Journal of Universal Computer Science, vol. 13, no. 11 (2007), 1722-1754
submitted: 10/9/07, accepted: 19/11/07, appeared: 28/11/07 © J.UCS

High-level Structured Interactive Programs with Registers and Voices[1,2]

Alexandru Popa
(Department of Computer Science, University of Bucharest, Romania
Email: `alexpopa9@gmail.com`)

Alexandru Sofronia
(Department of Computer Science, University of Bucharest, Romania
Email: `alexandrusofronia@yahoo.com`)

Gheorghe Stefanescu
(Department of Computer Science, University of Bucharest, Romania
Email: `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 introduced by Stefanescu in 2004 using register machines and space-time duality, see [Stefanescu 2006, Stefanescu 2006b]. Later on, Dragoi and Stefanescu have introduced structured programming techniques for programming rv-systems and have presented a kernel programming language AGAPIA v0.1 for interactive computing systems, see [Dragoi and Stefanescu 2006a, Dragoi and Stefanescu 2006b].

AGAPIA v0.1 has a restricted format for program construction, using a "3-level" grammar for their definition: the procedure starts with simple while programs, then modules are defined, and finally AGAPIA v0.1 programs are obtained applying structured rv-programming statements on top of modules.

In the current paper the above restriction is completely removed. By an appropriate reshaping interface technique, general programs may be encapsulated into modules, allowing to reiterate the above "3-level" construction of programs, now starting with arbitrary AGAPIA programs, not with simple while programs. This way, high-level interactive programs are obtained. The extended version is called AGAPIA v0.2.

As a case study we consider a cluster of computers, each having a dynamic set of running processes. We present a protocol for the communication and termination detection in this system and implement the protocol in our AGAPIA v0.2 language. We also describe the operational semantics of the program using high-level scenarios, i.e., scenarios where, recursively, the cells may themselves contain scenarios, at a lower, refined level.

Key Words: interactive systems, programming languages, typing systems, rv-systems, registers and voices, distributed termination protocols, AGAPIA programming
Category: D.1, D.3

[1] C. S. Calude, G. Stefanescu, and M. Zimand (eds.). *Combinatorics and Related Areas. A Collection of Papers in Honour of the 65th Birthday of Ioan Tomescu.*

[2] The authors acknowledge partial support from the Romanian Ministry of Education and Research (CEEX Program, Project 2-CEx06-11-97 and PN-II Program, Project *GlobalComp - Models, semantics, logics and technologies for global computing*).

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, e.g., [Agha, 1986], [Broy and Olderog 2001], [Gadducci and Montanari 1999], [Jensen and Milner 2003], [Wadge and Ashcroft 1985], [Wegner 1998], to mention just a few references from a very rich literature. However, a simple, general, and unifying model for interactive computation to extend the classical, popular imperative programming paradigm is still to be find.

In an attempt to re-conciliate interactive and imperative computation styles, a model based on register machines and space-time duality has been recently proposed in [Stefanescu 2006]. Based on this model, a low level programming language, for writing interactive programs with registers and voices (rv-programs), has been presented [Stefanescu 2006]. 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).

In a couple of papers Dragoi and Stefanescu have developed structured programming techniques for rv-programs and a kernel programming languages for structured rv-systems, called AGAPIA v.01 (see the references).

AGAPIA is a kernel language for high-level programming of interactive systems. 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.

An example of AGAPIA v0.1 program is program `P` in Sec. 2 which presents an implementation for 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 the modules inherited from rv-programs. Such a module has both a spatial and a temporal interface. The spatial interface is specified using registers, while the 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 `listen/read` statements to access the temporal/spatial input of the module and `speak/write` statements to provide the temporal/spatial output of the module. The output is computed using a usual sequential program applied on the input of the module. An example of module is `R` in program `P`.

The structured programming operations for AGAPIA v0.1 programs extend the classical structured programming operations to this context. Composition has extensions to AGAPIA which exploit the multiple possibilities to compose blocks: (1) Temporal (vertical) composition via spatial interfaces; (2) Spatial (horizontal) composition via temporal interfaces; (3) Spatio-temporal (diagonal) composition, where both, the spatial and the temporal output data of a block become the spatial and 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*, *spatial*, and *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`.

If we use only "if" and the vertical composition and while, then we obtain usual structured programs. Using all types of compositions we obtain a kind of two dimensional network, whose nodes are modules that communicate vertically by spatial variables and horizontally by temporal variables.

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 the appendices).

In [Dragoi and Stefanescu 2007b], a typing system of AGAPIA v0.1 programs is presented . While it is the programmer duty to ensure the correctness of his/her program, the type checking procedure help him/her by checking the types of the programs and returning a four-level answer: (1) *ok* (the program is correct); (2) $war0$ (a running time miss-typing is possible); (3) $war1$ (there is a chance to have a well running program); and (4) *err* (each running using that piece of code fails).

The aim of the present paper is to broaden the class of structured rv-programs allowing to include arbitrary structured rv-programs into modules. This way *high-level structured rv-programs* are obtained. In these programs, recursively, the modules themselves may contain structured rv-programs. This extension is formally incorporated into AGAPIA language leading to a richer and more powerful version of AGAPIA, called AGAPIA v0.2.

As a case study we consider a cluster of computers, each having a dynamic set of running processes. We present a protocol for the communication and termination detection in this system and implement the protocol in our AGAPIA v0.2 language. We describe the operational semantics of the program using high-level scenarios, i.e., scenarios where, recursively, the cells may themselves contain scenarios, at a lower, refined level.

2 The AGAPIA v0.1 language

In this section we briefly present AGAPIA v0.1, a kernel programming language for interactive systems [Dragoi and Stefanescu 2007b].

2.1 Syntax

The syntax of the AGAPIA v0.1 programming language is presented in Fig. 1. It is given in a somehow theoretically oriented style. For practical use, the syntax has to be richer[3] and closer to common programming languages notations.

Interfaces

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

Expressions

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

Programs

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

Figure 1: The syntax of AGAPIA v0.1 programs

The types for spatial interfaces ST are built up starting with integer and boolean types sn, sb, applying $\cup$, ‘,’, $(\_)^*$ to get process interfaces, then applying $\cup$, ’;’, $(\_;)^*$ to get system interfaces.[4] Similarly, the temporal types TT are introduced. In practical programs, the description of data types will follow a more conventional approach: The star * defines an array, hence the usual [ ] notation will be used, with two formats `A[ ]` and `(A; )[ ]`. For union types, the “`or`” keyword will be used. Finally, the “,” and “;” product types are specified using the record notation, with items being separated by “,” and “;”, respectively.

Given a spatial or a temporal type X, the notations $X(k), X.k, X.[k], X@k$, and $X@[k]$ are used to refer to its components. For instance, in the case of spatial interfaces, they refer to: $X(k)$ - a component of a choice; $X.k$ - a component of

[3] E.g., to include a rich set of useful types and derived statements.

[4] They look slightly too complicate. An argument presented in Example 1 in [Dragoi and Stefanescu 2007b] shows that whenever `if` and the temporal, spatial, and spatio-temporal `composition` and `while` statements are legitimate programming language constructs, one has to allow for such types.

a tuple within a process; $X.[k]$ - a component of an iterated tuple within a process; $X@k$ - a component of a tuple of processes; and $X@[k]$ - a component of an iterated tuple of processes.

Next, expressions E, usual while programs W, modules M, and structured rv-programs P are introduced. This v0.1 version of AGAPIA has a restricted format for programs: module construction and structured programming statements are separated. More precisely, program construction starts with simple while programs, then modules are defined, and finally AGAPIA v0.1 programs are obtained applying structured rv-programming statements on top of module.[5] Three basic composition and iterated composition operations are used: a horizontal version # and $while\_s$, a vertical version % and $while\_v$, and a diagonal version $ and $while\_st$. See the included appendices for an extended presentation of the structured programming operations and their scenario-based semantics.

2.2 Dual-pass termination detection protocol

We describe a slightly extended[6] 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., [Dijkstra 1987]. The AGAPIA v0.1 program below is taken from [Dragoi and Stefanescu 2006c], where its correctness is proved, as well.

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.

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 be 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.

[5] Also, notice that the general "while" statement (presented in Appendix A) is not included in this version.

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

Suppose there are m processes, denoted 0,...,m-1. Besides the input m, the program uses the spatial variables id : sn, c : {white, black}, active : sb and the temporal variables tm, tid : tn, msg : tnSet[ ]. (sn, sb, ... represent spatial integers, booleans, etc.; tn, tb, ... denote the temporal versions.) We suppose the default initial value of variables of type sn, tn is 0, of those of type sb, tb is true, and this convention is naturally extended to complex interface types; e.g., the default initial value for sets is $\emptyset$.

The program P is presented in Fig. 2. 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 a process execution from its former state. It is worthwhile to mention that the system here is closed. It is possible to model an open version where processes may freely join or leave the ring. We describe such an extension briefly in an Appendix and in more details in our exemple on cluster communication in Sec. 4.

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

module I1{listen nil}{read m}{
   tm=m; token.col=black; token.pos=0;
}{speak tm,tid,msg[ ],token(col,pos)}{write nil}

module I2{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}

module R{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}
```

Figure 2: An AGAPIA v0.1 program for termination detection.

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 for 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 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 lists 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).

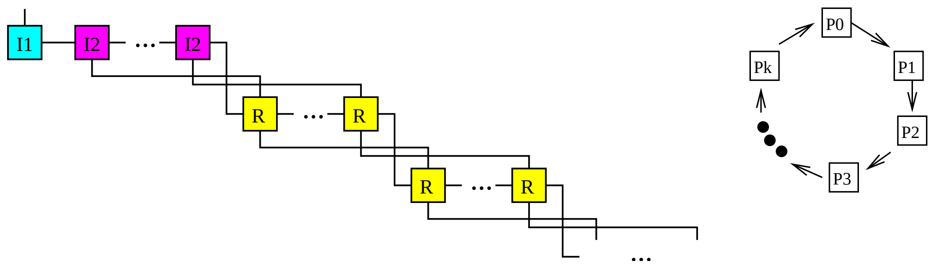


Figure 3: Typical scenarios for termination detection.

3 High-level structured rv-programs

The syntax of AGAPIA v0.1 in Fig. 1 is extended to allow for the construction of high-level structured rv-programs. The extension is done along the following steps.

First, modules with general ST, TT interfaces are introduced. They are an useful technical concept, but should be avoided as they lead to less well-structured programs.

Next, we particularly identify Scatter/Gather communication modules whose aim is to make a bridge between simple SST/STT and general ST/TT interfaces. They are general modules where either the west-east or the north-south direction has empty types, while the other has at one side a simple type and at the other a general type. These modules looks like communication networks passing data between its interfaces.

Finally, by an appropriate composition with Scatter/Gather modules a general program may be encapsulated into a high-level module with simple interface types. Then, one can reiterate the construction of complex programs using such high-level modules, obtaining *high-level structured interactive programs with registers and voices*.

3.1 General modules

In the first released version of AGAPIA (i.e. v0.1), the module structure was restricted to have simple spatial and temporal types SST, STT for its interfaces. This restriction may be dropped, hence modules with general ST, TT types for their interfaces may be used.

Two main complications come with this extension:[7]

1. First, a program encapsulated within such a module is "atomic", namely: (i) one has to wait for all the input data from its western and northern interfaces before starting the computation; and (ii) no output to its eastern or southern interfaces will be released before the end of the module computation.

2. Second, the cell composition in scenarios is more complicated than for simple modules as those used in AGAPIA v0.1 programs. Indeed, a cell corresponding to a general module is to be composed with potentially large scenarios which meet its interfaces, not with simple cells. Consequently, the grid structure of scenarios is lost, and an extension to some kind of acyclic hyper-graphs is to be used.

The first critics is common to many modularization techniques: often, a structured approach leads to more readable, better organized, but less efficient programs.

High-level structured programs with registers and voices (or AGAPIA v0.2 programs) are introduced here attempting to alleviate the second of the above drawbacks, still preserving a useful modular approach to program development.

[7] A small extra-price is the increased complication in accessing data from such general interfaces.

By a proper encapsulation of programs into modules with simple interface types, the operational semantics of programs may still be defined using rectangular grid-like scenarios. However, there is an important extension here: recursively, within each cell of a scenario, one may find other scenarios defined at a lower, refined level.

3.1.1 Syntax

The syntax for general modules is similar to that of AGAPIA v0.1 modules, with two changes:

1. General modules use general ST, TT types for its interfaces;
2. The body of a general module is still a usual while program, but using variables accessing the components of these general type interfaces.[8]

3.1.2 Comments, examples

In the example below we have a module with: (1) two incoming processes with their weights represented by spatial variables `X1,Y1`; and (2) two transactions with their priorities represented by variables `A1,B1`. The module sorts the inputs, i.e., the most powerful process is put on left, and the transaction with the strongest priority is put on top.

Example 1.

```
900  module Sort {listen A1:tn; B1:tn)}{read X1:sn; Y1:sn}
901  {
902    A2 = max(A1,B1);
903    B2 = min(A1,B1);
904    X2 = max(X1,Y1);
905    Y2 = min(X1,Y1);
906  }
907  {speak  A2:tn; B2:tn}{write  X2:sn; Y2:sn}
```

□

One can use different ways to write the module, particularly different ways to define data on lines 900,907. For instance, the western interface may be defined as an array {`listen A: (tn;)[2]`}, its components being `A@[1]` and `A@[2]`. Or, it may be written as {`listen tn;tn`} and their components may be accessed using the notation `west@1` and `west@2`, where "`west`" is the default name for this interface (if no other name is given); similarly, the default names `north`, `est`, and `south` are used for the other interfaces.

[8] In particular, the `@k` and `@[k]` notations are used to access the variables in different ST, TT components.

3.2 Encapsulating programs in modules with simple interfaces

To encapsulate a program into a module, we use particular general modules, called Scatter and Gather. In these modules, either the west-east or the north-south direction has empty types, while the other has at one side a simple type and at the other a general type. Moreover, no computation is performed into these modules, namely each atomic sn, sb variable in the output interface inherits the value of an appropriate variable in the input interface.[9]

3.2.1 Syntax

Interfaces

$ScatterS$ ::= **module** $module\_name\{listen\ nil\}\{read\ x : SST\}$
$\{\ body;\ \}\{speak\ nil\}\{write\ x : ST\}$
where $body$ is a (while-type) program using only $x := y$ assignments with x variable from the southern interface and x variable from the northern interface;
— it is used to "scatter" data from the spatial interface of a process (a simple spatial interface) to the spatial interface of a collection of processes (a general spatial interface)

$ScatterT$ ::= **module** $module\_name\{listen\ x : STT\}\{read\ nil\}$
$\{\ body;\ \}\{speak\ x : TT\}\{write\ nil\}$
— similar for temporal interfaces

$GatherS$ ::= **module** $module\_name\{listen\ nil\}\{read\ x : ST\}$
$\{\ body;\ \}\{speak\ nil\}\{write\ x : SST\}$
— it is used to "gather" data from the spatial interface of a collection of processes to the spatial interface of a single process

$GatherT$ ::= **module** $module\_name\{listen\ x : TT\}\{read\ nil\}$
$\{\ body;\ \}\{speak\ x : STT\}\{write\ nil\}$
— similar for temporal interfaces

3.2.2 Comments, examples

The example below describes a particular scatter module used in our cluster communication protocol in Sec. 4. We have an array of processes `P[ ]`. Process `P[0]` is a "root" process and the relevant information for its root activity is represented in a structure `stemp`. The states of these processes are recorded in an array `Pst[]`. All this information (`stemp,Pst[]`) is hold by a unique, master process. The role of this Scatter module is to scatter this information, sending to

[9] It's somehow similar to a connection network, where data are passed between connection ports, only (no real computation takes place into such a network).

each process the state for starting the computation and, additionally, to process `P[0]` the information for its root activity.

Example 2.

```
75  module Scatter {listen nil}{read stemp, Pst[]}
76  {
77    south@1.stemp = north.stemp;
78    south@1.Pst = north.Pst[0];
79    for(i=1; i<length(Pst[]); i++)
80      south@2@[i-1] = north.Pst[i];
81  }
82  {speak nil}{write (stemp,Pst); ((Pst;)[])}
```

□

In this example we use the default `north/south` name for the corresponding interfaces and the specified mechanism for accessing its components. E.g., in line 80, `south@2@[i-1]`, first by `@2` it refers to the 2nd process in the south interface (i.e., `(Pst;)[]`), then by `@[i-1]` to the (`i-1`)-th process in this array of processes - hence, globally, this is the `i`-th process in the south interface.

3.3 Functions

The starting point of the discussion on functions is provided by the following comments on the module instantiation. A module may be used in two, slightly different, ways.

1. A first, default use, is to simply insert the module name into a specific place of a program. In such a case, the temporal or spatial input data are implicitly taken from the context. This is the way programs and the typing procedure was developed in the previous version of AGAPIA.

2. A second, expanded use, may be to use the module name and actual expressions or parameters for its temporal and spatial input data. The syntax may be:

   `module_name{listen EAB : TT}{read EAB : ST}`

   where `EAB` denotes a *generalized expression* - such expressions are obtained by extending expressions from arithmetic or boolean types to general spatial or temporal types in ST, TT. While this clearly is a powerful and useful generalization, we do not present the details of the extension here.[10]

[10] Getting a right definition for expressions using and returning values of complex ST, TT types may require a detailed and careful presentation.

This observation opens the way to introduce functions and general assignments in AGAPIA. Actually, functions may be seen as particular modules. Indeed, with the interpretation described in the 2nd item above, if a module is exporting either a spatial or a temporal data type, but not both, then it may be thought of as a *generalized function* and it may be used in *generalized expressions*. As a byproduct, we get a way to use *generalized assignments* in program definition. Generalized assignments are `X = Exp`, where `X` is a variable with a general interface type and `Exp` is a corresponding generalized expression. Notice that assignment statements are not included in the program definition of AGAPIA v0.1 - they only appear in the while programs used in module definition and they are usual C-like assignment statements.

3.3.1 Comments, examples

The example below describes a gather module, which is a dual version of the previous scatter example. It is taken from the communication protocol in Sec. 4, as well.

Example 3.

```
84  module Gather {listen nil}{read (stemp,Pst); ((Pst;)[])}
85  {
86    south.stemp = north@1.stemp;
87    south.Pst[0] = north@1.Pst;
88    for(i=0; i<length(north@2); i++)
89      south.Pst[i+1] = north@2.Pst@[i];
90  }
91  {speak nil}{write stemp, Pst[]}
```

□

This module may be turned into a function, changing in line 84 the keyword “`module`” to “`function`”. The return type may be included in the definition to be closer to usual practice, e.g., line 84 may be rewritten as:

```
84 Type function GatherX ...
```

with the return type `Type = (stemp, Pst[])` quite complex, but still a simple spatial type in SST. Nevertheless, this is redundant as the type of the function is clearly specified by the module interfaces.

3.4 An extension of Agapia programs

Agapia v0.2 extends the module definition from Agapia v0.1, to include a new alternative, using the Scatter and Gather general modules defined in sec. 3.2, to encapsulate general programs into modules with simple interfaces. The full syntax is presented in Appendix 9.

3.4.1 Syntax

The syntax for programs is extended using the following new definition of modules.

Programs

$$
\begin{aligned}
M ::= \;& \textbf{module}\; module\_name\{listen\ x : STT\}\{read\ x : SST\}\\
& \quad\{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}\\
\mid\;& \textbf{module}\; module\_name\{listen\ x : STT\}\{read\ x : SST\}\\
& \quad\{\ ScatterT \# (ScatterS \,\%\ P\ \%\, GatherS) \# GatherT\ \}\\
& \quad\{speak\ x : STT\}\{write\ x : SST\}
\end{aligned}
$$

Recall that P in the 2nd part is the metavariable used to denote programs, hence the definitions for modules and programs are mutually recursive.[11]

3.4.2 Comments

The comment here is on the expressive power of the programs described with this extended syntax. The class of computations/scenarios described by well-formed programs is not actually extended by this syntax. Indeed, the key step of the extension is the second item in the module definition. The body of the module definition is a previously defined program. Then, scatter and gather modules are constants which may be defined using the basic constants and operations of AGAPIA v0.1. Finally, the `listen/read` and `speak/write` parts are superfluous, provided the program is well-typed. To conclude, a high-level program generated by the above syntax is equivalent to a flat program described using only the AGAPIA v0.1 syntax.

There is nothing unexpected here. Indeed, this is similar to the known fact that functions or procedures do not add to the expressive power of flat while or flowchart programs. However, functions and procedures provide a shorter and more readable presentation of the algorithms, and, more important, they are the stepping stone towards powerful modularization mechanisms, including those used in current OO-programming languages. We hope, the present extension to high-level interactive programs may be a bridge to clarify the connections between interactive (AGAPIA-like) programming and OO-languages.

3.4.3 Examples

The example below is a simplified version of a key module used in the communication protocol in Sec. 4.

Example 4.

[11] See the full syntax described in Fig. 8.

```
47  module R {listen Temp}{read Sp}
48  {
49    Scatter
50  %
51    RootInit1# for_s(ti=1; ti<tm; ti++) {Id1}#
52  %
53    (
54      (RootInit2# for_s(ti=1; ti<tm; ti++) {Id2}#)$
55      for_st(step = 0; C1
56                           && C2; step++ )
57        {
..          Act
65        }$
66      RootEnd# for_s(ti=1; ti<tm; ti++) {Id3}#
67    )
68  %
69    Gather
70  }
71  {speak Temp}{write Sp}
```

□

The program between lines 51–67, denoted by `W`, is a flat AGAPIA v0.1 program with simple temporal interface in STT, but with general spatial interfaces in $ST \setminus SST$. Using appropriate scatter and gather modules (lines 49,69), the program `W` is encapsulated into the module `R` with simple temporal and spatial interfaces. Note that the Scatter and Gather modules in lines 47,67 are spatial, and that temporal scatter and gather modules are omitted since W already has simple temporal interfaces. The scenarios of the program inside this module are described in the bottom part of Fig. 4.

The scenarios of the full, high-level program for cluster communication is presented in the top part of Fig. 4. Each `R` cell in this figure has inside scenarios at a lower level described in the bottom "Ring" part of the figure. A detailed presentation of the scenarios is included in figures Fig. 4,5. Notice that the diagonal composition in Fig. 4,5 is somewhat different from that described in Fig. 7(c). The change to a non-rectangular format of the cells is merely graphical (including some crossing) to simplify the number of graphical cross-overs between spatial and temporal interfaces in a diagonal composition.

4 Case study: Communication in a cluster of dynamic processes

In this example we consider the communication structure within a cluster of computers, each node having a set of running processes. The set of processes in each node is dynamic allowing new processes to join the set and old processes to leave it. The protocol considers the nodes and the processes to be logically organized into rings. We get a kind of "ring of rings" structure, where the outer ring is fixed, while the inner ones are dynamic. The protocol and its AGAPIA

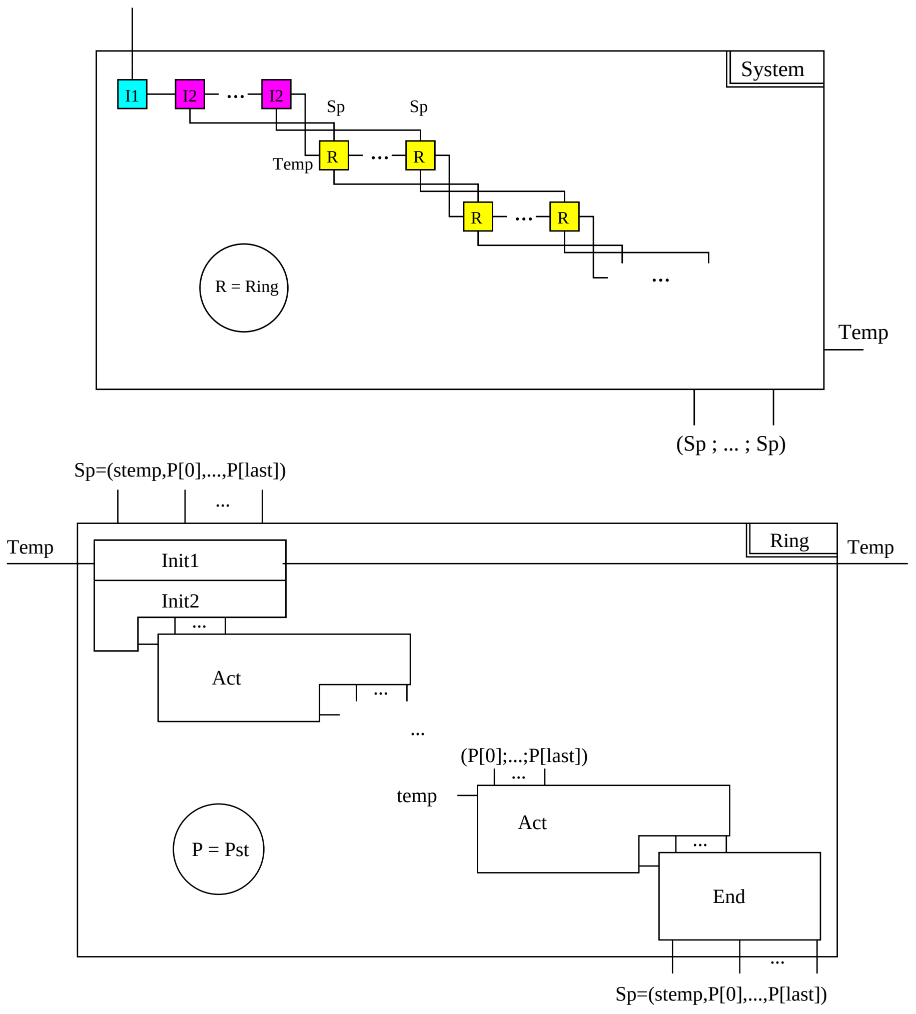


Figure 4: Scenarios for cluster termination protocol - I.

v0.2 implementation describe the communication within this network of dynamic processes, paying more attention to the termination detection by extending a classical dual-pass ring termination detection protocol.

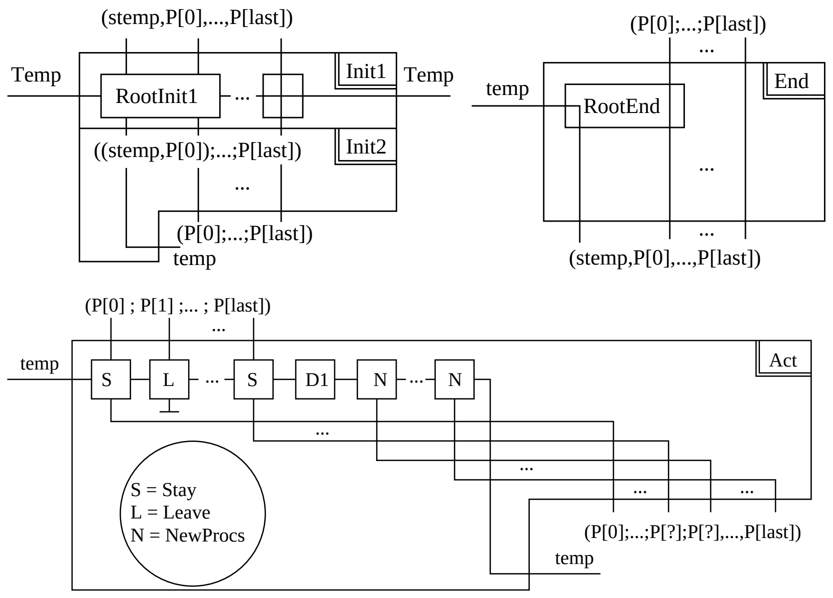


Figure 5: Scenarios for cluster termination protocol - II.

4.1 Termination in a cluster of sets of dynamic processes

In a previous article [Dragoi and Stefanescu 2007b][12] an AGAPIA v0.1 program that implements a dual pass termination detection protocol was described. Our purpose here is to describe an AGAPIA v0.2 program that implements an extended version of that protocol.

The architecture is similar with the previous one, but in the ring, each process is replaced by a ring of processes (i.e. we have “a ring of rings”). The number of rings is known from the beginning and cannot be changed, but in each of those inner rings processes may freely enter or leave. If it looks too complicated just imagine a network of computers, and on each computer some running processes.

We have two types of tokens: a small token in each ring (we will call it simply “token”) and a big token which is passed from ring to ring. In each ring things act in the same manner as in the first version of the protocol. There is also a process (called “root process”) that cannot leave the ring. A bunch of new processes will enter the ring after the last process, and individually each process decides to leave the ring or not. If a process decides to leave the ring it passes the token to the next process in the ring and then exits.

[12] A brief presentation is included in Sec. 2.

The big-token and the rings may also be black or white with the same meaning. A process from one ring may send a messages to processes from other rings.

4.1.1 The algorithm

The algorithm works as follows:

– The ring R_0 becomes white when all its processes have finished and it generates a white big-token that is passed to R_1. The token from the ring R_1 becomes black.

– The big-token is passed from one ring R_i to the next one when all its processes have finished. If a process from a ring R_i passes a task to a process from a ring R_j with $j < i$, then the ring becomes black; otherwise it is a white ring. A white ring will pass on a black big-token, while a white ring will pass on a big-token in its original color. After R_i has passed on a big-token, it becomes a white ring. When R_i passes a big-token it changes in black the color of the token from the next ring.

– When R_0 receives a black big-token, it passes on a white big-token; if it receives a white big-token, all processes have terminated.

– In a ring things act almost in the same manner. The difference is that new processes may enter or leave the ring. The new processes are inserted at the end. When a process leaves the ring it first passes the token to the next process in the ring and then exits.

4.1.2 Implementation

We have three variables to store messages. The first one is `msg_int[]` which is an array of sets; `msg_int[k]` contains pairs `(ring, pid)`, which represents the ring and the id of the processes that sent a message to process k. This is similar to `msg[]` from the previous example.

The second one is `msg_out`, i.e., structures `(exp_ring,exp_pid,dest_ring, dest_pid)` which contains all the messages that should leave the ring. Here, `exp_ring` and `exp_pid` represent the ring-id address of the sending process and `dest_ring` and `dest_pid` represent the ring-id address of the destination process.

The third variable is `msg_ext`. This is a set of structures similar to `msg_out`. The difference is that `msg_ext` contains messages from all the rings. The first two variables `msg_int[], msg_out` are local (they are visible inside of a ring), while `msg_ext` is global and it passes through all rings.

Communication with messages works in the following way. Each process has a set of received messages (kept in `msg_int[id]`, where id is the process' id).

When a process wants to send a message there are two cases: (i) the process sends a message inside his ring so it writes the message directly in `msg_int[]` (just like in the dual-pass termination detection protocol); (ii) the process sends a message outside his ring so, in this case, it writes the message in `msg_out`.

The process P_0 (also called a "root process") updates `msg_ext` and `msg_out`. Things act like in a postal system. It checks all the messages from the outside (i.e., from `msg_ext`) and if a message is for the local ring, it is moved into `msg_int`. Then `msg_out` is moved into `msg_ext`. After it finishes this job, P_0 sends `msg_ext` to the next ring.

The main program has two parts: an initialization part (line 17) and a computation part (lines 19 - 22). These two parts are composed diagonally, thus ensuring proper communication of the last ring with the first one, as it may be seen in the top part of Fig. 4. The initialization part is done by modules I1 and I2. I1 initializes the whole system, while I2 initializes each ring. Suppose there are `trings` rings, denoted `0,...,trings-1`. This variable is received by I1 through its spatial interface. After the initialization part, until the first ring receives a white big-token each ring executes his code.

Module R handles the behavior of a ring. It begins and ends with two special modules `Scatter` and `Gather` described previously in this article, which were introduced to match the interfaces. After `Scatter` follows another initialization part, corresponding to `Init1` and `Init2` from the bottom part of Fig. 4.

The code from `RootInit1` and `RootInit2` is executed by process P_0. In `RootInit1` the message lists are updated (the update of the message lists works according to the mechanism described above in this section). After that, P_0 decides if it should pass the big-token or not: if it has received a white small token, it passes the big-token to the next ring, otherwise the big-token stays. `RootInit2` only decides (randomly) how much work the processes from the ring have to do (i.e., it sets a maximal number of steps for the diagonal composition). Modules `Id1`, `Id2`, `Id3` have nothing to do, being used just for matching the interfaces.

When the initialization is done, the processes may begin their jobs. Processes work until they finish (line 55), or until the maximum number of steps (decided by `RootInit2`) is exceeded (line 56). Now, each process, except P_0, may leave the ring. If it decides to do so, it executes the `Leave` code, otherwise it executes the `Stay` code.

If it chooses to stay in the ring, it first checks to see if it has received messages from other processes, and, if it has, becomes active (if not already). Next, if it is an active process, it does something, sends messages to other processes (which may not be in its ring) and decides if he has more work to do (i.e., to stay active or not). In the end if it has finished his job (i.e., it is a passive process), it passes the token to the next process.

Finally, new processes may enter the ring (line 64). Module `D1` decides how many processes will enter, and `NewProcs` creates the processes. The id of a process is assigned using a variable `counter`, which has initial value 1 and increases each time a new process starts.

4.1.3 A short overview of the code: syntax, variables, functions ...

Variables are organized into several groups (these are just some notations to make the code easier to read).

4.1.3.1 Temporal groups

- `temp`:
  (i) `ti` - just a counter, it doesn't have a special role;
  (ii) `tm` - number of processes from a ring;
  (iii) `msg_int[]` - an array of sets, where `msg_int[k]` contains pairs `(ring, pid)`, which represents the ring and the id of the processes that sent a message to process `k`;
  (iv) `msg_out` - structures `(exp_ring, exp_pid, dest_ring, dest_pid)` which contains all the messages that should leave the ring;
  (v) `token.col` - an element of {`white,black`} representing the color of the token;
  (vi) `token.pos` - the number of the process that has the token;
  (vii) `procs[]` - an array of sets, where `procs[k]` contains the id of the processes that exist in ring `k`;
  (viii) `counter` - a variable used to assign an id to a newly created process;
  (ix) `rid` - ring id;
  (x) `rc` - an element of {`white,black`} representing the color of the ring.
- `Temp`:
  (i) `trings` - number of rings (rings are numbered from `0` to `trings - 1`);
  (ii) `trid` - temporal ring id;
  (iii) `procs[]` - the same as in `temp`;
  (iv) `msg_ext` - a set of structures similar to `msg_out` that contains messages from all the rings;
  (v) `bigtoken.col` - an element of {`white,black`} representing the color of the big-token;
  (vi) `bigtoken.pos` - the number of the process that has the token.

4.1.3.2 Spatial groups

- `stemp`: spatial version of `temp`;

– Pst: (i) pid - the id of the process; (ii) c - an element of {white,black} representing the color of a process; (iii) active - a boolean variable set true iff the process is active.

– Sp = (stemp,Pst[]) - it keeps the relevant information on a ring as a spatial data of the root process, including the information on process states and a record of the current temporal communication interface.

4.1.3.3 Other notations

There are several notations in the code, that will be explained further.

– length() function was used for returning the length of an array (for example in line 79), or cardinal of a set (line 152), depending on the context.

– random() function was also used several times. If it is used with no arguments it returns a random natural number. If it is used with an integer argument it returns a natural number between 0 and the argument. We also use this function to return a random element from a set (lines 60, 186).

– Union operator is used for the union of two sets. Of course, the implementation of this operator depends of how sets are represented (for example, sets may be thought as linked lists).

– delay() function is used in line 181, to simulate that the process has something to do.

– next() is used to pass the token to the next process in the ring (lines 205 and 212). The list of processes is circular, so the next process after the last one is P_0.

– Module I2 writes (stemp,Pst[0]) and it may seem that its interface doesn't directly match with the one of module R. But (stemp,Pst[0]) is the same with (stemp,Pst[]), because at the initialization point we have only one process P_0 (the root process), and (stemp,Pst[]) is denoted Sp.

4.2 The full code of the termination protocol

```
 1  // Notations:
 2
 3  temp = // temporary ring data passed between processes in a ring
 4          (ti, tm, msg_int[], msg_out, token(col,pos),
 5             procs[], counter, rid, rc)
 6  Temp = //temporary system data passed between rings
 7           (trings, trid, procs[], msg_ext, bigtoken(col,ring))
 8  stemp = (sti, stm, smsg_int[], smsg_out, stoken(col,pos),
 9             sprocs[], scounter, srid, src)
10  Pst = (pid,c,active) // the type of process state is (pid,c,active)
11  Sp = (stemp,Pst[]) // record states and temporal interface
```

```

// Main program and modules

main {listen nil}{read rings}
{
  I1# for_s(trid = 0; trid < trings; trid++) {I2}#
$
  while_st(!(bigtoken.col == white && bigtoken.ring == 0))
  {
    for_s(trid=0; trid<trings; trid++){R}
  }
}
{speak Temp}{write (Sp;)[]}

module I1 {listen nil}{read rings}
{
  trings  = rings; bigtoken.col = black; bigtoken.ring = 0;
  trid = 0;
  for(i=0; i < rings; i++)
    procs[i] = {0};
  msg_ext = emptyset;
}
{speak Temp}{write nil}

module I2 {listen Temp}{read nil}
{
  srid = trid; rc = white;
  stm = sti = 0; scounter = 1;
  stoken.col = black; stoken.pos = 0;
  Pst[0].pid=0; Pst[0].c= white; Pst[0].active = true;
  src = white;
  smsg_out = emptyset;
}
{speak Temp}{write (stemp,Pst[0])}

module R {listen Temp}{read Sp}
{
  Scatter
%
  RootInit1# for_s(ti=1; ti<tm; ti++) {Id1}#
%
  (
    (RootInit2# for_s(ti=1; ti<tm; ti++) {Id2}#)$
    for_st(step = 0; !(token.col == white && token.pos != 0)
                         && step < maxsteps; step++ )
      {
        for_s(ti=0; ti<tm; ti++)
        {
          if(pid == 0 || random({leave, stay}) == stay)
            Stay   // the process stay in the ring
          else
            Leave  // the process will leave the ring
        }# D1# for_s(ti=told; ti<tm; ti++){NewProcs}#
      }$
    RootEnd# for_s(ti=1; ti<tm; ti++) {Id3}#
  )
%
  Gather
}
{speak Temp}{write Sp}

// Scatter and Gather modules
```

```

module Scatter {listen nil}{read stemp, Pst[]}
{
  south@1.stemp = north.stemp;
  south@1.Pst = north.Pst[0];
  for(i=1; i<length(Pst[]); i++)
    south@2@[i-1] = north.Pst[i];
}
{speak nil}{write (stemp,Pst); ((Pst;)[])}

module Gather {listen nil}{read (stemp,Pst); ((Pst;)[])}
{
  south.stemp = north@1.stemp;
  south.Pst[0] = north@1.Pst;
  for(i=0; i<length(north@2); i++)
    south.Pst[i+1] = north@2.Pst@[i];
}
{speak nil}{write stemp, Pst[]}

module RootInit1 {listen Temp}{read stemp, Pst}
{
  //update process list for this ring
  Temp.procs[trid]= stemp.sprocs[trid];

  //update process list from other rings
  stemp.sprocs=Temp.procs; //vector assignment

  // get the messages for this rings and discard them from msg_ext
  for_each (exp_ring,exp_pid,dest_ring,dest_pid) from msg_ext
  {
    if(dest_ring == trid) // a message is for a process in this ring
    {
      if(dest_pid In sprocs[trid])
  smsg_int[dest_pid] = smsg_int[dest_pid] Union {(exp_ring,exp_pid)};
      msg_ext = msg_ext Minus {(exp_ring,exp_pid)};
    }
  }

  // put the messages sent to other rings
  msg_ext = msg_ext Union smsg_out;

  // keep big-token, if necessary
  if(bigtoken.ring == trid)
  {
    if(stoken.col == white && stoken.pos == 0)
    {
      if (trid == 0)
        bigtoken.col = white;
      if (trid != 0 && src == black)
      {
        bigtoken.col = black;
        src = white;
      }
      bigtoken.ring = bigtoken.ring + 1 [mod trings];
    }
  }
  ti = sti;
  tm = stm;
}
{speak Temp, ti, tm}{write stemp, Pst}

module RootInit2 {listen nil}{read stemp, Pst}
```

```
{
  temp = stemp; // vector assignment
  // choose a random number of communication rounds in the ring
  step = 0;
  maxsteps = random () + 1;
}
{speak temp, step, maxsteps}{write Pst}

module RootEnd {listen temp}{read Pst}
{
  stemp = temp; // vector assignment
}
{speak nil}{write stemp, Pst}

module D1 {listen temp}{read nil}
{
  told = length(procs[trid]);  // the number of current processes
  tnew = random();  // choose how many new proccesses to add
  tm = told + tnew;  // update tm
}
{speak temp, told}{write nil}

module Id1 {listen Temp}{read Pst}{ null; }{speak Temp}{write Pst}
module Id2 {listen temp}{read Pst}{ null; }{speak temp}{write Pst}
module Id3 {listen  nil}{read Pst}{ null; }{speak  nil}{write Pst}

module NewProcs {listen temp}{read Pst}
{
  pid = counter;  // pick the next free id from the list
  counter ++;  // increase the counter
  procs[trid] = procs[trid] Union {pid};
  c = white;
  active = true;
}
{speak temp}{write Pst}

module Stay {listen temp}{read Pst}
{
  if(msg_int[pid] != emptyset)  // take my jobs
  {
    msg_int[pid] = emptyset;
    active = true;
  }
  if(active)
  {
    delay(random_time);  // execute
    r = random(tm - 1);  // choose how many messages to send
    for(i=0; i<r; i++)
    {
      q = random(trings - 1); // choose the ring
      w = random in procs[q]; // choose the process
      if(q != rid) // for a process in another ring
        msg_out = msg_out Union {(rid,pid,q,w)};
      elseif (w != pid) // for a process in the same ring
        msg_int[w] = msg_int[w] Union {(q,w)};
      if(q < rid) // color the ring in black
        rc = black;
      if(q == rid && w < pid) // color the process in black
        c = black;
    }
    active = random(true,false);
  }
  if(!active && token.pos == pid) // termination
  {
    if(pid == 0) token.col = white;
    if(pid != 0 && c == black)
    {
      token.col = black; c = white;
    }
    token.pos = next(procs[rid],token.pos); // pass the token
  }
}
{speak temp}{write Pst}

module Leave {listen temp}{read Pst}
{
  token.pos = next(procs[rid],token.pos);  // pass the token
  procs[rid] = procs[rid] - pid;  // update process list
}
{speak temp}{write nil}
```

5 Conclusions and future work

In this paper we have introduced high-level structured interactive programs with registers and voices. They are incorporated into the new version v0.2 of AGAPIA programming language. As a case study we have developed an implementation for an intriguing problem related to the communication and termination detection in a cluster of dynamic processes. Many problems are still open, both, for the flat and for the high-level structured rv-programs.

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 are many interesting and deep results dealing with two-dimensional (or picture) languages and the lifting of some of them to the level of rv-system may be worthwhile attempt. Particularly useful may be to find language preserving transformations which may be useful for developing efficient compilers for structured rv-systems.

A more practical topics is to develop efficient and fully flagged compilers for AGAPIA-like programs. Our current approach uses the following route: we translate structured rv-programs to rv-programs, then we use the running machine for rv-programs to get the program output. Currently, we have an automatic procedure for the translation, but it is not fully implemented as we still look for optimizations to improve the compiler.

Finally, an important point to focus on is to develop applications. There are many area of great impact for interactive computation where such an approach may be valuable. A few areas to consider may be: cluster/grid computing, web services, developing games, modeling biological systems, etc.

References

[Abramsky 1996] Abramsky, S.: "Retracing some paths in process algebra"; Proceedings of CONCUR'96, LNCS Vol. 1119, Springer (1996).

[Abramsky et al. 2004] Abramsky, S., Ghica, D.R., Murawski, A.S., Luke Ong C.-H.: "Applying game semantics to compositional software modeling and verification"; Proceedings of TACAS'04, LNCS Vol. 2988, Springer (2004).

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

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

[Dijkstra 1987] Dijkstra, E.W.: "Shmuel Safra's version of termination detection"; EWD Manuscript 998 (1987).
http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF.

[Dragoi and Stefanescu 2006a] Dragoi, C., Stefanescu, G.: "Structured programming for interactive rv-systems"; Institute of Mathematics of the Romanian Academy, Preprint 9/2006, Bucharest (2006).

[Dragoi and Stefanescu 2006b] Dragoi, C., Stefanescu, G.: "Towards a Hoare-like logic for structured rv-programs"; Institute of Mathematics of the Romanian Academy, Preprint 10/2006, Bucharest (2006).

[Dragoi and Stefanescu 2006c] Dragoi, C., Stefanescu, G.: "Implementation and verification of ring termination detection protocols using structured rv-programs"; Annals of University of Bucharest, Math.&Inf. Series 55 (2006), 129-138.

[Dragoi and Stefanescu 2007a] Dragoi, C., Stefanescu, G.: "Structured interactive programs with registers and voices and their verification"; Draft, Bucharest, January 2007.

[Dragoi and Stefanescu 2007b] Dragoi, C., Stefanescu, G.: "AGAPIA v0.1: A programming language for interactive systems and its typing system"; Proc. FINCO 2007, ETAPS Workshop on the Foundations of Interactive Computation, Braga, Portugal (2007), 61-76. ENTCS Volume, in press.

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

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

[Goldin et al. 2006] Goldin, D., Smolka, S., Wegner, P., (Eds.): "Interactive Computation: The New Paradigm." Springer (2006).

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

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

[Pratt 1992] Pratt, V.R.: "The duality of time and information": Proceedings of CONCUR'92, LNCS Vol. 630, Springer (1992).

[Preoteasa 1999] Preoteasa, V.: "A relation between unambiguous regular expressions and abstract data types"; Fundamenta Informaticae 40 (1999), 53–77.

[Stefanescu 2000] Stefanescu, G.: "Network algebra"; Springer (2000).

[Stefanescu 2002] 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.

[Stefanescu 2006] 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.

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

[Wadge and Ashcroft 1985] Wadge, W., Ashcroft, E.A.: "Lucid, the dataflow programming language"; Academic Press (1985).

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

A Appendix: Scenarios

A.1 Spatio-temporal data

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. Most of the usual data structures have natural temporal representations implemented on streams, see [Stefanescu 2006]. Examples: timed booleans, timed integers, timed arrays of timed integers, etc.

A.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. 6(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 it resumes its execution.

A *scenario* is a grid enriched with data around each letter. The data may be given in an abstract form as in Fig. 6(2), or in a more detailed form as in Fig. 6(3).

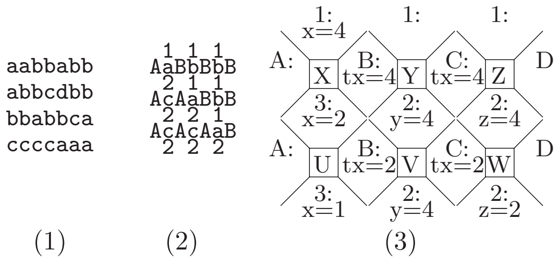


Figure 6: A grid (1), an abstract scenario (2), and a concrete scenario (3).

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. An empty tuple is also written 0 or *nil* and can be freely inserted to or omitted from 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. 6(c), the type is $\langle nil; nil|sn; nil; nil\rangle \to \langle nil; nil|sn; sn; sn\rangle$, where sn denotes the spatial integer type.

A.3 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 there exists a way to insert *nil* elements in these interfaces such that the resulting interfaces are equal.

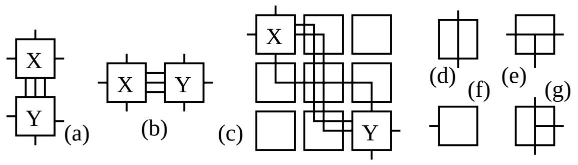


Figure 7: Operations on scenarios.

A.3.1 Constants

A few examples from Fig. 7: An identity Id (the center cell in (c)), a recorder R (2nd cell in the 1st row of (c)), a speaker S (1st cell in the 2nd row of (c)), empty cell Λ (last cell in the 1st row of (c)), a transformed recorder TR (e), and a transformed speaker TS (g).

A.3.2 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 $\overline{f_2}$, see Fig. 7(b).

A.3.3 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 is obtained putting $\overline{f_1}$ on top of $\overline{f_2}$. See Fig. 7(a).

A.3.4 Diagonal composition

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

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

for appropriate constants R, S, Id, Λ. See Fig. 7(c).

B Appendix: Structured rv-programs

B.1 The syntax of structured rv-programs

The syntax is given by the BNF grammar

$$\begin{aligned} P ::= {} & X \mid if(C)then\{P\}else\{P\} \mid P\%P \mid P\#P \mid P\$P \\ & \mid while\_t(C)\{P\} \mid while\_s(C)\{P\} \mid while\_st(C)\{P\} \\ X ::= {} & module\_name\{listen\ t\_vars\}\{read\ s\_vars\} \\ & \quad \{\ code;\ \}\{speak\ t\_vars\}\{write\ s\_vars\} \end{aligned}$$

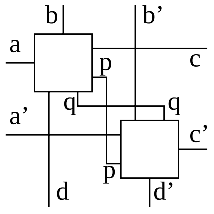
Structured rv-programs use modules X as their basic blocks. On top of them, larger programs are built up by "if" and both composition and iterative composition constructs for the vertical, the horizontal, and the diagonal directions. These statements aim to capture at the program level the corresponding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

B.2 On the choice of the composition/iteration operators

The number of composition and iteration operators used in the previous sections is pretty high. Our choice was driven by a practical approach, i.e., to have a set of easy to understand and use operators. In this appendix we show that all composition and iteration operators can be obtained from a unique composition and a unique iteration, albeit less structured and not so easy to handle.

B.2.1 General composition

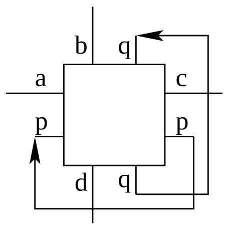
The general composition $\star$ is defined on two programs $E : \langle a|b\rangle \rightarrow \langle c;p|d;q\rangle$ and $E' : \langle a';p|b';q\rangle \rightarrow \langle c'|d'\rangle$. The result is a program $E \star E' : \langle a;a'|b;b'\rangle \rightarrow \langle c;c'|d;d'\rangle$.



From this general composition we get the following particular cases: (1) Case $d = b' = \mathsf{I}_0$ and $p = -_0$ (vertical composition); (2) Case $c = a' = -_0$ and $q = \mathsf{I}_0$ (horizontal composition); (3) Case $d = b' = \mathsf{I}_0$ and $c = a' = -_0$ (diagonal composition). On the opposite direction, general composition may be specified using vertical and horizontal composition and appropriate constants.

B.2.2 General iteration

The general iteration operator $\uparrow^{(p,q)}$ is a kind of 2-dimensional feedback: When it is applied to a program $E : \langle a;p|b;q\rangle \rightarrow \langle c;p|d;q\rangle$ it produces a program $E\uparrow^{(p,q)}: \langle a|b\rangle \rightarrow \langle c|d\rangle$.



As above, this general iteration operator cover the following particular cases: (1) Case $p = -_0$ (temporal while); (2) Case $q = \mathsf{I}_0$ (spatial while); (3) Case $a = c = -_0$ and $b = d = \mathsf{I}_0$ (spatio-temporal while). The proof is based on the representation of "while" by "feedback", see [Stefanescu 2000].

B.2.3 Programming constructs and examples

As previously stated, the three composition and iterated composition statements used in AGAPIA are instances of a unique pair of more general, but less "structured" statements:

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

In this case, at an identification border, only a part of the connecting interfaces are to be matched, namely the *tv* part at a temporal interface and the *sv* part at a spatial interface. Then, the horizontal form corresponds to the choice $tv = all$, $ts = \emptyset$, the vertical form to $tv = \emptyset$, $ts = all$, and the diagonal form to $tv = all$, $ts = all$. Currently, we do not know whether the general *while* may be simulated using the particular forms in $\{while\_t, while\_s, while\_st\}$.

Example 5. The structured rv-program for a termination detection protocol, presented in a previous 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}
      }]
```

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` with `while{all \ k}{all}(...)` we get a program where `k` is a temporal variable coming from the external temporal interface, not from `R`. This variable specifies the number of new processes that have to be inserted into the ring, a procedure that is handled by the new module `R` similarly as in `I2`; moreover, the counter for the numeber of processes is increased by `k`. Modeling the leaving of the ring is much easier: in the `R` code of the process which leaves the ring, the `write` part is replaced by `write nil` and the counter of the process number is decreased by 1. □

C Appendix: Operational semantics

The operational semantics

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

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

The type of a program P, denoted $P : \langle w(P)|n(P)\rangle \rightarrow \langle e(P)|s(P)\rangle$, indicates the types of its west, north, east, and south scenarios borders. Each of these types may be quite complex (see AGAPIA interface types in Sec. 2) Two interface types *match* if they have a nonempty intersection.

C.1 Modules

The modules are the starting blocks for building structured rv-programs. 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 concrete data on the borders, and such that the output data are obtained from the input data applying the module code.

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

C.2.1 Horizontal composition

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

C.2.2 Vertical composition

$P_1 \% P_2$ is similar.

C.2.3 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 $\langle w_1 | n_1 \rangle \rightarrow \langle e_2 | s_2 \rangle$. A scenario for $P_1 \$ P_2$ is a diagonal composition of a scenario in P_1 and a scenario in P_2.

C.3 If

Given two programs $P_i : \langle w_i | n_i \rangle \rightarrow \langle e_i | s_i \rangle$, $i = 1, 2$, 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 $Q : \langle w_1 \cup w_2 | n_1 \cup n_2 \rangle \rightarrow \langle e_1 \cup e_2 | s_1 \cup s_2 \rangle$.

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 is a scenario of P_2.

C.4 While

Given a program $P : \langle w | n \rangle \rightarrow \langle e | s \rangle$, we have introduced three while statements, each corresponding to the iteration of a composition operation.

C.4.1 Temporal while

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 $\langle (w;)^{*}|n \cup s\rangle \rightarrow \langle (e;)^{*}|n \cup s\rangle$. A scenario for $while\_t\ (C)\{P\}$ is either an identity, 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.

C.4.2 Spatial while

The spatial while "$while\_s\ (C)\{P\}$" is similar to the temporal one.

C.4.3 Spatio-temporal while

The statement $while\_st\ (C)\{P\}$ is defined if each pair of interfaces w, e and n, s match 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 $\langle w \cup e|n \cup s\rangle \rightarrow \langle w \cup e|n \cup s\rangle$. A scenario for $while\_st\ (C)\{P\}$ is either an identity, 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.

Notice that when the body program P of a temporal while has a dummy temporal interface, the temporal while coincides with the while from imperative programming languages. Similarly, for the spatial while it is easier to understand the case when the body $P : \langle w|n\rangle \rightarrow \langle e|s\rangle$ has dummy spatial interface, i.e., $n = s = nil$.

D Appendix: The syntax for AGAPIA v0.2 programs

The extended syntax introduced in the present paper provides the basis for a new version of AGAPIA, called AGAPIA v0.2. It is presented in Fig. 8.

Interfaces

$SST ::= nil \mid sn \mid sb$
$\quad \mid (SST \cup SST) \mid (SST, SST) \mid (SST)^*$
$ST ::= (SST) \mid (ST \cup ST) \mid (ST; ST) \mid (ST;)^*$
$STT ::= nil \mid tn \mid tb$
$\quad \mid (STT \cup STT) \mid (STT, STT) \mid (STT)^*$
$TT ::= (STT) \mid (TT \cup TT) \mid (TT; TT) \mid (TT;)^*$

Expressions

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

Programs

$W ::= nil \mid new\ x : SST \mid new\ x : STT$
$\quad \mid x := E \mid if(B)\{W\}else\{W\}$
$\quad \mid W; W \mid while(B)\{W\}$
$M ::=$ **module** $module\_name\{listen\ x : STT\}\{read\ x : SST\}$
$\quad \{\ W\ \}\{speak\ x : STT\}\{write\ x : SST\}$
$\quad \mid$ **module** $module\_name\{listen\ x : STT\}\{read\ x : SST\}$
$\quad \{\ ScatterT\ \#\ (ScatterS\ \%\ \ P\ \ \%\ GatherS)\ \#\ GatherT\ \}$
$\quad \{speak\ x : STT\}\{write\ x : SST\}$
$P ::= nil \mid M \mid if(B)\{P\}else\{P\}$
$\quad \mid P\%P \mid P\#P \mid P\$P$
$\quad \mid while\_t(B)\{P\} \mid while\_s(B)\{P\}$
$\quad \mid while\_st(B)\{P\}$
$ScatterS ::=$ **module** $module\_name\{listen\ nil\}\{read\ x : SST\}$
$\quad \{\ body;\ \}\{speak\ nil\}\{write\ x : ST\}$
$ScatterT ::=$ **module** $module\_name\{listen\ x : STT\}\{read\ nil\}$
$\quad \{\ body;\ \}\{speak\ x : TT\}\{write\ nil\}$
$GatherS ::=$ **module** $module\_name\{listen\ nil\}\{read\ x : ST\}$
$\quad \{\ body;\ \}\{speak\ nil\}\{write\ x : SST\}$
$GatherT ::=$ **module** $module\_name\{listen\ x : TT\}\{read\ nil\}$
$\quad \{\ body;\ \}\{speak\ x : STT\}\{write\ nil\}$
(The *body* part in these Statter/Gather modules use only $x := y$ assignments, i.e., no real computation, only copy/delete data.)

Figure 8: The syntax for AGAPIA v0.2 programs