nil|sn; nil; nil\rangle \to \langle nil; nil; nil|sn; sn; sn\rangle$, where sn denotes the spatial integer type.

Operations with scenarios

We say two scenario interfaces $t = t_1; t_2; \ldots; t_k$ and $t' = t'_1; t'_2; \ldots; t'_{k'}$ are *equal* if $k = k'$ and the types and the values of each pair t_i, t'_i are equal. Two interfaces are *equal up to the insertion of nil elements*, written $t =_n t'$, if one can insert *nil* elements into these interfaces such that the resulting interfaces are equal.

[4] If only registers and voices are used, then one may simply replace each tuple by the number of its components.

We denote by $Id_{m,p} : \langle m|p\rangle \to \langle m|p\rangle$ the identity constant, i.e., the temporal/spatial output is equal to the temporal/spatial input, respectively.

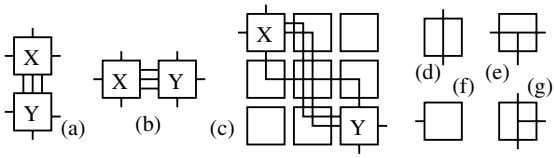


Fig. 2. Operations on scenarios

Horizontal composition: Suppose we start with two scenarios $f_i : \langle w_i|n_i\rangle \to \langle e_i|s_i\rangle$, $i = 1, 2$. Their *horizontal composition* $f_1 \triangleright f_2$ is defined only if $e_1 =_n w_2$. For each inserted *nil* element in an interface, a dummy row is inserted in the corresponding scenario, resulting a scenario $\overline{f_i}$. After these transformations, the result is obtained putting $\overline{f_1}$ on left of $\overline{f_2}$. Notice that $\overline{f_1} : \langle w_1|n_1\rangle \to \langle t|s_1\rangle$ and $\overline{f_2} : \langle t|n_2\rangle \to \langle e_2|s_2\rangle$, where t is the resulting common interface. The result, $f_1 \triangleright f_2 : \langle w_1|n_1; n_2\rangle \to \langle e_2|s_1; s_2\rangle$, is unique up to insertion or deletion of dummy rows. See Fig. 3 (and Fig. 2(b)). Its identities are $Id_{m,0}$.

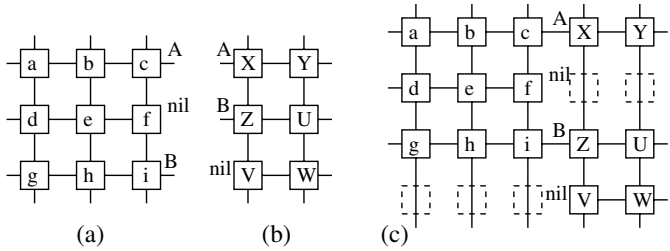


Fig. 3. Horizontal composition of scenarios

Vertical composition: The definition of *vertical composition* $f_1 \cdot f_2$ is similar, but now $s_1 =_n n_2$. For each inserted *nil* element, a dummy column is inserted in the corresponding scenario, resulting a scenario $\overline{f_i}$. The result, $f_1 \cdot f_2 : \langle w_1; w_2|n_1\rangle \to \langle e_1; e_2|s_2\rangle$, is obtained putting $\overline{f_1}$ on top of $\overline{f_2}$. See Fig. 2(a). Its identities are $Id_{0,m}$.

Constants: Except for the already defined identities I, additional constants may be used. Some of them may be found in Fig. 2: A recorder R (2nd cell in the 1st row of (c)), a speaker S (1st cell in the 2nd row of (c)), an empty cell Λ (3rd cell in the 1st row of (c)), etc.

Diagonal composition: The *diagonal composition* $f_1 \bullet f_2$ is defined only if $e_1 =_n w_2$ and $s_1 =_n n_2$. It is a derived operation defined by

$$f_1 \bullet f_2 = (f_1 \triangleright R_1 \triangleright \Lambda_1) \cdot (S_2 \triangleright Id \triangleright R_2) \cdot (\Lambda_2 \triangleright S_1 \triangleright f_2)$$

for appropriate constants R, S, Id, Λ. See Fig. 2(c). In this case $R_1 : \langle t| \ \rangle \to \langle \ |t\rangle$, $S_1 : \langle \ |t\rangle \to \langle t| \ \rangle$, $Id : \langle u|t\rangle \to \langle u|t\rangle$, $R_2 : \langle u| \ \rangle \to \langle \ |u\rangle$, $S_2 : \langle \ |u\rangle \to \langle u| \ \rangle$, where t (resp. u) is a common representation for e_1 and w_2 (resp. s_1 and n_2) obtained inserting *nil* elements. Its identities are $Id_{m,n}$.

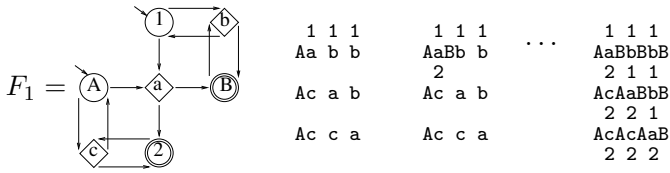
2.2 Rv-systems

In this subsection we briefly describe rv-programs (interactive programs with registers and voices); see [47] for more details.

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: the first type of vertices is for *states* (labeled by numbers), the second is for *classes* (labeled by capital letters) and the edges/transitions are labeled by letters denoting the atoms of the grids; each transition has two incoming arrows (one from a class and the other from a state), and two outgoing arrows (one to a class and 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., [46,47]. 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 only, then 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 `abb` `cab` `cca` are shown below.



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 (that is, for the letters labeling the edges).

We study programmable rv-systems specified using *rv-programs*. An example of rv-program is presented in Fig. 4. A computation is described by a scenario like in a FIS, but with data around each cell (the data represent values of their spatial and temporal variables); see Fig. 1(c) for an example.

Syntax of rv-programs

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. 4. 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)

in: A,1; out: D,2

X::

(A,1)	x : sInt

	tx : tInt;
	tx = x;
	x = x/2;
	goto [B,3];

W::

(C,2)	z : sInt
tx : tInt	z = z - tx; goto [D,2];

U::

(A,3)	x : sInt

	tx : tInt;
	tx = x; x = x - 1;
	if (x > 0) {goto [B,3]}
	else {goto [B,2];}

Y::

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

V::

(B,2)	y : sInt

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

Z::

(C,1)	z : sInt
tx : tInt	z = tx; goto [D,2];

Fig. 4. The rv-program **Perfect** (for perfect numbers)

The body of a module is its bottom-right part, including type declarations and C-like code. The exit from the module is specified 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 and denotational semantics of rv-programs

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**:

(1) Each cell has a module name as label.

(2) In the areas around a cell we show how variables are modified.

(3) In a current cell, the values of spatial variables are obtained going vertically up and collecting the last updated values.

(4) Similarly, the full information on temporal variables in a current cell is obtained collecting their last updated values going horizontally on left.

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

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

(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**.

The *input-output denotation* of an rv-program is the relation between the input data on the north/west borders and output data on the south/east borders of the program scenarios.

Notice that a global scoping rule is implicitly used here: once defined, a variable is always available. It is also possible to introduce rv-programs obeying a stronger typing discipline, where each module comes with an explicit type at each border. This option is actually used for structured programs to be introduced in the next subsection.

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* $^\vee$ 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 $^\vee$ to the underlying 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).

Theorem 2.1 *For any rv-program R, its space-time dual $R^\vee$ is an rv-program and $(R^\vee)^\vee = R$. Moreover, space-time duality respects operational semantics and input-output denotation of rv-programs.*

2.3 Structured rv-programs

The rv-programs, presented in [47] (and briefly recalled in the previous subsection), resemble flowcharts and assembly languages: one freely uses `goto` statements, with both temporal and spatial labels. The aim of this section is to introduce structured programming techniques on top of rv-programs. The resulting structured rv-programs may be described directly, from scratch. The lower level of rv-programs is used as a target language for compiling.

The syntax of structured rv-programs

The syntax is given by the BNF grammar

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

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

Structured rv-programs use modules X as their basic blocks. On top of them, larger programs are built up by "if" and composition and iteration constructs for the vertical, the horizontal, and the diagonal directions [5]. These statements aim to capture at the program level the corresponding operations on scenarios.

On structured programming operations

Our choice of the above structured programming statements is determined by practical considerations: to have a set of easy to understand and use statements. Actually, these three types of composition and iterated composition statements are instances of a unique more general and complex, but less "structured" form:

$$P1\ comp\{tv\}\{sv\}\ P2 \quad \text{and} \quad while\{tv\}\{sv\}\{C\}\{P\}$$

In this case, at an identification border, only a part of the connecting interfaces is to be matched, namely the tv part at a temporal interface and the sv part at a spatial interface. (The operations are illustrated in Fig. 5; see [43] for translations between "while" and "feedback" operations.)

Then, the horizontal form correspond to the choice $tv = all$, $ts = \emptyset$, the vertical form to $tv = \emptyset$, $ts = all$, and the diagonal form to $tv = all$, $ts = all$. To state it formally,

Proposition 2.2 *Either* $\{comp\}$ *or the pair* $\{\#, \%\}$ *suffices to represent all composition operations in* $\{\#, \%, \$, comp\}$*, provided constants as those in Fig. 2 may be used. All particular iteration operators in* $\{while\_t, while\_s, while\_st\}$ *are instances of the general* while *operator.*

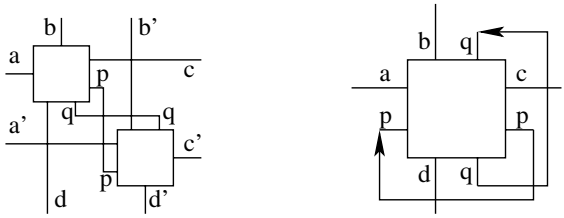


Fig. 5. General "composition" and "feedback" (equivalent to "while")

Currently, we do not know whether the general *while* may be simulated using the particular forms in $\{while\_t, while\_s, while\_st\}$.

Examples

We include a simple, but rather general example of structured rv-program to give a clue to the reader on the naturalness and the expressiveness of the language. (See [14,17,18] for more examples.)

[5] The iteration operators are also called *the temporal, the spatial*, and *the spatio-temporal while* statements.

The structured rv-program for a termination detection protocol, to be presented in full detail in a next section, has the following format

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

It starts with an initialization step where processes are created and inserted into the ring. Next, an autonomous "diagonal" iteration takes places where, in each step, the processes do their jobs and interact horizontally by passing a message list from one process to the next, in the order from process 0 to process `tn-1`. At the end of an iteration, if the guard condition is fulfilled, a new iteration takes place where the message list of process `tn-1` is passed to process 0 and all processes continue the execution from their last memory states.

Such a program is rather generic and it may be used for many other problems, like n-player games, 8-queen problem, implementations of OO-systems based on message passing communication, etc.

The dynamic case where processes may freely join or leave the ring is an easy extension using the general while. By replacing in the above code `while_st` by

```
while{all\k}{all}
```

we get a program where `k` is a temporal variables coming from the external temporal interface, not from `R`; this `k` may be used to specify the number of new processes that have to be inserted into the ring, a procedure that is handled as in `I2`. (A slightly different approach is presented in [37] using high-level structured rv-programs.)

2.4 Operational semantics of structured rv-programs

The operational semantics

$$|\ \ | : \text{Structured rv-programs} \to \text{Scenarios}$$

associates to each program the set of its possible running scenarios.

The type of a program P, which is denoted by $P : \langle w(P)|n(P)\rangle \to \langle e(P)|s(P)\rangle$, indicates the types at its west, north, east, and south border. On each side, the type may be quite complex (see AGAPIA interface types in Sec. 3). In their specification, the convention is to separate by "," the data coming from within a module and by ";" the data coming from different modules. This convention refer to both, spatial data (coming from different processes) and temporal data (coming from different transactions).

Two interface types *match* if they have a nonempty intersection.

Modules

The starting blocks for building structured rv-programs are the modules. The `listen (read)` instruction is used to get the temporal (spatial) input and the `speak (write)` instruction to return the temporal (spatial) output. The `code` consists in

simple instructions as in the C code. No distinction between temporal and spatial variables is made within a module.

A scenario for a module consists of a unique cell, with particular data on the borders, and such that the output data are obtained from the input data applying the module code.

Composition

Due to their two dimensional structure, programs may be composed horizontally and vertically, as long as their types on the connecting interfaces agree. They can also be composed diagonally by mixing the horizontal and the vertical compositions.

Suppose two programs $P_i : \langle w_i | n_i \rangle \to \langle e_i | s_i \rangle$, $i = 1, 2$ are given. We define the following composition operators.

Horizontal composition: $P_1 \# P_2$ is defined if the interfaces e_1 and w_2 match. The type of the composite is $\langle w_1 | n_1; n_2 \rangle \to \langle e_2 | s_1; s_2 \rangle$. A scenario for $P_1 \# P_2$ is a horizontal composition of a scenario in P_1 and a scenario in P_2.

Vertical composition: $P_1 \% P_2$ is similarly defined.

Diagonal composition: $P_1 \$ P_2$ 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 if each pair of interfaces e_1, w_2 and s_1, n_2 matches. The type of the composite is $\langle w_1 | n_1 \rangle \to \langle e_2 | s_2 \rangle$. A scenario for $P_1 \$ P_2$ is a diagonal composition of a scenario in P_1 and a scenario in P_2.

If

Given two programs $P_i : \langle w_i | n_i \rangle \to \langle e_i | s_i \rangle$, $i = 1, 2$, a new program $Q = if\ (C)\ then\ P_1\ else\ P_2$ is constructed, for a condition C involving both, the temporal variables in $w_1 \cap w_2$ and the spatial variables in $n_1 \cap n_2$. The type of the result is $Q : \langle w_1 \cup w_2 | n_1 \cup n_2 \rangle \to \langle e_1 \cup e_2 | s_1 \cup s_2 \rangle$.

A scenario for Q is a scenario of P_1 if the data on west and north borders of the scenario satisfy condition C, otherwise it is a scenario of P_2.

While

We have introduced three while statements, each being the iteration of a corresponding composition operation.

Temporal while: For a program $P : \langle w | n \rangle \to \langle e | s \rangle$, the statement $while\_t\ (C)\{P\}$ is defined if the interfaces n and s match and C is a condition on the spatial variables in $n \cap s$. The type of the result is $\langle (w;)^* | n \cup s \rangle \to \langle (e;)^* | n \cup s \rangle$. A scenario for $while\_t\ (C)\{P\}$ is either an identity (if C is false), or a repeated vertical composition $f_1 \cdot f_2 \cdot \ldots \cdot f_k$ of scenarios for P, such that the north border of each f_i satisfies C, while the south border of f_k does not satisfy C.[6]

Spatial while: The spatial version $while\_s\ (C)\{P\}$ is similar.

[6] When the body program P of a temporal while has dummy temporal interfaces, the temporal while is the same as the usual while from imperative programming languages.

Spatio-temporal while: If $P : \langle w|n\rangle \to \langle e|s\rangle$, the statement $while\_st\ (C)\{P\}$ is defined if each pair of interfaces w, e and n, s matches and C is a condition on the temporal variables in $w \cap e$ and the spatial variables in $n \cap s$. The type of the result is $\langle w \cup e|n \cup s\rangle \to \langle w \cup e|n \cup s\rangle$. A scenario for $while\_st\ (C)\{P\}$ is either an identity (if C is false), or a repeated diagonal composition $f_1 \bullet f_2 \bullet \ldots \bullet f_k$ of scenarios for P, such that the west and north border of each f_i satisfies C, while the east and south border of f_k does not satisfy C.

3 The AGAPIA v0.1 language

The syntax of the AGAPIA v0.1 programs and an example are presented in this section. Extended comments, a type checking procedure, more examples are all deferred for the next section.

3.1 The syntax

Interfaces

$$
\begin{aligned}
SST ::=\ & nil \mid sn \mid sb \\
& \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^* \\
ST ::=\ & (SST) \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^* \\
STT ::=\ & nil \mid tn \mid tb \\
& \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^* \\
TT ::=\ & (STT) \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^*
\end{aligned}
$$

Expressions

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

Programs

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

Fig. 6. The syntax of AGAPIA v0.1 programs

The syntax of the AGAPIA v0.1 programming language is presented in Fig. 6. This version of the language is intentionally kept simple to illustrate the key features of our approach.

With respect to the scoping discipline, we depart here from the global "once declared, always available" discipline used in rv-programs [47]. Here, via the `module`

construct, one is able to discards variables, as well. For instance, a newly declared temporal variable which is not present in the `speak` statement of a module is discarded from its temporal interface.

The types for spatial interfaces are built up starting with integers and booleans, denoted sn, sb, and applying $\cup, `,', (\_)^*$ to get process interfaces, and then applying $\cup, `;', (\_;)^*$ to get system interfaces. They look slightly too complicate. An argument presented in Example 4.1 shows that whenever `if` and the temporal, spatial, and spatio-temporal `composition` and `while` statements are legitimate programming constructs, we have to allow such types. Similarly, the temporal types are introduced. Examples are included in the next section.

In practical programs, the description of data types will follow a more conventional approach: Star defines an array, hence the usual `[ ]` notation will be used. For union types, the "`or`" keyword will be used. Finally, the "," and ";" product types are specified using the record notation, with items separated by "," and ";", respectively.

Given a type V, the notations $V(k), V.k, V.[k], V@k, V@[k]$ are used to refer to its components. For instance, in the case of spatial interfaces, they refer to: $V(k)$- a component of a choice; $V.k$ - a component of a tuple within a process; $V.[k]$ - a component of an iterated tuple within a process; $V@k$ - a component of a tuple of processes; and $V@[k]$ - a component of an iterated tuple of processes.

Expressions, usual while programs, modules, and structured rv-programs are naturally introduced. This v0.1 version of AGAPIA has a strongly restricted format: the module and the rv-programming statements are not mixed (see [37] for a new, more powerful release: AGAPIA v0.2.). The development starts with simple while programs, then modules are defined, and finally AGAPIA v0.1 programs are obtained applying structured rv-programming statements on modules. [7]

Finally, notice that the language is space-time invariant. This means, one can formally define a space-time duality operator which maps an AGAPIA v0.1 program P to an AGAPIA v0.1 program $P^\vee$ such that $P = P^{\vee\vee}$.

A useful derived statement, to be used in the sequel, is a simple form of a spatial "for" statement

```
for_s(i=a;i<b;i++){R}
```

This is a macro for the pure AGAPIA v0.1 program

```
i=a# while_s(i<b){R# i++#}
```

where `i=a` or `i++` denotes a module with such a code, with empty spatial interfaces, and whose temporal interfaces are equal to the temporal interfaces of `R` (we suppose, `i` is included in the temporal interfaces of `R`).

[7] Notice that the general while (presented in Subsec. 2.3) is not included in this version.

3.2 An example: Dual-pass termination detection protocol

We describe a slightly extended[8] AGAPIA v0.1 program `P` that implements a *dual pass termination detection protocol* for a network of distributed processes logically organized into a ring. This is a popular termination detection protocol, see, e.g., [13].

The protocol is used for termination detection of a ring of processes. It can handle the case when processes may be reactivated after their local termination. To this end, it uses colored (i.e., black or white) tokens. Processes are also colored: a black color means global termination may have not occurred. Then, the algorithm works as follows:

- The root process P_0 becomes white when it has terminated and it generates a white token that is passed to P_1.
- The token is passed through the ring from one process P_i to the next when P_i has terminated. However, the color of the token may changed. If a process P_i passes a task to a process P_j with $j < i$, then it becomes a black process; otherwise it is a white process. A black process will pass on a black token, while a white process will pass on the token in its original color. After P_i has passed on a token, it becomes a white process.
- When P_0 receives a black token, it passes on a white token; if it receives a white token, all processes have terminated.

Suppose there are `m` processes, denoted `0,...,m-1`. Besides the input `m`, the program uses the spatial variables `id` : `sInt`, `c` : {`white`, `black`}, `active` : `sBool` and the temporal variables `tm`, `tid` : `tInt`, `msg` : `tIntSet`[ ]. (`sInt`, `sBool`, `tInt`, and `tIntSet` stands for spatial integers, spatial booleans, temporal integers, and temporal sets of integers, respectively.)

The program `P` is presented in Fig. 7. It is the diagonal composition `P` = `I` $ `Q` of an initialization program `I` and a core program `Q`. The diagonal composition ensures the communication of the last process with the first, as well as a correct continuation of each process execution from the former state. It is worthwhile to mention that the system here is closed. By a small change, it is possible to model an open version where processes may freely join or leave the ring.

The spatial variables `id`, `c`, `active` represent the process identity, its color, and its active/passive status. The temporal variables used in this program are: (i) `tm,tid` - temporal versions of `m,id`; (ii) `msg[ ]` - an array of sets, where `msg`[`k`] contains the `id` of the source processes of the pending messages sent to process `k`; (iii) `token.col` - an element of {`white,black`} representing the color of the token; and (iv) `token.pos` - the number of the process that has the token.

The program starts with the initialization of the network (program `I`) by activating all the processes (and setting the fields `id`, `c`, `active`). Initially, `msg`[`i`] = $\emptyset$, for all `0` $\leq$ `i` $<$ `m`, because no jobs were sent and the default color/position of the

[8] In this extension, except for simple and common programming conventions, we suppose to have an implementation of sets with their basic operations.

The program is P = I $ Q where:

```
 I = I1# for_s(tid=0;tid<tm;tid++){I2}#
I1 = module{listen nil}{read m}{
     tm=m; token.col=black; token.pos=0;
     }{speak tm,tid,msg[ ],token(col,pos)}{write nil}
I2 = module{listen tm,tid,msg[ ],token(col,pos)}{read nil}{
     id=tid; c=white; active=true; msg[id]=emptyset;
     }{speak tm,tid,msg[ ],token(col,pos)}{write id,c,active}
 Q = while_st(!(token.col==white && token.pos==0)){
        for_s(tid=0;tid<tm;tid++){R}}
 R = module{listen tm,tid,msg[ ],token(col,pos)}{read id,c,active}{
     if(msg[id]!=emptyset){ //take my jobs
        msg[id]=emptyset;
        active=true;}
     if(active){ //execute code, send jobs, update color
        delay(random_time);
        r=random(tm-1);
        for(i=0;i<r;i++){ k=random(tm-1);
          if(k!=id){msg[k]=msg[k]∪{id}};
          if(k<id){c=black};}
        active=random(true,false);}
     if(!active && token.pos==id){ //termination
        if(id==0)token.col=white;
        if(id!=0 && c==black){token.col=black;c=white};
        token.pos=token.pos+1[mod tm];}
     }{speak tm,tid,msg[ ],token(col,pos)}{write id,c,active}
```

Fig. 7. An AGAPIA v0.1 program for termination detection

token is black/0.

After the initialization part and until the first process receives a white token back, each process executes its code. If one process has the token and terminates, it passes the token to the next process (only the first process has the right to change the color of the token into white once it terminates).

When a process executes the code R, whether active or passive, it checks if new jobs were assigned to it; if the answer is positive, it collects its jobs from the jobs list and stays/becomes active. When it is active, it executes some code, sends new jobs to other processes, and randomly goes to an active or passive state. If it has the token, it keeps it until it reaches termination and afterward it passes it. A white process will pass the token with the same color as it was received and a black process will pass a black token (after passing the token, the process becomes white).

4 Types for AGAPIA v0.1 programs

In this section we present a typing system for AGAPIA v0.1 programs. The starting types are those used for the declared variables. The typing is naturally extended to expressions, simple while-programs, modules, and full AGAPIA v0.1 programs.

4.1 Interface types

In the representation of the types for interfaces we use two special separators "," and ";". On spatial interfaces, the role of "," is to separate the types of the (spatial) variables used within a specific process, while ";" is used to separate the types of the variables from different processes. On temporal interfaces, the first separator "," is used to separate the types of the temporal variables used within the same transaction, while the second separator ";" is a deliminator for the temporal variables which occur in different transactions. Finally, union of types, denoted by "∪", is allowed.

Simple spatial types

Simple spatial types are obtained with the following syntax:

$$SST ::= nil \mid sn \mid sb \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$$

where "," and "nil" respect the monoid laws[9] and "∪" is associative. They are intended to be used in a process.

An example is $((((sn)^*)^*, sb, (sn, sb, sn)^*)^*, (sb \cup sn))$. In a conventional representation, the type represents two variables $(\mathtt{x}, \mathtt{y})$, where `x` an array of type `struc1` and `y` is a boolean or an integer[10]. Next, `struc1` is a structure with three fields $(\mathtt{a}, \mathtt{b}, \mathtt{c})$, where `a` is an array of arrays of integers, `b` is a boolean, and `c` is an array of type `struc2`. Finally, `struc2` is a structure with three fields $(\mathtt{p}, \mathtt{q}, \mathtt{r})$, where `p` is an integer, `q` a boolean, and `r` an integer. Inserting such variables, one may write the type as

```
x:  struc1[], where
    struc1 = ( a:  Int[][],
               b:  Bool,
               c:  struc2[], where
                   struc2 = (p:Int, q:Bool, r:Int)
             ),
y:  Bool or Int
```

General spatial types

General spatial types specify system interfaces, i.e., the spatial type of a collection of processes running in different nodes. They are obtained with the following syntax:

[9] I.e., "," is associative, and has nil as a neutral element.

[10] This is a simple instance of a polymorphic type.

$$ST ::= nil \mid (SST) \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^*$$

where, as above, ";" and "nil" respect the monoid laws and "∪" is associative. [11]

An example is $((sn)^*)^*; nil; sb; ((sn)^*;)^*$. This represents the spatial type of a collection of processes (A, B, C, D), where A is a process using an array of arrays of integers, B is a process with no starting spatial data [12], C is a process using a boolean variable, and D is an array of processes, each process using an array of integers. Notice the difference between the first and the last subexpressions: In $((sn)^*)^*$ both iterates are in the same place, while in $((sn)^*;)^*$ the first iterate is in a certain place (within a process), while the second is spread on different places, hence generating an array of differently located processes. [13]

Temporal types

They are similarly handled (use space-time duality).

Reshaping types

One may use *transformed speakers and recorders* [14] (i.e., space-to-time and time-to-space converters mixed with identities [17]) to reshape the spatial interfaces. For instance, there is a natural morphism from $(sn;)^*$ to $(sn)^*$, which may be implemented as follows: (1) use generalized speakers to transform the spatial integers sn into temporal integers tn and to propagate them along the processes; then, (2) use a large recorder to transform the temporal integers received at the temporal output interface of the last process into an array of spatial integers on a unique process. Formally, the morphism is

$$TS_{\epsilon,a_1} \triangleright TS_{(a_1)^\vee,a_2} \triangleright TS_{(a_1,a_2)^\vee,a_3} \triangleright \ldots \triangleright TS_{(a_1,a_2,\ldots,a_{n-1})^\vee,a_n} \triangleright R_{(a_1,a_2,\ldots,a_n)^\vee}$$
$$: \langle (a_1; a_2; \ldots; a_n) | \ \rangle \to \langle (a_1, a_2, \ldots, a_n) | \ \rangle$$

where $TS_{a,b}$ is a generalized speaker of type $\langle a|b\rangle \to \langle a, b^\vee| \ \rangle$ and R_a is a recorder of type $\langle a| \ \rangle \to \langle \ |a^\vee\rangle$. [15]

Types vs. formal languages

Ultimately, the types specify complex structures of occurrences of primitive types. In a variable-free approach, two types may be considered equal if the formal languages represented by their expressions are equal. A simple example involving data structures from a single process is $(sn, (sn)^*) = ((sn)^*, sn)$. It corresponds to

[11] As usual, redundant parentheses are often omitted. E.g. the syntactically different expressions nil and (nil) or $(a; (b; c))$ and $a; b; c$ are identified.

[12] A typical example is the creation of a process or an object where the starting data come from their temporal interfaces (using a message, or a constructor).

[13] To be more consistent, we should perhaps include the delimiter "," also in the definition of SST using $(SST,)^*$. But the notation becomes a bit boring; e.g., the above type would become $((((sn,)^*,)^*, sb, (sn, sb, sn,)^*,)^*, (sb \cup sn))$. The case of ";" looks more natural as we are familiar with the use of ";" as a specification of the termination of a statement in usual imperative programming languages.

[14] They are presented in Subsec. 2.1, Fig. 2(e)(g).

[15] Here, $^\vee$ denotes the space-time duality operator, ',' is a generic notation for the monoidal operation on simple spatial or temporal interfaces, and $\triangleright$ denoted horizontal composition.

a list where the first (on left) and the last (on right) element is emphasized, respectively. In a conventional representation, they are represented as $(x : Int, y : Int[\,])$ and $(z : Int[\,], w : Int)$. They look different, but there is a matching function which may be used to connect these interfaces [16]. Such an approach is common in functional programming studies dedicated to abstract data type, see, for instance, [38]. One can argue on the soundness of the equality $(sn; (sn;)^*) = ((sn;)^*; sn)$ in our setting, as well. We do not use this implicit matching from "variable-free programming," hence the interface matching below will be more restrictive. For instance, the above identification is accepted provided one has specific constants to reshape these interfaces.

4.2 Typing expressions

The typing here is relatively easy, provided one starts with proper types for V in both, E and B.

Typing declarations

The syntax for type declarations is $x : ST$ or $x : TT$. In order to handle such complex data types, a mechanism for accessing their primitive integer or boolean components is needed. We use the following notation:

(k) for accessing the k-th element of an alternative choice (separated by "∪")

.k for accessing the k-th element of a structure (separated by ",")

.[k] for accessing the k-th element of an array (defined by $(...)^*$);

@k for accessing the k-th process/transaction (separated by ";")

@[k] for accessing the k-th process/transaction of an array of processes/transactions (defined by "$(...;)^*$")

With this convention, if w is a datum with the type presented in a previous paragraph, i.e., $((((sn)^*)^*, sb, (sn, sb, sn)^*)^*, (sb \cup sn))$, then a component corresponding to the 2nd sb, (i.e., the `q` component of the above conventional representation of this type) may be specified by $w.1.[i].3.[j].2$, for appropriate i, j. For another example, if $w : ((sn)^*)^*; nil; sb; ((sn \cup sb)^*;)^*$ the integer data corresponding to the last sn may be accessed by $w@3@[i].[j](1)$. Notice that nil does not increase the counter.

Typing spatial variables

For a spatial variable V, the type $\sigma(V)$ represents its type, paired with a flag in $\{\texttt{ok}, \texttt{war0}, \texttt{err}\}$ indicating ok, a warning, or an error resulting from an attempt to access its components. A min function is defined on the set $\{ok, war0, war1, err\}$, thought of as an ordered set, decreasing from left to right (it returns the minimum of two elements). The typing is presented in Fig. 8.

[16] For instance, if y, z have equal nonzero length it maps x in $z[0]$, then $y[0..len-1]$ in $z[1..len]$, and finally, $y[len]$ in w; if both are empty, it maps x in w; if the lengths are different, an error is raised.

$$\sigma(x : ST) = (\texttt{ok}, ST)$$

$$\sigma(x(k)) = \begin{cases} (\texttt{min}(\texttt{ok}, \beta), \alpha) & \text{if } \sigma(x) = (\beta, \gamma),\ \gamma \text{ is a union ``}\cup\text{'' type and } \alpha \\ & \text{is the type of its } k\text{-th component} \\ (\texttt{err}, \texttt{not defined}) & \text{otherwise} \end{cases}$$

$$\sigma(x.k) = \begin{cases} (\texttt{min}(\texttt{ok}, \beta), \alpha) & \text{if } \sigma(x) = (\beta, \gamma),\ \gamma \text{ is a product ``,'' type and} \\ & \alpha \text{ is the type of its } k\text{-th component} \\ (\texttt{err}, \texttt{not defined}) & \text{otherwise} \end{cases}$$

$$\sigma(x[k]) = \begin{cases} (\texttt{min}(\texttt{war0}, \beta), \alpha) & \text{if } \sigma(x) = (\beta, \gamma),\ \gamma \text{ is an iteration type ``*'' and} \\ & \alpha \text{ is the type of its components} \\ (\texttt{err}, \texttt{not defined}) & \text{otherwise} \end{cases}$$

$$\sigma(x@k) = \begin{cases} (\texttt{min}(\texttt{ok}, \beta), \alpha) & \text{if } \sigma(x) = (\beta, \gamma),\ \gamma \text{ is a product ``;'' type and} \\ & \alpha \text{ is the type of its } k\text{-th component} \\ (\texttt{err}, \texttt{not defined}) & \text{otherwise} \end{cases}$$

$$\sigma(x@[k]) = \begin{cases} (\texttt{min}(\texttt{war0}, \beta), \alpha) & \text{if } \sigma(x) = (\beta, \gamma),\ \gamma \text{ is an iteration type ``\_;*''} \\ & \text{and } \alpha \text{ is the type of its components} \\ (\texttt{err}, \texttt{not defined}) & \text{otherwise} \end{cases}$$

Fig. 8. Typing spatial variables

Typing temporal variables

This case is similar to the case of spatial variables.

Typing arithmetic expressions

For an arithmetic expression, we collect in a set the variables occurring in the expression, together with their types. The leafs of the expression should have an integer *sn* or *tn* type, otherwise the error type `err` is associated. The typing of the leafs is naturally extended to the full expression, collecting the types of the components and propagating their `ok, war0, err` status flags.

Typing boolean expressions

For boolean expressions the typing is similar: The starting variables should have a boolean *sb* or *tb* type, otherwise the error type `err` is associated. This starting typing is extended to full expressions as above.

4.3 Typing programs

The types of AGAPIA v0.1 programs may be specified using the following mapping from programs to types

$$\sigma : P \mapsto (st_{\sigma(P)}, \langle w_{\sigma(P)} | n_{\sigma(P)} \rangle \rightarrow \langle e_{\sigma(P)} | s_{\sigma(P)} \rangle)$$

where:

- The status flag st is an element of $\{\texttt{ok}, \texttt{war0}, \texttt{war1}, \texttt{err}\}$ specifying whether the program is: (1) well-typed; (2-3) partially well-typed with two levels of warning: a weak warning message `war0` and a stronger warning message `war1`; or (4) wrongly typed. [17] When $st = \texttt{err}$, the second component is meaningless.
- On each west, north, east, or south interface, the type $w_{\sigma(P)}$, $n_{\sigma(P)}$, $e_{\sigma(P)}$, or $s_{\sigma(P)}$ consists of a set of variables paired with their associated types.

  The comparison for the type matching on an interface proceeds along the following steps: (1) Check if the same set of variables is used; (2) For each variable the comparison returns: `ok` - they are equal and singleton; `war0` - they are equal and not singleton; `war1` - they are not equal, but have a nonempty intersection; `err` - they are not equal and have an empty intersection; (3) Finally, the overall status flag is the minimum of the status flags for each variable in the interface set.

4.3.1 Typing simple while programs and modules

The typing of simple while programs and modules is a simple extension of the classical typing used in sequential programs.

Simple while programs

These are simple, usual while programs. Once we have a typing for the basic blocks (expressions and assignment statements), their typing may be defined as in [36]. [18]

For instance, in the case of an assignment, the type of the variable on left is compared to the type of the expression on right. The status part $\{\texttt{ok}, \texttt{war0}, \texttt{war1}, \texttt{err}\}$ collects the information on the status part for the variable, the status part for the expression, and the comparison result regarding the matching of these two types.

Modules

For a module, take the type of the body program and export on the interfaces the variables occurring in the corresponding `listen/speak` and `read/write` statements.

4.3.2 Typing structured rv-programs

On programs, the typing morphism is inductively defined as follows.

[17] As said before, a *min* function is defined on the set $\{ok, war0, war1, err\}$, thought of as an ordered set, decreasing from left to right.

[18] Notice that a global scoping is used here, hence subtyping rules are to be used, as well.

Vertical composition

$\sigma(S1\%S2) = (st, \langle w_{\sigma(S1)}; w_{\sigma(S2)} | n_{\sigma(S1)} \rangle \rightarrow \langle e_{\sigma(S1)}; e_{\sigma(S2)} | s_{\sigma(S2)} \rangle)$, where

$$st = \begin{cases} min(\texttt{ok}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} = n_{\sigma(S2)} = \text{singleton} \\ min(\texttt{war0}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} = n_{\sigma(S2)} = \neg\text{singleton} \\ min(\texttt{war1}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } s_{\sigma(S1)} \cap n_{\sigma(S2)} \neq \emptyset \\ \texttt{err} & \text{if } s_{\sigma(S1)} \cap n_{\sigma(S2)} = \emptyset \end{cases}$$

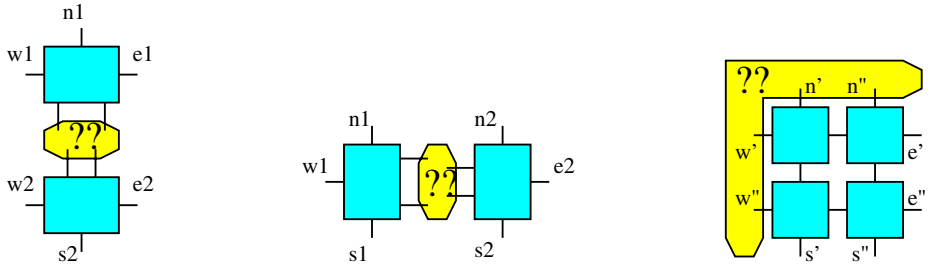


Fig. 9. Typing vertical/horizontal composition and "if" statements

Horizontal composition

$\sigma(S1\#S2) = (st, \langle w_{\sigma(S1)} | n_{\sigma(S1)}; n_{\sigma(S2)} \rangle \rightarrow \langle e_{\sigma(S2)} | s_{\sigma(S1)}; s_{\sigma(S2)} \rangle)$, where

$$st = \begin{cases} min(\texttt{ok}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } e_{\sigma(S1)} = w_{\sigma(S2)} = \text{singleton} \\ min(\texttt{war0}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } e_{\sigma(S1)} = w_{\sigma(S2)} = \neg\text{singleton} \\ min(\texttt{war1}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } e_{\sigma(S1)} \cap w_{\sigma(S2)} \neq \emptyset \\ \texttt{err} & \text{if } e_{\sigma(S1)} \cap w_{\sigma(S2)} = \emptyset \end{cases}$$

Diagonal composition

$\sigma(S1\$S2) = (st, \langle w_{\sigma(S1)} | n_{\sigma(S1)} \rangle \rightarrow \langle e_{\sigma(S2)} | s_{\sigma(S2)} \rangle)$, where denoting

$P1 := s_{\sigma(S1)} = n_{\sigma(S2)} = \text{singleton}$, $Q1 := e_{\sigma(S1)} = w_{\sigma(S2)} = \text{singleton}$,
$P2 := s_{\sigma(S1)} = n_{\sigma(S2)} = \neg\text{singleton}$, $Q2 := e_{\sigma(S1)} = w_{\sigma(S2)} = \neg\text{singleton}$,
$P3 := s_{\sigma(S1)} \cap n_{\sigma(S2)} \neq \emptyset$, $Q3 := e_{\sigma(S1)} \cap w_{\sigma(S2)} \neq \emptyset$,
$P4 := s_{\sigma(S1)} \cap n_{\sigma(S1)} = \emptyset$, $Q4 := e_{\sigma(S1)} \cap w_{\sigma(S2)} = \emptyset$

we have

$$st = \begin{cases} min(\texttt{ok}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } P1 \wedge Q1 \\ min(\texttt{war0}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } P2 \wedge (Q1 \vee Q2) \vee (P1 \vee P2) \wedge Q2 \\ min(\texttt{war1}, st_{\sigma(S1)}, st_{\sigma(S2)}) & \text{if } P3 \wedge (Q1 \vee Q2 \vee Q3) \vee (P1 \vee P2 \vee P3) \wedge Q3 \\ \texttt{err} & \text{if } P4 \vee Q4 \end{cases}$$

If

$\sigma(if(B)\{S1\}else\{S2\}) = (st, \langle w_{\sigma(S1)} \cup w_{\sigma(S2)} | n_{\sigma(S1)} \cup n_{\sigma(S1)} \rangle \rightarrow \langle e_{\sigma(S1)} \cup e_{\sigma(S2)} | s_{\sigma(S1)} \cup s_{\sigma(S2)} \rangle)$ where

$$st = \begin{cases} min(\texttt{ok}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{if } \sigma(B) \subseteq (w_{\sigma(S1)} \cup n_{\sigma(S1)}) \cap (w_{\sigma(S2)} \cup n_{\sigma(S2)}) = \text{singleton} \\ min(\texttt{war0}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{if } \sigma(B) \subseteq (w_{\sigma(S1)} \cup n_{\sigma(S1)}) \cap (w_{\sigma(S2)} \cup n_{\sigma(S2)}) = \neg\text{singleton} \\ min(\texttt{war1}, st_B, st_{\sigma(S1)}, st_{\sigma(S2)}) \\ \qquad \text{if } \sigma(B) \cap ((w_{\sigma(S1)} \cup n_{\sigma(S1)}) \cap (w_{\sigma(S2)} \cup n_{\sigma(S2)})) \neq \emptyset \\ \texttt{err} \\ \qquad \text{if } \sigma(B) \cap ((w_{\sigma(S1)} \cup n_{\sigma(S1)}) \cap (w_{\sigma(S2)} \cup n_{\sigma(S2)})) = \emptyset \end{cases}$$

Temporal while

$\sigma(while\_t(B)\{S\}) = (st, \langle (w_{\sigma(S)};)^* | n_{\sigma(S)} \cup s_{\sigma(S)} \rangle \rightarrow \langle (e_{\sigma(S)};)^* | n_{\sigma(S)} \cup s_{\sigma(S)} \rangle)$ where denoting

$P1 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \text{singleton}$, $Q1 := s_{\sigma(S)} = n_{\sigma(S)} = \text{singleton}$,
$P2 := \sigma_B \subseteq w_{\sigma(S)} \cup n_{\sigma(S)} = \neg\text{singleton}$, $Q2 := s_{\sigma(S)} = n_{\sigma(S)} = \neg\text{singleton}$,
$P3 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) \neq \emptyset$, $Q3 := s_{\sigma(S)} \cap n_{\sigma(S)} \neq \emptyset$,
$P4 := \sigma_B \cap (w_{\sigma(S)} \cup n_{\sigma(S)}) = \emptyset$, $Q4 := s_{\sigma(S)} \cap n_{\sigma(S)} = \emptyset$

we have

$$st = \begin{cases} min(\texttt{ok}, st_B, st_{\sigma(S)}) & \text{if } P1 \wedge Q1 \\ min(\texttt{war0}, st_B, st_{\sigma(S)}) & \text{if } P2 \wedge (Q1 \vee Q2) \vee (P1 \vee P2) \wedge Q2 \\ min(\texttt{war1}, st_B, st_{\sigma(S)}) & \text{if } P3 \wedge (Q1 \vee Q2 \vee Q3) \vee (P1 \vee P2 \vee P3) \wedge Q3 \\ \texttt{err} & \text{if } P4 \vee Q4 \end{cases}$$

Spatial while

The spatial version $\sigma(while\_s(B)\{S\})$ is similar to the temporal one.

Spatio-temporal while

The spatio-temporal while $\sigma(while\_st(B)\{S\})$ is similar to the temporal while, but slightly more complicate to write down as we have 3 pairs of interfaces to compare now: first, σ_B vs. $w_{\sigma(S)} \cup n_{\sigma(S)}$; then, $n_{\sigma(S)}$ vs. $s_{\sigma(S)}$; and, finally, $w_{\sigma(S)}$ vs. $e_{\sigma(S)}$.

4.4 Examples

Example 4.1 This example shows that the presence of regular-like expressions on interface type specifications is a natural consequences of the presence of "composition/if/while" statements in AGAPIA v0.1 programs. Consider the program

$P1 =$ `while_t(x>0)`$\{R\}$, where
R = `module{listen nil}{read x}`
`{new y:tn; y = x; x--;}{speak y}{write x}`

and its equivalent one-step unfolding

$P2 =$ `if(x>0)` $\{R$ `%` `while_t(x>0)`$\{R\}\}$ `else` `{nil}`

where x is a spatial integer of type sn and y a temporal integer of type tn. Except for the flag, the type of $P1$ is $\langle nil|sn\rangle \to \langle (tn;)^*|sn\rangle$, while the type of $P2$ is $\langle nil|sn\rangle \to \langle nil \cup (tn;(tn;)^*)|sn\rangle$. If the programs are to be identified, then $(tn;)^*$ and $nil \cup (tn;(tn;)^*)$ are to be valid interface specifications and equal. [19]

Consequently, we have to allow for union "$\cup$", composition ";", and star "$(..;)^*$" on temporal interfaces. A similar argument apply to spatial interfaces. Using space-to-time and time-to-space constants, one may reshape such an interface to have the operations in the same process or transaction (as in a previous example of reshaping interfaces, presented in Subsec. 4.1). To conclude, if one agrees to use "if" and the temporal, the spatial, and the spatio-temporal "composition" and "while" statements in programs, as well as reshaping of interfaces, than one has to allow for the interface types included in AGAPIA v0.1 programs, too.

Example 4.2 In this example we compute the type of the AGAPIA v0.1 termination detection program presented in Sec. 3.

To start with, let us use the notation: $a = (\texttt{tm}, \texttt{tid}, \texttt{msg}[\,], \texttt{token}), b = (\texttt{id}, \texttt{c}, \texttt{active}), c = (\texttt{m})$. The corresponding interface type declarations are not formally specified and we simply refer to the types via these variables. Then:

Init:
$\texttt{I1} \mapsto (\texttt{ok}, \langle nil|c\rangle \to \langle a|nil\rangle)$
$\texttt{I2} \mapsto (\texttt{ok}, \langle a|nil\rangle \to \langle a|b\rangle)$
$\texttt{for\_s}(\,)\{\texttt{I2}\} \mapsto (\texttt{ok}, \langle a|(nil;)^*\rangle \to \langle a|(b;)^*\rangle) = (\texttt{ok}, \langle a|nil\rangle \to \langle a|(b;)^*\rangle)$
$\texttt{I1\#for\_s}(\,)\{\texttt{I2}\} \mapsto (\texttt{ok}, \langle nil|c; nil\rangle \to \langle a|nil; (b;)^*\rangle) = (\texttt{ok}, \langle nil|c\rangle \to \langle a|(b;)^*\rangle)$

Repeat:
$\texttt{R} \mapsto (\texttt{ok}, \langle a|b\rangle \to \langle a|b\rangle)$
$\texttt{for\_s}(\,)\{\texttt{R}\} \mapsto (\texttt{ok}, \langle a|(b;)^*\rangle \to \langle a|(b;)^*\rangle)$
$\texttt{while\_st}(\,)\{\texttt{for\_s}(\,)\{\texttt{R}\}\} \mapsto (\texttt{war0}, \langle a|(b;)^*\rangle \to \langle a|(b;)^*\rangle)$

Full program:
$\texttt{P} \mapsto (\texttt{war0}, \langle nil|c\rangle \to \langle a|(b;)^*\rangle)$

While we know the program is correctly typed, the typing procedure rises a weak

[19] By a well-known regular expression identity $1 \cup aa^* = a^*$, the above types are equal, if formal language equivalence is used.

warring `war0` as it doesn't check whether the number `tm` of iterates in the inner `for` does not change from one loop of the `while_st` to the next.

Example 4.3 The typing procedure described in this section may reject correct programs, as it happens with many other similar type checkers. For instance, if a program has the structure

```
if (B) then { if (!B) then {X} else {Y} } else {Z}
```

then the `X` component is unreachable (provided there are no side-effects to change a variable value during a test). If a wrongly typed component is put on the place of `X`, then the program is rejected by the typing procedure, but it may be correct (provided the remaining part is correct).

To conclude, the typing procedure just presented has to be complemented with other program analysis techniques to get a friendly and powerful compiler.

5 Conclusions and future work

In this paper we have presented AGAPIA v0.1, a kernel programming language for interactive systems. A typing system for this language has been developed. A few lines for future work are presented below.

A first line of research is to develop the mathematics and the logics behind (structured) rv-systems. If one makes abstraction of both spatial and temporal data, one gets a mechanism equivalent to tile systems, existential monadic second order logics, etc. used for recognizable two-dimensional languages. There is a large literature dedicated to two-dimensional (or picture) languages (see, e.g., [20,21,27,31,32,33,34]) and it may be worthwhile to try to lift some results to the level of rv-systems. Particularly useful may be to find language preserving transformations which may be useful for developing efficient compilers for structured rv-programs.

In the last 20 years, a rich and successful algebraic approach to cyclic structures has been developed, see, e.g., [10,11,12,30,25,26,39,40,43], either for control or for reactive models. There are attempts to mix these two models, see, e.g., [23,24,42,41,45,44] or the last chapter of [43]. The model of rv-systems, presented in this paper, falls into this class. A difficult, but worthwhile, research topics is to extend such algebraic techniques to rv-programs.

Finally, a general topics is to develop an efficient and fully flagged compiler for AGAPIA programs. Our current approach is to translate structured rv-programs to rv-programs, then we use a running machine for rv-programs to get the program output. Currently, we have an automatic procedure for the translation (see [18]), but it is not fully implemented as we still look for the possible optimizations to improve the compiler.

References

[1] Abramsky, S. *Retracing some paths in process algebra.* In: "Proceedings of CONCUR'96, " 1-17. LNCS 1119, Springer, 1996.

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

[3] Bergstra, J.A., and J.W. Klop. *Process algebra for synchronous communication.* Information and Control **60**(1984), 109-137.

[4] Bergstra, J.A., C.A. Middelburg and G. Stefanescu. *Network algebra for asynchronous dataflow.* International Journal of Computer Mathematics **65**(1997), 57-88.

[5] Brock, J.D., and W.B. Ackermann. *Scenarios: A model of non-determinate computation.* In: "Proceedings of Formalization of Programming Concepts," 252-259. LNCS 107, Springer, 1981.

[6] Broy, M. *Nondeterministic dataflow programs: How to avoid the merge anomaly.* Science of Computer Programming **10**(1988), 65-85.

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

[8] Broy, M., and G. Stefanescu. *The algebra of stream processing functions.* Theoretical Computer Science **258**(2001), 99-129. (Technical Report TUM-I9620 and SFB-Bericht Nr. 342/11/96 A, Institute of Informatics, Technical University Munich, 1996.)

[9] Bruni, R. "Tile logic for synchronized rewriting of concurrent systems." PhD Thesis, Department of Computer Science, University of Pisa, 1999.

[10] Cazanescu, V.E., and G. Stefanescu. *Towards a new algebraic foundation of flowchart scheme theory.* Fundamenta Informaticae, **13**(1990), 171-210.

[11] Cazanescu, V.E., and G. Stefanescu. *A general result of abstract flowchart schemes with applications to the study of accessibility, reduction and minimization.* Theoretical Computer Science, **99**(1992), 1-63. (Fundamental Study.)

[12] Corradini, A., and F. Gadducci. *Rewriting on cyclic structures: equivalence between the operational and the categorical description.* Theoretical Informatics and Applications, **33**(4/5)(1999), 467-493.

[13] Dijkstra, E.W. *Shmuel Safra's version of termination detection.* EWD Manuscript 998, URL: http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF, January 1987.

[14] Dragoi, C., and G. Stefanescu. *Structured programming for interactive rv-systems.* Institute of Mathematics of the Romanian Academy, IMAR Preprint 9/2006, Bucharest 2006.

[15] Dragoi, C., and G. Stefanescu. *Towards a Hoare-like logic for structured rv-programs.* Institute of Mathematics of the Romanian Academy, IMAR Preprint 10/2006, Bucharest, 2006.

[16] Dragoi, C., and G. Stefanescu. *Implementation and verification of ring termination detection protocols using structured rv-programs.* Annals of University of Bucharest, Mathematics-Informatics Series, **55**(2006), 129-138.

[17] Dragoi, C., and G. Stefanescu. *Structured interactive programs with registers and voices and their verification.* Draft, Bucharest, January 2007.

[18] Dragoi, C., and G. Stefanescu. *On compiling structured interactive programs with registers and voices.* In: "Proc. SOFSEM 2008," 259-270. LNCS 4910, Springer, 2008.

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

[20] Giammarresi, D., and A. Restivo. *Two-dimensional languages.* In: "Handbook of formal languages. Vol. 3: Beyond words" (Rozenberg, G., and A. Salomaa, eds.), 215-265. Springer, 1997.

[21] Giammarresi, D., A. Restivo, S. Seibert, W. Thomas. *Monadic second order logic over rectangular pictures and recognizability by tiling systems.* Information and Computation, **125**(1996), 32-45.

[22] Goldin, D., S. Smolka and P. Wegner (Eds.). "Interactive computation: The new paradigm." Springer, 2006.

[23] Grosu, R., D. Lucanu, and G. Stefanescu. *Mixed relations as enriched semiringal categories.* Journal of Universal Computer Science, **6**(1)(2000), 112-129.

[24] Grosu, R., G. Stefanescu, and M. Broy. *Visual formalism revised.* In: "Proceeding of the CSD'98" (International Conference on Application of Concurrency to System Design, March 23-26, 1998, Fukushima, Japan), 41-51. IEEE Computer Society Press, 1998.

[25] Hasegawa, M. "Models of Sharing Graphs: A Categorical Semantics of let and letrec. PhD thesis, University of Edinburgh, Department of Computer Science, 1997.

[26] Hasegawa, M., M. Hofmann and G. Plotkin. *Finite dimensional vector spaces are complete for traced symmetric monoidal categories.* In: "Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday, " 540-559. LNCS 4800, Springer 2008.

[27] Inoue, K., and I. Takanami. *A survey of two-dimensional automata theory.* Information Sciences, **55**(1991), 99-121.

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

[29] Jonsson, B. *A fully abstract trace model for dataflow and asynchronous networks.* Distributed Computing, **7**(1994), 197-212.

[30] Joyal, A., R. Street and D. Verity. *Traced monoidal categories.* Proceedings of the Cambridge Philosophical Society, **119**(1996), 447-468.

[31] Lindgren, K., C. Moore and M. Nordahl. *Complexity of two-dimensional patterns.* Journal of Statistical Physics, **91**(1998), 909-951.

[32] Latteux, M., and D. Simplot. *Recognizable picture languages and domino tiling.* Theoretical Computer Science **178**(1997), 275-283.

[33] Latteux, M., and D. Simplot. *Context-sensitive string languages and recognizable picture languages.* Information and Computation, **138**(1997), 160-169.

[34] Matz, O. *Regular expressions and context-free grammars for picture languages.* In: "Proceedings STACS'97", LNCS 1200, 283-294. Springer, 1997.

[35] Peri, S. and N. Mittal. *On termination detection in an asynchronous system.* In: "Proc. 17th Int'l. Conf. on Parallel and Distributed Computing Systems (PDCS-2004)." ISCA Publications, 2004, 209-215.

[36] Pierce, B. "Types and programming languages." MIT Press, 2002.

[37] Popa, A., A. Sofronia and G. Stefanescu. *High-level structured interactive programs with registers and voices.* Journal of Universal Computer Science, **13**(11)(2007).

[38] Preoteasa, V. *A relation between unambiguous regular expressions and abstract data types.* Fundamenta Informaticae, **40**(1999), 53-77.

[39] Stefanescu, G. *Feedback theories (a calculus for isomorphism classes of flowchart schemes).* Revue Roumaine de Mathematiques Pures et Applique, **35**(1990), 73-79. (Early distributed as INCREST Preprint No.24/1986.)

[40] Stefanescu, G. Algebra of flownomials: Part I. Binary flownomials; basic theory. Technical Report TUM-I9437 and SFB-Bericht Nr. 342/16/94 A, Institute of Informatics, Technical University Munich, 1994.

[41] Stefanescu, G. *Reaction and control I. Mixing additive and multiplicative network algebras.* Logic Journal of the IGPL, **6**(2)(1998), 349-368.

[42] Stefanescu, G. *Axiomatizing mixed relations.* In: "Proceedings of 3rd RelMiCS Seminar, Hammamet, Tunisia," 177-186. University of Science, Technology and Medicine of Tunis, 1997.

[43] Stefanescu, G. "Network algebra." Springer, 2000.

[44] Stefanescu, G. *Kleene algebras of two dimensional words: A model for interactive systems.* Dagstuhl Seminar on "Applications of Kleene algebras", Seminar 01081, 19-23 February, 2001.

[45] Stefanescu, G. On space-time duality in computing: Imperative programming versus wave computation. In: "Proc. 4th 3rd RelMiCS Seminar," 197-202. Stefan Banach Mathematical Centre, Warsaw, 1998.

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

[47] Stefanescu, G. *Interactive systems with registers and voices.* Fundamenta Informaticae **73**(2006), 285-306. (Early draft, School of Computing, National University of Singapore, July 2004.)

[48] Stefanescu, G. *Towards a Floyd logic for interactive rv-systems.* In: "Proc. 2nd IEEE Conference on Intelligent Computer Communication and Processing" (Ed. A.I. Letia). Technical University of Cluj-Napoca, September 1-2, 2006, 169-178.

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

[50] Wegner, P. *Interactive foundations of computing.* Theoretical Computer Science **192**(1998), 315-351.