PERGAMON

Computer Languages 26 (2000) 67–104

COMPUTER LANGUAGES

www.elsevier.com/locate/complang

Iteration constructs in data-flow visual programming languages

M. Mosconi, M. Porta*

Dipartimento di Informatica e Sistemistica, Università di Pavia, Via Ferrata, 1, 27100-Pavia, Italy

Received 12 January 2001; received in revised form 3 March 2001; accepted 7 April 2001

Abstract

Many visual programming languages (VPLs) rely on the data-flow paradigm, probably because of its simple and intuitive functioning mechanism. However, there are cases where more powerful programming constructs are needed to deal with complex problems. For example, iteration is undoubtedly an important aspect of programming, and should allow repetitive behaviors to be specified in compact and easy ways. Most existing data-flow VPLs provide special constructs to implement iterations, therefore infringing the pure data-flow paradigm in favor of program simplicity. This paper has three main purposes: (1) To provide a survey of the mechanisms used by some representative data-flow VPLs to carry out iterations; (2) To investigate, given a pure data-flow VPL, what should be the minimum set of characteristics which, after being added to the VPL, allow iterations to be implemented; and (3) To show real data-flow iteration implementations which rely on the characteristics pertaining to such a minimum set. © 2001 Elsevier Science Ltd. All rights reserved.

Keywords: Visual programming; Data-flow visual languages; Iterations

1. Introduction

Although the debate about the usefulness of visual programming languages (VPLs) compared with textual ones is quite far from being over, it is indisputable that, at least for certain applications, interacting with objects placed in a two-dimensional space may be extremely worthwhile [1]. In fact, graphic elements have the advantage of being characterized by shape, dimension, position and possibly color, all attributes which may help better understand the meaning of what is displayed on the screen. As far as we are concerned, we think that visual programming, if properly exploited, holds very great potential and can speed up reasoning processes.

* Corresponding author. Tel.: +39-0382-505372; fax: +39-0382-505373.
E-mail address: marco.porta@unipv.it (M. Porta).

Data-flow is one of the most popular computational models for visual programming languages, and provides a view of computation in which data flows through filter functions being transformed as it goes [2]. Data-flow VPLs are widespread and used for various purposes, ranging from image processing to data mining with forms, to construction of control panels for laboratory instruments. In many cases, they give excellent support for rapid prototyping and user interface construction activities (see, for example, LabView [3,4] and VEE [5,6]). Moreover, constructed visual data-flow programs are often readable for end users, thereby simplifying communication between software developers and software users.

In the pure data-flow model (that is, in the data model with no added control-flow constructs) the sequence in which functions or nodes must execute is not specified: a node can fire when all of its inputs are available [2]. Indeed, a powerful and useful data-flow VPL must provide all the necessary programming constructs to deal with complex problems (in the language's application domain). Our experience with the VIPERS system [7,8], developed at the University of Pavia, confirms once again that the pure data-flow model needs to be enriched with some forms of control-flow constructs in order to tackle non-trivial applications.

Iterations, for instance, have been provided in different ways in several data-flow languages [2]. Nevertheless, we feel that satisfactory solutions are very difficult to achieve, because sometimes these solutions use a notation which is not consistent with the data-flow paradigm. Moreover, in general, visual control structures may be difficult to write and even more difficult to understand.

This paper is structured into three main parts, each with a precise purpose. The first part surveys solutions adopted by some representative data-flow VPLs to carry out iterations, through practical examples. Many other data-flow VPLs solve the iteration problem in ways traceable to those presented in the survey. The second part aims at exploring certain aspects of the data-flow paradigm which can be exploited to implement iterations using "artificial" mechanisms as little as possible. That is, it tries to answer the question: "What should be the minimum set of characteristics which added to a pure data-flow VPL allow iterative behaviors to be implemented?". Lastly, the third part illustrates practically some real iterative constructs implemented through the VIPERS visual language, exploiting concepts emerging during the analysis carried out in the second part.

2. Iteration forms for data-flow visual programming languages

Whatever kind of programming language one may be concerned with, be it textual or visual, *iteration* means repeating a body of code several times, usually for repeated modifications of some variables.

More precisely, two basic iteration forms can be distinguished [9]: *horizontally parallel iteration* and *temporally dependent iteration* (also called *sequential iteration*). Whereas in the former the outcome of one cycle does not affect the outcome of the next one, the latter implies such a dependence.

A typical example of horizontally parallel iteration is: 'do for all' elements in a set (any data structure) a certain number of operations. Since in this form each iteration is completely independent of the others, there is no conflict with the notion of *single assignment*, which says that once a variable has been bound to a value, it remains bound to that value. The concept of parallel iteration, therefore, can easily be added to any data-flow VPL, without abandoning the concept of *stateless programming* [10].

However, temporally dependent iteration raises the problem of how to let the data-flow paradigm infringe the single assignment rule. Although one way of overcoming this problem is recursion, which is computationally adequate, it is indisputable that there are arguments for not totally abandoning iteration. A case in point is the fact that many algorithms are most naturally expressed iteratively [9].

Two main approaches have been proposed to tackle sequential iteration in data-flow VPLs [10]. One avoids cycles completely, being based on the creation of a "fresh copy" of the iterated subdiagram for every loop execution. Such an approach reflects the opinion that iteration is just a special form of recursion (*tail recursion*).

The other approach for tackling sequential iteration, instead, admits the presence of cycles within the program graph. Essentially, every function or "variable" is considered as having a stream of values associated with it, instead of a single value. At the entrance of the looped subgraph, pairs of data inputs (old, new) follow one another during the various iteration steps, so that temporal dependencies can be properly treated.

3. Iteration constructs provided by some representative data-flow VPLs

Existing data-flow VPLs adopt different solutions to face the problem of how to implement sequential and parallel iterations within program graphs. Here, we have selected five languages which, for historical reasons or because they are particularly widespread, can be considered as representative for a wide range of data-flow VPLs. Practical examples will help us to understand the basic mechanisms underlying the various solutions adopted and will provide clues for better comprehending the ways they are exploited.

3.1. Show and Tell

Show and Tell [11] is a general-purpose language mainly intended for school children. Graphically, it is based on *boxes*, i.e. rectangles which can represent functions, constants, variables, and other elements of the language. Data flows through links (lines with arrows) connecting boxes. The concept of variable also exists in Show and Tell. Practically, a variable is an empty box which can contain data. A *boxgraph* consists of one or more boxes, possibly connected by links.

When a data value is transferred to a box which already holds a different value or when an *open* box containing a predicate (e.g. '> 0') is evaluated to be false, the box's boxgraph becomes *inconsistent*. Inconsistency is a key concept in Show and Tell, in that it allows a form of control over the computation.

3.1.1. Sequential iteration in Show and Tell

Fig. 1 shows a Show and Tell program which calculates the factorial of a positive integer. The program of Fig. 1 visually implements the following (textual) C code fragment, where 'n' is the

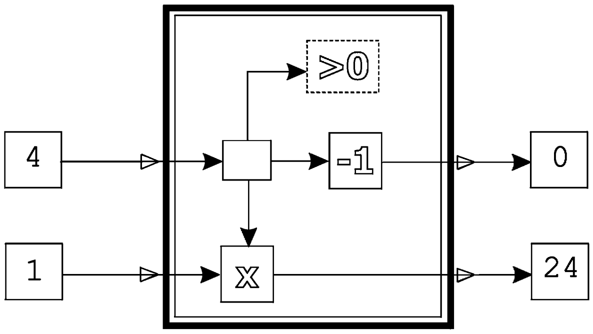


Fig. 1. Factorial calculus in Show and Tell.

number whose factorial is to be calculated:

```
int n, factorial;
scanf("%d", n);
factorial = 1;
while (n > 0) {
  factorial = factorial * n;
  n--;
}
printf("%d", factorial);
```

The big double rectangle in the figure is an *iteration box*, which means that what is included within it constitutes the body of an iteration. The empty arrows on the left and on the right of the box are *sequential ports*. The two boxes on the left contain constant values and, precisely, '4', in this example, is the number whose factorial is to be calculated. Function boxes '-1' and 'x' perform the corresponding arithmetic operations on their inputs, once they are all available (according to the data-flow model).

When program execution starts, values '4' and '1' are provided to the iteration box. Therewith, the '4' value is first transferred to a variable box (the empty one) and, from here, to functions 'x' and '-1' and to predicate '>0'. At the same time, value '1' is provided to function 'x'. If the variable box's content is not greater than zero, an inconsistency is generated and program execution halts. Otherwise, the value in the variable box is decreased by one by function '-1' and placed into the upper box outside the iteration structure. Concurrently, function 'x' multiplies its two inputs ('4' and '1') and gives out the result, which is placed into the lower box outside the iteration structure. At this point, the first step of the iteration is finished and a new one starts. Values within the boxes on the right of the iteration box virtually re-enter the boxes on the left, whose initial values are now substituted with the new ones. The process goes on until the variable box's contents reaches '0'.

In practice, such an iteration mechanism behaves as if subsequent identical boxgraphs (one for each step of the iteration) were arranged so that a boxgraph's outputs are the inputs for the next one (Fig. 2).

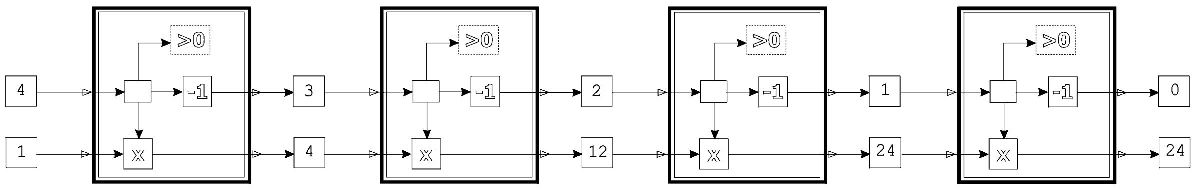


Fig. 2. The iterative program in Fig. 1 “unfolded”.

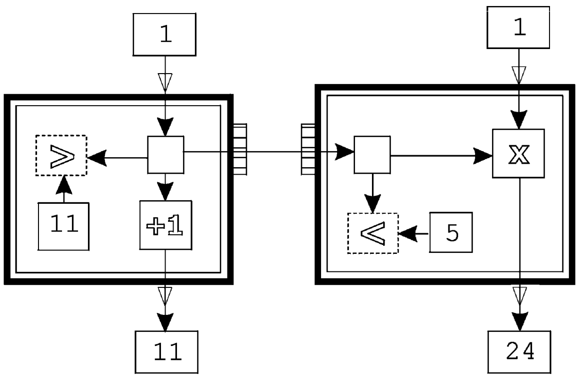


Fig. 3. An example of parallel iteration in Show and Tell.

3.1.2. Parallel iteration in Show and Tell

Fig. 3 shows a program composed of two iteration boxes.

The first box (on the left) produces a sequence of numbers, from 1 to 10, and the second (on the right) consumes the sequence, by selecting the numbers less than 5 and calculating their product. Although both these iterations, singly, are temporally dependent, here we are only interested in the process of extracting some data from a set, which is a typical parallel activity.

The two iteration boxes communicate with each other through a link crossing their *parallel ports* (the striped rectangles). An iteration box can have any number of parallel ports and more than one arrow may pass through them. Referring to Fig. 3, at each iteration step data contained in the variable box within the first iteration structure is provided to the variable box within the second iteration structure, therefore implementing a form of parallel iteration.

When the variable’s value reaches 11, an inconsistency is generated in the first iteration box and the process ends.

3.2. LabView

LabView [3,4] is a language designed to create software interfaces for hardware devices, especially in the field of real time data acquisition. Program modules (or procedures) are called *virtual instruments* (VIs). Each instrument has an associated front panel which represents the user interface to the procedure and is capable of accepting various types of inputs and providing imagery as feedback. Essentially, a LabView program is made up of a graph whose nodes identify functions, variables and constants.

Two iterative forms are available in LabView: the *FOR loop* and the *WHILE loop*. In both the iteration's body, that is the program subgraph whose execution is to be repeated, is placed within a proper pane (a square-shaped special big block). For the FOR loop, an integer value indicating the number of iterations to be accomplished must be specified, whereas in the case of the WHILE loop the cycle execution is repeated for as long as a boolean condition, visually expressed, is satisfied.

3.2.1. Sequential iteration in LabView

Since LabView's program graphs are acyclic (data flows from left to right only), a mechanism for updating possible variables depending on previous iterations is necessary. Fig. 4 shows a LabView program which calculates the factorial of a positive integer using the FOR loop construct.

In the program of Fig. 4, the `n` in the double box on the left is a *control*, that is, an input specifying the integer number whose factorial is to be calculated.

It is connected to the `N` in the upper left corner of the FOR loop pane representing the iteration body, meaning that the loop will be executed for `n` times.

The `i` in the single box within the pane is the loop variable and is initialized to zero. The `1`'s in the single boxes are constants.

The textual (C) code corresponding to this visual program is the following:

```
int n, factorial, i;
scanf("%d", n);
factorial = 1;
for (i = 0; i < n; i++)
  factorial = factorial * (i + n);
printf("%d", factorial);
```

A downward arrow can also be noted on the left-hand side of the pane representing the iteration's body, as well as an upward arrow on its right. They can be thought of as input and output ports for the loop: an arc connected to an upward arrow on the right virtually re-enters through the horizontally corresponding downward arrow on the left during the next iteration step, thus implementing a virtual cycle. In other words, these input and output ports behave as *shift registers* transferring data from one step of the loop process to the following one, in a manner very similar to Show and Tell's. A loop (both FOR and WHILE) can have any number of input/output ports, according to its needs.

Referring to Fig. 4, at the beginning of the iteration, constant `1` is provided to the FOR loop (through the downward arrow) as an initial value. Such a value is an input for *function* `x`, along with the loop index `i` (whose initial value is zero) increased by one by function `+`. The output of function `x` represents the partial factorial and is connected to the upward arrow on the right of the loop pane, thus exiting it. At this point, the current iteration step is finished, the partial factorial re-enters the loop body through the downward arrow on the left and a new step starts. Iteration stops when variable `i` reaches `n-1`.

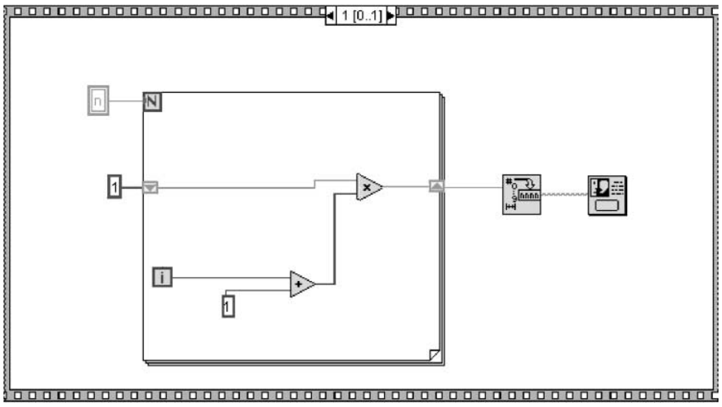


Fig. 4. Factorial calculus in LabView.

Fig. 5 shows a LabView program using the WHILE loop construct. Its purpose is to calculate the square root of a number through the Newton algorithm. The C code corresponding to this visual program is the following:

```
float alfa, x0, noise_limit, x_old, x_new, error, residual_error;
scanf("%f", alfa);
scanf("%f", x0);
scanf("%f", noise_limit);
scanf("%f", residual_error);
x_old = x0;
do {
  x_new = x_old - (x_old * x_old - alfa - noise_limit)/(2 * x_old);
  error = abs(x_new - sqrt(alfa));
while (error > residual_error);
printf("%f", x_new);
```

As can be noted from the textual transposition of the visual program, despite its name, the WHILE construct is really a DO ... WHILE (or REPEAT ... UNTIL) loop structure.

Referring to Fig. 5, ‘x0’ is a *control* (that is, an input in the user interface) indicating the initial value from which to start with the Newton algorithm.

Along with control ‘noise limit’, which simulates a noise component, it is actually an input to the loop. ‘alfa’ is the number whose square root is to be calculated.

Once the first step of the WHILE iteration has been completed, *predicate* ‘$>$’ tests whether the error is greater than the value specified through control ‘residual error’. If so, iteration must go on. Otherwise, iteration must stop. Therefore, in this example, predicate ‘$>$’ is actually the WHILE loop condition, as indicated by the fact that its output is connected to the circular arrow icon in the lower right-hand corner. The output of the upper ‘-’ function is connected to the upward arrow on the right of the WHILE loop pane and re-enters from the downward arrow on the left at the next iteration step. The iteration process ends when the WHILE loop condition is no longer true.

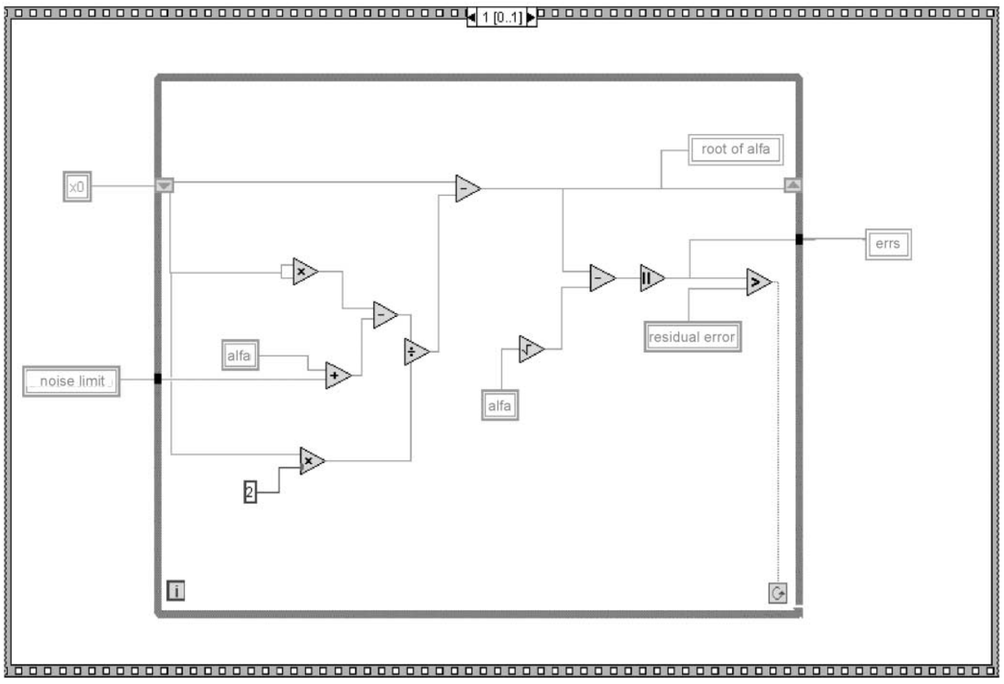


Fig. 5. Calculating the square root of a number through the Newton algorithm in LabView.

3.2.2. Parallel iteration in LabView

Consider the following C code fragment:

```
float sum, i;
sum = 0;
for (i = 0; i < 100; i++)
  sum += x[i] * x[i];
```

All the elements of array ‘x’ are accessed sequentially. Actually, since the value of variable ‘sum’ is updated at each step of the loop, this is also a sequential iteration. However, here we are interested only in the process of extracting elements from a data structure, quite apart from what such elements will be used for.

In Fig. 6, three LabView solutions to the problem are shown [4].

Solution (a) uses *direct translation* for accessing elements of ‘x’, which is at the entrance of the FOR loop body. String ‘EXT’ enclosed within two square brackets indicates that ‘x’ is an array of extended precision floating point numbers. This array, along with the loop variable ‘i’, is connected to a particular function icon which extracts the ith element, thereby allowing all the elements to be accessed during the whole loop process.

Solution (b) uses *auto-indexing*, a LabView feature which allows elements of an array to be automatically extracted, according to the iteration variable value, when the array is connected to a FOR loop pane: the element to be extracted is implicitly addressed by loop variable ‘i’.

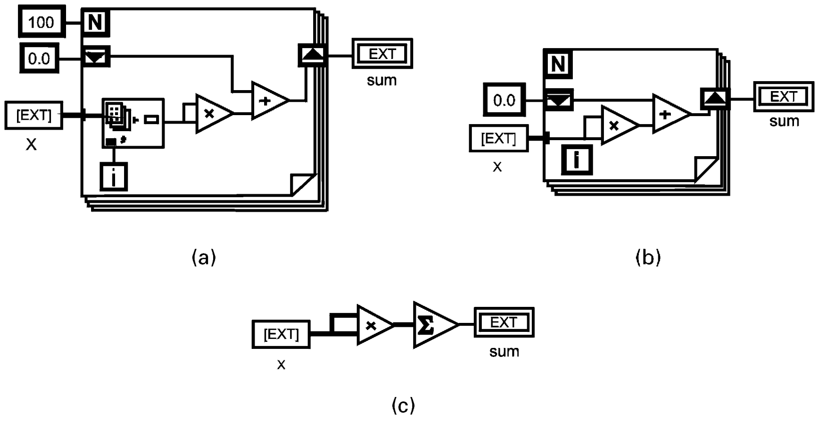


Fig. 6. Extracting elements from an array in LabView.

Solution (c), at last, is the optimal, compact solution: elements of the array are implicitly extracted and processed, one at a time.

3.3. Prograph

Prograph [12,13] is a general-purpose language combining the data-flow model with object-oriented programming. All elements of Prograph take the form of icons in windows and text is used only for naming elements and for comments which can be attached to those elements.

Prograph programs are made up of *methods* (*class methods* if they pertain to a class, *universal methods* otherwise). A method consists of a sequence of one or more *cases*, which are distinct data-flow diagrams of *operations*. Operations, in turn, are the basic executable components of a case and look like rectangular-shaped icons, with a name inside them. Data flows from top to bottom, and operations can execute whenever all their inputs are available, according to the data-flow model.

Control flow, in Prograph, is based on method cases. When a method is called, execution begins with its first case, which, typically, performs some processing. Then, based on the value of some condition, it either terminates or passes control to the next case. Method cases are executed in sequence, until one succeeds completely or until activation of a *control* stops execution of the method.

3.3.1. Sequential iteration in Prograph

To carry out sequential iteration, Prograph uses an implicit control flow construct similar to a WHILE loop, which can be applied to methods. Fig. 7 shows two cases (one for method *calculate factorial* and one for method *factorial loop*) making up a Prograph solution to the factorial calculus problem.

A case consists of an *input bar* (at the top) and an *output bar* (at the bottom) which, respectively, pass parameters to the case and return values, and in between, operations with connecting data links. Data flows into a case through the input bar, and out of the case through the output bar. Each case of a method has the same number of inputs and the same number of outputs (that is, the same arity).

Referring to Fig. 7, execution starts with method *calculate factorial*. Operation '`ask`' opens a dialog box prompting the user for input (the number whose factorial is to be calculated). Prograph

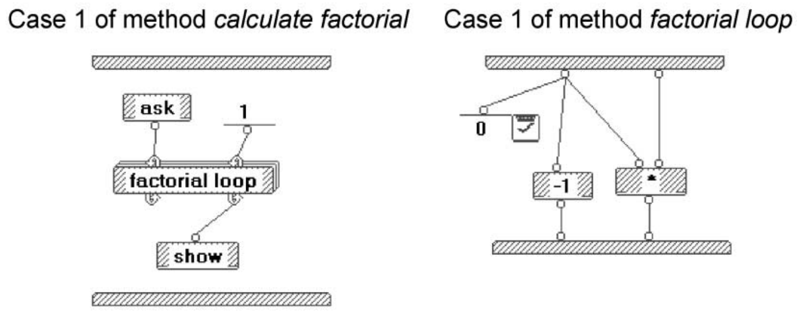

Case 1 of method *calculate factorial*
Case 1 of method *factorial loop*
ask
1
factorial loop
show
0
-1
*

Fig. 7. Factorial calculus in Prograph.

implements sequential iteration through a particular loop mechanism called *loop annotation*, which allows repeated calls to be made to a method, passing the output of each iteration step to the input of the next one. In Fig. 7, loop annotation is applied to method *factorial loop* (note its 3D appearance). The circular arrows at the top and at the bottom of the method's icon highlight the fact that data exiting the method at one step of the iteration will re-enter the method, as inputs, at the next step.

Also method *factorial loop* is made up of one case only. Its first input (that on the left) is tested by a "conditional" operation to check if it is zero. The overlined check mark at the right of the operation is a *terminate* control, which, if the condition is satisfied, causes both the execution of the case and the entire loop process to be halted. Otherwise, if the first input is not zero, its value is decreased by one by operation '`-1`' and given out as a first output. Then, operation '*' multiplies the two inputs of the method and yields the result as a second output. Once both the outputs are available, they become inputs for the next step of the iteration and virtually re-enter the method. The process continues until the first input is zero. Operation '`show`' in method *calculate factorial*, at last, displays the value of the just calculated factorial.

3.3.2. Parallel iteration in Prograph

Prograph provides a mechanism called *list annotation* to apply an operation to every item of a list. Fig. 8 shows five cases (one for method *Qsort*, two for method *quicksort* and two for method *GTE*) making up a Prograph solution to the quick sort algorithm, for sorting a list of numbers.

Execution starts with case 1 of method *Qsort*, which asks the user for a list of numbers, sorts it by means of the recursive method *quicksort* and then shows the resulting list.

In method *quicksort*, operation '`detach-l`' receives a list and gives out its head (first output) and its rest (second output). Operation '`attach-l`' attaches an element (first input) in front of a list (second input) and yields the resulting list as an output. Operation '`(join)`' joins its two input lists. The input to case 1 of method *quicksort* is connected both to operation '`detach-l`' and to a conditional operation as well. It checks whether the input is an empty list and, if so, stops the current case, causing program execution to pass to case 2. The check mark symbol at the right of the operation, in fact, is a *next case on success* control.

List annotation is applied to method *GTE*, which is made up of two cases. It receives an element (a number) and a list as inputs and produces two lists. In the first (on the left) there are all the elements of the input list which are greater than or equal to the input number, while in the second there are the other elements. Referring to the cases of method *GTE* in Fig. 8, it can be noted that

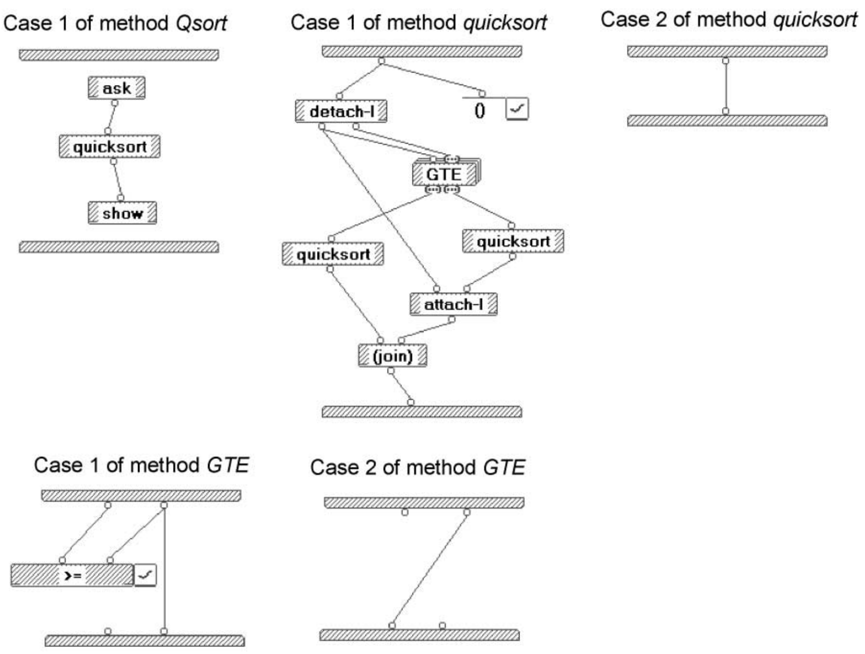


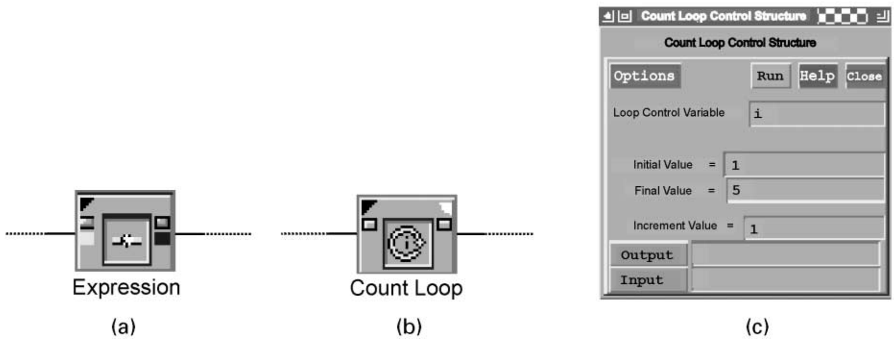
Fig. 8. The *quicksort* algorithm in Prograph.

case 1 applies if the first input (a number) is lower than the second one (also a number) and that it generates an element for the second output only. In contrast, case 2 applies if the first input is greater than or equal to the second and generates an element for the first output only. List annotation implicitly repeats method *GTE* for all the elements of the input list, thus generating, as a result, two separate sequences of numbers (the output lists).

The described list annotation process for method *GTE* is equivalent to the following C code portion, where variable 'val' stands for the first input (the number) and array 'x' stands for the second input (the list). Array 'left' corresponds to the first *GTE*'s output list, while array 'right' corresponds to the second one. Variable 'x_length' holds the length of array 'x'.

```
int  i, j, k, left[], right[];
j=0;
k=0;
for (i=0; i < x_length; i++)
  if (val < x[i])
    right[k++]=x[i];
  else
    left[j++]=x[i];
```

Since the outputs of method GTE are partitions of its input list, another annotation mechanism (*partition annotation*) could have been conveniently used. However, the underlying basic concept is fundamentally the same.



Loop Control Variable i

Initial Value = 1

Final Value = 5

Increment Value = 1

Fig. 9. An example of sequential iteration in Cantata.

3.4. Cantata

Cantata [14,15] is a language which provides a visual programming environment within the Khoros system (a software integration and development environment for information processing and data exploration).

In Cantata, icons (called *glyphs*) are typically used to represent programs from the Khoros system, although, if properly integrated, they can be used to represent non-Khoros programs as well. Since each glyph corresponds to an entire process rather than to a code segment or a sub-procedure, Cantata programs are referred to as *coarse grained*. Glyphs may be executed both locally and remotely, to exploit a heterogeneous network of computers.

Cantata programs are acyclic. A program may have global variables associated with it, i.e. alphanumeric symbols denoting, for example, numeric values. Such variables, which are globally defined and accessible through a proper pane (a dialog window), can be set to particular initial values and modified by glyphs during program execution. Since variables are textual and not directly visible, they represent a sort of hidden information within the visual program. The two iteration forms provided by Cantata, the *Count Loop* and the *While Loop*, are based on the same philosophy, concealing the loop implementation details from the programmer and founding iteration on variable values and on conditions defined in the control panel of the loop glyphs.

To create an iterative construct, those glyphs pertaining to the program's subgraph which will form the loop's body, must be selected. The glyphs picked will be replaced with a new single glyph (Count Loop or While Loop). Following this, loop behavior must be defined through a proper pane. In the case of the Count iteration form, the programmer has to specify a loop variable and its initial and final values, as well as the increment amount at every step. Instead, for the case of the While iteration form, the conditional expression (generally involving variables) must be provided which will be used to determine how many times the loop is to be executed (and, possibly, an initial and an update expression). Loop glyphs' input data are the same at every iteration step.

3.4.1. Sequential iteration in Cantata

In Cantata, data re-circulation is not visual and can only be possible through the use of variables. Suppose, for example, we want to calculate the factorial of a positive integer. Cantata provides a glyph named '`Expression`' (Fig. 9a) to update a variable according to an arbitrary expression, which

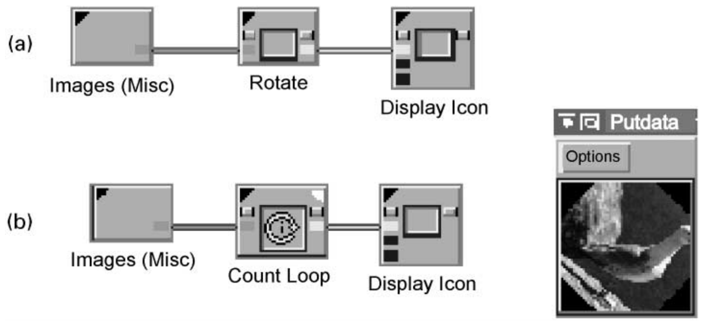


Fig. 10. An example of parallel iteration in Cantata.

can be specified through the glyph's pane. Suppose we specify '`fact = fact * i`' as an expression, where '`fact`' and '`i`' are variables. Then, suppose we create a Count loop which incorporates the just defined '`Expression`' glyph (Fig. 9b). The Count loop pane requires some data to be specified (Fig. 9c), and precisely: a name for the loop variable (for this example, '`i`'), its initial value ('`1`'), its final value (the number whose factorial is to be calculated, say '`5`'), and an increment value ('`1`'). When the program runs, glyph '`Expression`' (and therefore expression '`fact = fact * i`') is executed five times.

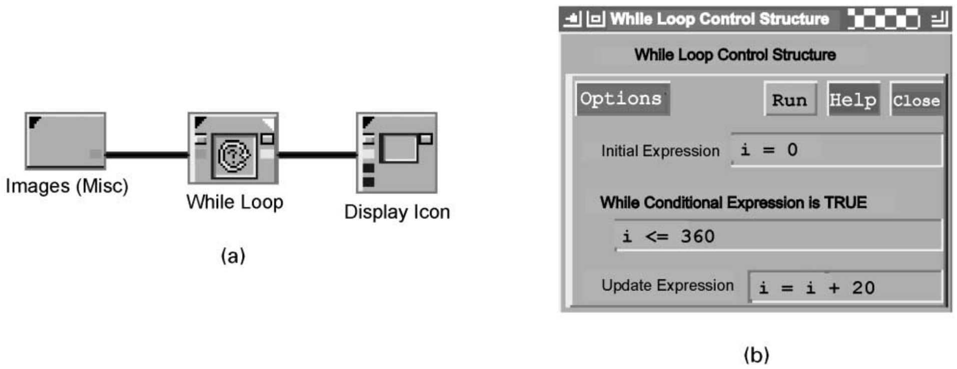
If variable '`fact`' has been initialized to '`1`', at the end of the loop process it will hold the value of the factorial.

3.4.2. *Parallel iteration in Cantata*

Fig. 10b shows a small Cantata program, deriving from that in Fig. 10a, which uses the Count loop to repeatedly rotate an image.

In this example too, the *Count loop* block incorporates only one glyph, '`Rotate`', which receives an image as an input and rotates it according to an angle specified in its pane (for this example, suppose the rotation angle is specified by variable '`i`'). As already stated, the *Count loop* parameters, which have to be entered through its pane, are: a name for the loop variable (for this example, '`i`'), its initial value (say '`0`'), its final value (say '`360`') and an increment value (say '`20`'). Then, when the program runs, the image at the input of the Count loop glyph is rotated by 20 degrees for 18 times ($360 : 20 = 18$). It should be noted that each time glyph '`Rotate`' is executed, during the loop process, its input image always remains the same (not rotated, since there is no data re-circulation). This is the reason why the rotation angle value for glyph '`Rotate`' must be the Count loop variable '`i`'.

The same effect as achieved by the program in Fig. 10b could also have been obtained by using a While loop instead of the Count loop. The program in Fig. 11a behaves exactly as Fig. 10b but uses a While glyph instead of the Count glyph. The While loop parameters, which must be entered through its pane (Fig. 11b), are: an initial expression (for this example, '`i = 0`'), a conditional expression ('`i <= 360`') and an update expression ('`i = i + 20`'). When the program runs, the image at the input of the While loop glyph gets rotated by 20 degrees for 18 times.



While Loop Control Structure

Initial Expression i = 0

While Conditional Expression is TRUE

i <= 360

Update Expression i = i + 20

Fig. 11. Using the While loop in Cantata.

3.5. VEE

VEE [5,6] is a language optimized for scientific and engineering applications, particularly in testing and measurement. Programs are built by selecting *objects* (rectangular-shaped “blocks”) from menus and wiring them together. Like LabView, VEE allows to build user interfaces with both input and display objects and simplifies test development with enhancements for system integration, debugging, structured program design and documentation.

VEE objects are small windows, each with a title, *data input ports* (on the left), *data output ports* (on the right), *synchronization pins* (on the top and on the bottom) and possible *entries*, through which constant input data can be directly specified. Object windows can either be “open” or, to occupy less space, “closed”.

Since VEE programs, with some restrictions, are allowed to contain cycles, explicit data recirculation is permitted. However, VEE provides special predefined objects (three *For* objects, an *Until Break* object and an *On Cycle* object) which can be used to implicitly create loop behaviors.

3.5.1. Sequential iteration in VEE

Fig. 12 shows a VEE solution to the factorial calculus problem. The textual transposition of this visual program is the same as for LabView (already presented in Section 3.2.1).

Referring to Fig. 12, object *Integer* represents a constant, which is emitted when the object is activated (as soon as the program starts, given that it has no input). Object *JCT* is often used in feedback loops and is an exception to the standard data-flow firing mechanism, in that it emits the first input it receives instead of requiring both inputs to be available.

Object *Integer Input* (which, in this example, is closed) receives an integer number by means of an input dialog box it shows to the user, and gives it out. Object *For Count* allows a program subgraph to be executed a certain number of times (say ‘n’), which can be specified directly within it or by means of the input it receives (like in this example). Starting from zero up to ‘n−1’, object *For Count* emits the current loop counter value through its output port. The link connecting the *sequence out* pin on the bottom of object *For Count* to the *sequence in* pin on the top of object *AlphaNumeric* implements a form of synchronization. It guarantees that *AlphaNumeric* (whose purpose is to show its input data) will be activated only when iteration is completed, therefore preventing it from displaying

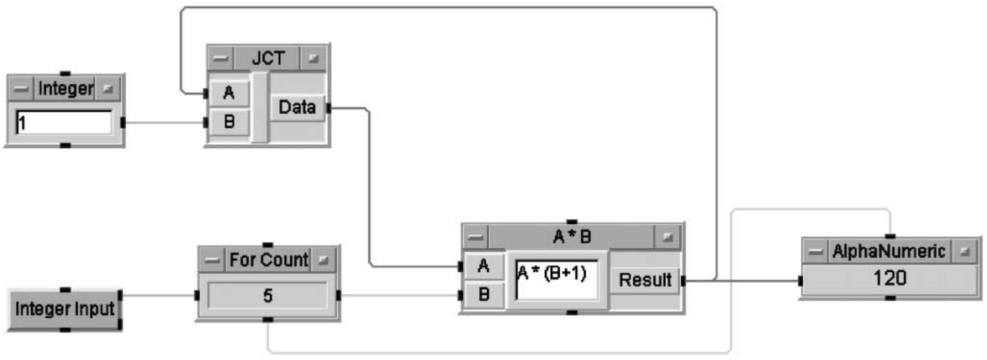


Fig. 12. Factorial calculus in VEE.

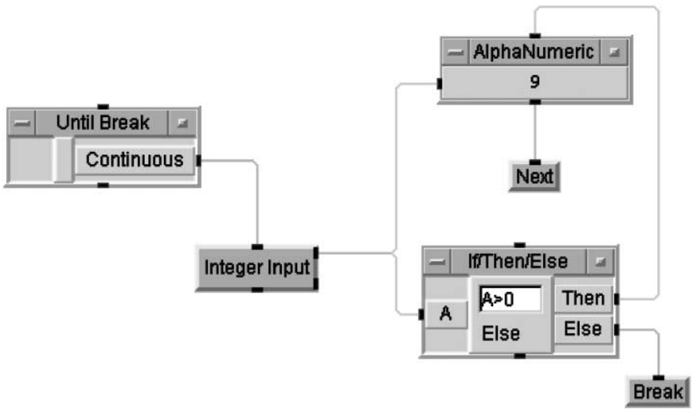


Fig. 13. A conditional iterative structure obtained through object *Until Break* in VEE.

the partial factorials at each step of the iteration. Object $A * B$ multiplies its inputs, directly or, like in this example, according to an expression specified in its entry.

Since the program graph needs to implement sequential iteration by cycles, the output of object $A * B$ is connected both to object *AlphaNumeric* and to an input of object *JCT* as well.

VEE also allows to set and read variables, through proper objects, which could be used to implement sequential iteration without introducing cycles into the program graph. However, such a mechanism is quite distant from the data-flow model and therefore will not be discussed here.

A conditional iterative structure similar to a WHILE loop can be obtained in VEE through the use of objects *Until Break*, *Next* and *Break*. Fig. 13 shows a simple program which keeps requesting an integer input from the user until a negative value is entered. When the program starts, object *Until Break* activates its output, which enables object *Integer Input*.

Then, the user is asked for an integer number, which is provided both to objects *If/Then/Else* and *AlphaNumeric*. Object *If/Then/Else* branches execution flow based on the values of its inputs. According to whether the condition expressed in its entry is satisfied or not, the `Then` or `Else` path will be enabled or not.

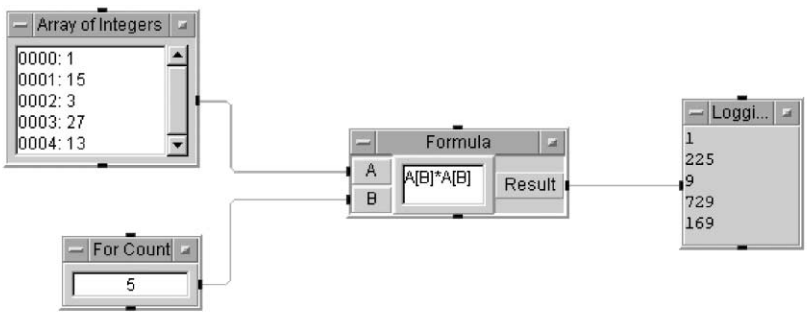


Fig. 14. Raising an array's numbers to the power of two in VEE.

In our example, if the '`Then`' path is followed, object *AlphaNumeric* displays the integer number just inserted and then object *Next* causes object *Until Break* to fire again, therefore starting another step of the iteration. If instead the '`Else`' path is followed, object *Break* stops the loop process.

3.5.2. Parallel iteration in VEE

VEE supports arrays and some operations which can be performed on them. Array traversals are typically accomplished through the *For Count* implicit loop structure.

Fig. 14 shows a VEE program calculating the squares of integer numbers contained in an array.

Object *Array of Integers* defines a constant array of integer numbers, which is provided to object *Formula*, as a first input, when the program starts.

Object *For Count* emits the iteration index value, five times, from 0 to 4. This index is used by object *Formula* to select each element from the array, whose square is then calculated and supplied to object *Logging Alphanumeric*, which displays it.

4. In the end, can the data-flow paradigm be used to implement iterations?

So far, we have surveyed the solutions adopted by some representative data-flow visual programming languages to carry out iterations, both sequential and parallel. All of them, to a lesser or greater degree, are "equipped" with mechanisms which aim at simplifying user interaction with the visual language when implementing iterations. But what about the data-flow paradigm? Is it respected? It seems not.

It is indisputable that generally, though not always, there is a tendency to consider cycles within visual program graphs as something to be avoided at all costs, because of the confusion they may generate. However, if such an attitude is comprehensible, because cyclic graphs sometimes may represent a serious hindrance to program comprehension, it is also true that in iteration processes the natural flow of data is better expressed through cycles, which do not need for implicit information or specific interpretation conventions.

In the following sections, we will first semi-formally investigate some aspects of the data-flow model which can be used to implement iterations using "artificial" mechanisms as little as possible, compared to the pure data-flow paradigm. Then, we will discuss some practical implementations of iterative structures in the VIPERS visual language. In doing our analysis, however, *we are not* *claiming that the proposed data-flow solutions are better than others*, for iteration implementations; we are only exploring their potentials and characteristics, which, often, have not been taken into account due to their apparent incompatibility with the notion of iteration itself.

4.1. Minimum "Add-On"s required by a pure data-flow VPL to implement iterations

Before starting with our analysis, an important premise is necessary. Data-flow systems can execute in two different ways, according to when a *function* (a node of the data-flow graph) is allowed to *fire* (i.e. transform its inputs and emit its outputs). In *data-driven* systems a function fires as soon as all its inputs are available. In contrast, in *demand-driven* systems a function fires as soon as all its inputs are available and at least one of its outputs is required by another function. Although the final output of the program is the same for both cases, the demand driven execution model has the advantage that it can prevent certain functions from firing unnecessarily, if the final output of the program does not need those functions' output data. In the following, we will explicitly indicate when something depends on either of these execution modes.

4.1.1. Pure data-flow VPLs

A pure data-flow VPL is a data-flow VPL with no added programming constructs. More precisely, we can give the following:

Definition 1. A *pure data-driven data-flow VPL* is a data-driven data-flow VPL which is made up only of visual elements representing functions and visual elements representing links (the arcs connecting the nodes). Moreover, a pure data-driven data-flow VPL is characterized by the following properties: (1) Data flows through links, which connect output data ports to input data ports, and is transformed by functions as it traverses them; (2) A function can have zero or more input data ports, to which its inputs are posted; (3) A function can have zero or more output data ports, through which its outputs are emitted; (4) A function can *fire*, i.e. can transform its inputs and emit its outputs, only if all its inputs are available; (5) A function fires as soon as all its inputs are available; (6) If a function has no input ports, then it fires once when the program starts; (7) When a function fires, it always emits some data for each one of its outputs; (8) Once a function has fired, for it to fire again it is necessary that it receives new input instances; (9) If two or more links are connected to the same output of a function, then, when the function fires, different identical data instances will flow through each link.

Definition 2. A *pure demand-driven data-flow VPL* is a demand-driven data-flow VPL for which all the properties of Definition 1 are satisfied, except for properties (5) and (6), which become the following: (5) A function fires when all its inputs are available and at least one of its outputs is required as an input by at least another function; (6) If a function has no output ports, then it requires its inputs only once when the program starts.

Moreover, to better clarify the demand-driven firing mechanism, another property is useful: (10) If at least one output of function A is required by at least another function B, then A requires its inputs from all the functions they come from.

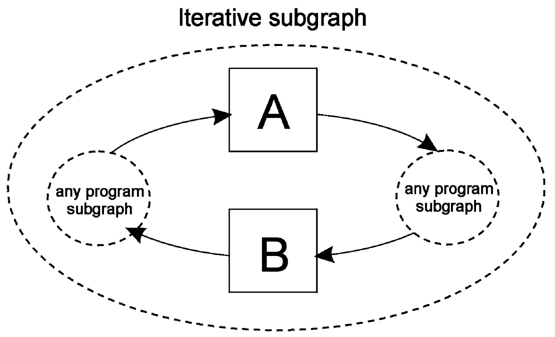


Fig. 15. A pure data-flow iterative subgraph.

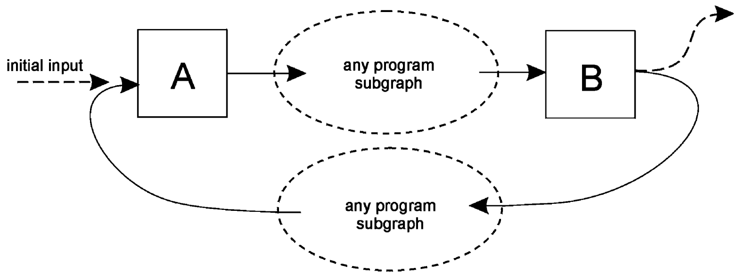


Fig. 16. Explanation for Theorems 1 and 2.

The important concept of *iteration* in pure data-flow VPLs can be formalized by the following:

Definition 3. A pure (data-driven or demand-driven) data-flow VPL subgraph is said to be *iterative* if at least one function A exists in the subgraph such that at least one of its inputs derives from an output of a function B for which, in turn, at least one input derives from an output of A (and, of course, vice versa).

From the previous definition, it is obvious that an iterative subgraph makes the program graph cyclic (Fig. 15).

Theorem 1. *In a pure data-driven data-flow VPL it is not possible to implement an iterative behavior unless at least one function in the looped subgraph is allowed to receive more than one link for the same input.*

Proof. Consider Fig. 16, which shows an iterative subgraph. Suppose that none of the functions within it is allowed to receive more than one link for the same input. Consider, for example, functions A and B. Since A’s input derives from B’s output and B’s input derives from A’s output, if A is not permitted to receive an initial input, it cannot ever fire, and this is true for any pair of

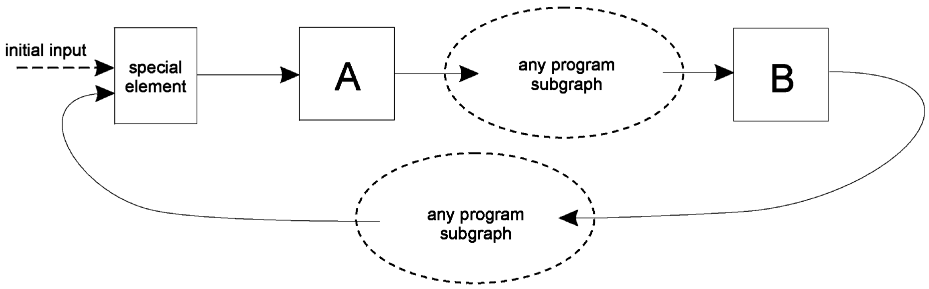


Fig. 17. Explanation for Corollary 1.

functions in the iterative subgraph. Now, suppose that, for example, A is permitted to receive an initial input. This will cause function A to fire, as well as, eventually, function B. Since A’s input derives from B’s output, A will fire again and the iteration process can go on.

It should be noted that, for property (4) of Definition 1, there must be an initial input value for each feedback link present in an iterative subgraph.

Theorem 2. *In a pure demand-driven data-flow VPL it is not possible to implement an iterative behavior unless at least one function in the looped subgraph is allowed to receive more than one link for the same input and at least one output of at least one function in the looped subgraph is an input for at least one function outside the subgraph.*

Proof. Still referring to Fig. 16, suppose that none of the functions within the iterative subgraph is allowed to receive more than one link for the same input. Since a necessary condition for a function to fire is that all its inputs are available, at least one function in the iterative subgraph must have an initial input (cf. demonstration for Theorem 1). But this is not sufficient. In the demand-driven execution model, a function can fire only if at least one other function requires at least one of its outputs. Suppose this is not satisfied and consider, for example, functions A and B, where A has an initial input. If its output were required, A could fire. However, since all the inputs of the functions within the iterative subgraph derives from A’s output, none of them can fire and therefore neither can A. This is true for any pair of functions. Now, suppose that, for example, B has an output link connected to a function outside the iterative subgraph. Then, each time B’s output is required, A, eventually can fire (at the first iteration step it consumes the initial input, while during the next ones it causes functions from which its input derives to fire).

Corollary 1. *If a pure data-driven or demand-driven data-flow VPL does not allow functions to receive more than one link for the same inputs, iterative behaviors can be obtained by introducing in the language a special element which has two or more inputs and which behaves the following way: it fires whenever one of its inputs is available, simply emitting it as an output* (Fig. 17).

According to Definitions 1 and 2, a data-flow VPL as described by the previous corollary is no longer pure. From now on, by “data-flow VPL” we will mean a pure data-driven or demand-driven

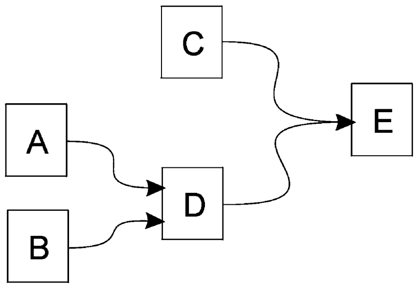


Fig. 18. Explanation for Corollary 2.

data-flow VPL to which some new properties have been added beyond the nine in Definition 1 or to the ten in Definition 2.

Corollary 2. *In a data-driven or demand-driven pure data-flow VPL, or in a data-driven or demand-driven data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *for a function to fire more than once it need not pertain to an iterative subgraph.*

In fact, by observing, for example, Fig. 18, it is evident that function E will fire twice (both with data coming from C and with data coming from D).

Theorem 3. *In a pure data-driven data-flow VPL, or in a data-driven data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *iterations are always endless.*

Proof. Consider, for example, Fig. 16. At the beginning, function A fires because it receives initial input data. Since a function always emits some data when it fires (property 7 of Definition 1), function B, which will also eventually receive input data, will fire and will produce output data. Since function A's input derives from function B's output, function A will fire again and the process will never end.

Theorem 4. *In a pure demand-driven data-flow VPL, or in a demand-driven data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *iterations always have an end.*

Proof. We will give an informal proof. Consider, for example, Fig. 19. For Theorem 2, an iteration can occur only if at least one output of at least one function in the iterative subgraph is an input for at least one function outside the subgraph. In Fig. 19, functions D and E satisfy this condition and therefore iteration is possible. Functions within the program subgraph can fire only if outside functions require their outputs.

These, in turn, can fire only if other functions require their outputs (or if they have no outputs at all). For the iterative subgraph to be executed endlessly, it would be necessary for at least one external function, for example function D or E in Fig. 19, to need its inputs an infinite number of

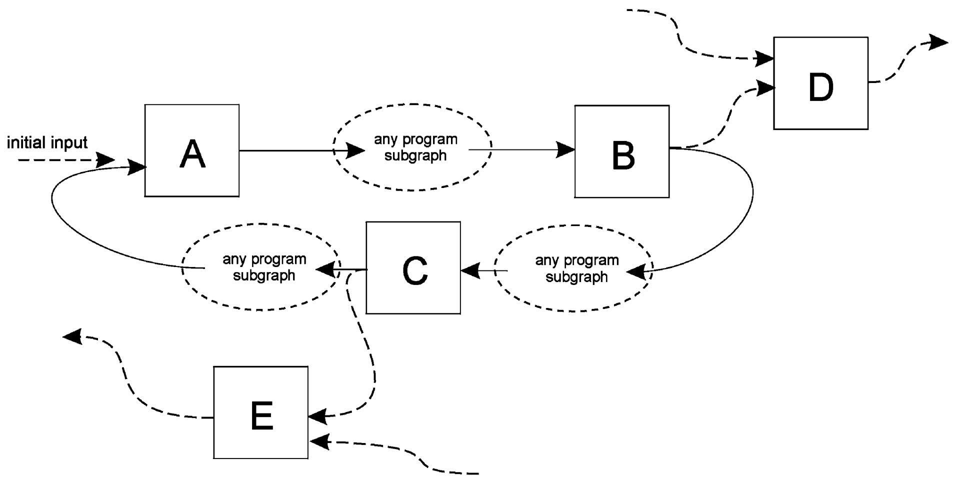


Fig. 19. Explanation for Theorem 4.

times. But this would mean that at least another function needs the output(s) of D or E an infinite number of times. And this would mean that there should be an infinite number of functions (think about Fig. 18). In fact, a finite number of functions placed within a cycle in the program graph cannot simulate such a behavior, because, as we are trying to prove, external functions behaving in the same manner would be required. Then, since a program cannot be composed of an infinite number of functions, iterations in demand-driven data-flow VPLs always have an end.

Corollary 3. *In a pure demand-driven data-flow VPL, or in a demand-driven data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *it is not guaranteed that all the functions within an iterative subgraph will fire even if some functions in the subgraph fire.*

This too is an informal proof for the corollary. Consider, for example, Fig. 19 and suppose that D and E require their inputs for the first time. This means that B (first) and C (second) have to fire. B's request is propagated to A, which can fire because it has an initial input. Then B fires, as well as, eventually C. But functions in the generic program subgraph between A and C do not fire during such a first-time execution. If, for example, E needs its input again, this event can occur, because A also needs input data.

Theorems 2 and 4 and Corollary 3 suggest the following important considerations. While the data-driven execution model allows iterations to be "self-sufficient", though endless, in the demand-driven execution model iterative behavior is always a consequence of some external events. If such events do not occur repeatedly, iteration cannot go on. Hence, in a certain sense, we may say that iteration does not pertain to the nature of the demand-driven execution model. And actually, among the five languages discussed in Section 3, only Cantata is demand-driven, which, as chance would have it, does not permit the implementation of real data-flow iterative structures. For all these reasons, *from now on in this paper, we will refer to data-driven data-flow VPLs only, even if not explicitly indicated.*

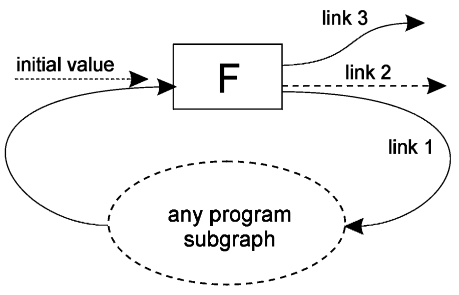


Fig. 20. Explanation for Corollaries 5 and 6.

Corollary 4. *In a pure data-flow VPL, or in a data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *an iteration can have an end if*: (1) *some mechanism is present which prevents at least one function in the iterated subgraph from firing if some conditions are satisfied*; *or* (2) *at least one function in the iterated subgraph is allowed not to emit all its output*, *if some conditions are satisfied when firing.*

It is clear that a data-flow VPL falling into one of the two above-mentioned cases cannot be pure. However, since our aim is to identify the minimum set of features which, added to a pure data-flow VPL, allow iterative behaviors to be achieved, the whole following discussion will try to rely, as much as possible, on the characteristics which make a data-flow VPL pure (Definition 1).

Although cases (1) and (2) of Corollary 4 might seem to be equivalent, in fact they are not. In case (2) a function with more than one output can selectively “decide” not to give out only some of its outputs (for example, only those forming cycles in the iterative subgraph). In case (1), on the contrary, a function which “decides” not to fire cannot give out any output at all. Therefore, case (1) is more restrictive than case (2).

Corollary 5. *In a data-flow VPL directly deriving from a pure data-flow VPL*, *or from a data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *a function which falls into case* (1) *of Corollary* 4 *cannot produce any output once the iteration process is finished.*

Consider Fig. 20, which shows an iterative subgraph. Suppose that function F is allowed not to fire when some condition C becomes true. Then, during the loop process (condition C is false), some data will flow through (the feedback) link 1 (and through any other possible links, such as links 2 and 3). As soon as C becomes true, however, function F will be forbidden to fire and therefore no data will be given out through any of its outputs (and no data will flow through either link 2 or 3).

Corollary 6. *In a data-flow VPL directly deriving from a pure data-flow VPL*, *or from a data-flow VPL not pure due to the only presence of the special element of Corollary* 1, *a function which falls into case* (2) *of Corollary* 4 *can still produce output data*, *once the iteration process is finished*, *only if it has more than one output.*

Consider Fig. 20 again. Suppose function F is allowed not to emit all its outputs, when firing, if some condition C becomes true. Then, when such an event occurs, function F will be able to

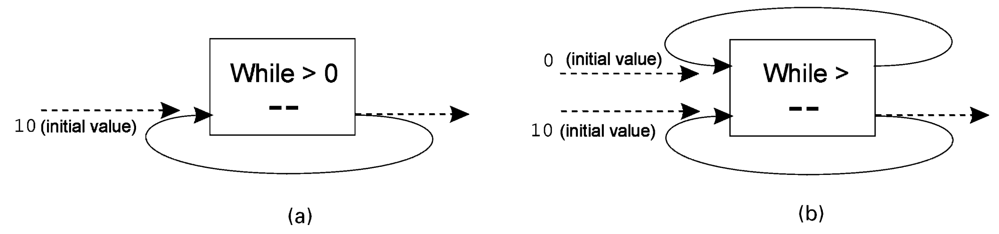


Fig. 21. Examples of iteration conditions of type (1), case (a).

prevent data from being emitted through its second output (to which links 1 and 2 are connected). If function F had not more outputs, no other data could be emitted. However, since it has another output, data can be emitted through it (and will flow through link 3).

Case (1) of Corollary 4 is more restrictive than case (2), and is also simpler and closer to the pure data-flow paradigm. Therefore, in the following discussion, we will consider only data-flow VPLs in which iterations are not endless for the only reason that some functions are able not to fire, although all their inputs are available, when some conditions are satisfied. We will call these conditions *iteration conditions.*

4.1.2. Types of iteration conditions

With reference to a function which is able to determine when to fire, iteration conditions can be of three types: (1) they may depend directly on at least one of the function's inputs; (2) they may depend on some inputs (or outputs) of some other function(s) in the program graph; or (3) they may depend on factors independent of any input (or output) of any function in the program graph. Since conditions of type (3) are very distant from the pure data-flow paradigm, they will not be considered any longer.

Iteration conditions of type (1). For iteration conditions of type (1), two sub-cases can be distinguished. Case (a): the function itself does incorporate the condition. Case (b): the function has a boolean input and decides whether to fire or not accordingly. Such an input receives data emitted by a function which is committed to evaluate the iteration condition.

Fig. 21a and b show simple iterative subgraphs, made up of one function only, which refer to case (a).

Functions in Fig. 21 keep on decreasing their input by one until the output reaches zero. Therefore, for an input '10', for example, the output data sequence will be: '10', '9',..., '1'. If we want the condition to depend on some constant input parameters as well, other feedback links must be introduced, and, precisely, one for each parameter. In fact, according to Property 8 of Definition 1, once a function has fired, for it to fire again it is necessary for it to receive new input instances. In Fig. 21b, the function has two inputs and the first one ('0') is the constant to be compared with the second to check the iteration condition.

Fig. 22a and b show the iterative subgraphs of Fig. 21 implemented according to case (b) of iteration conditions of type (1).

In both Figs. 22a and b, function '--' has two inputs. One receives the feedback data link, while the other receives the "condition feedback link", which is the output of function '> 0' or function '>'.

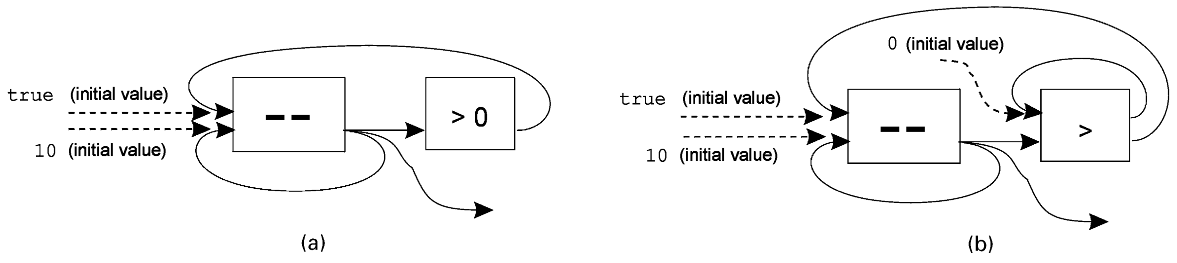


Fig. 22. Examples of iteration conditions of type (1), case (b).

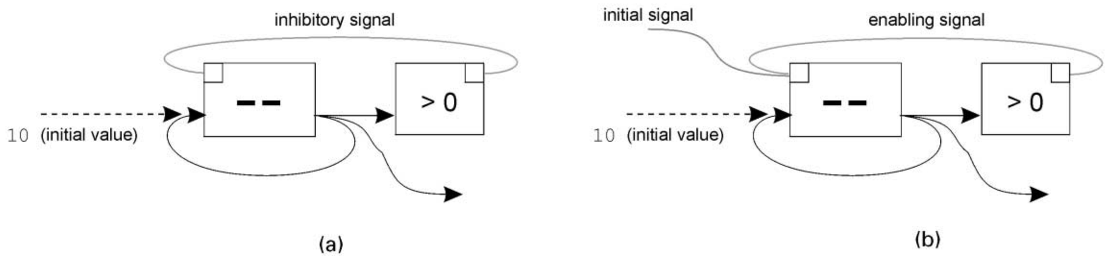


Fig. 23. Examples of visual control mechanisms.

Such functions evaluate their inputs to determine whether they are greater than zero and, if so, emit the boolean value ‘`true`’; otherwise, they emit the boolean value ‘`false`’. Function ‘`--`’ uses these values to decide whether to fire (‘`true`’) or not (‘`false`’), and, if it fires, the integer input is decreased by one.

Iteration conditions of type (2). For iteration conditions of type (2), there must be a mechanism which allows a function to determine whether to fire or not according to the inputs or outputs of another function (or of other functions) in the program graph. Practically, this must be a *control* mechanism through which a function can be indirectly informed about whether some condition has become true.

A control mechanism of this sort can be of three types: (a) visual; (b) not visual; or (c) hybrid (i.e. both visual and not visual). Types (b) and (c) assume, to a different extent, the ability of a function to incorporate implicit knowledge about other functions' inputs or outputs. Since such an ability is quite far from the visual nature which a visual language should imply, we will consider control mechanisms only of type (a).

A *visual control mechanism* must be characterized by some new special graphical element(s) in the data-flow language, besides functions and links. Such special element(s) may be of various kinds. As an example, consider Fig. 23a.

As in the examples of Fig. 22, function ‘`--`’ implements an iterative behavior and keeps on decreasing its input by one. In this case, however, it has an *input control port* as well (the square in the upper left corner) to which an *inhibitory signal* (a simple arc), coming from the *output control port* (the square in the upper right corner) of function ‘$>$ 0’, is connected. The meaning of such a visual arrangement, for example, may be the following: when function ‘$>$ 0’ fires and evaluates its

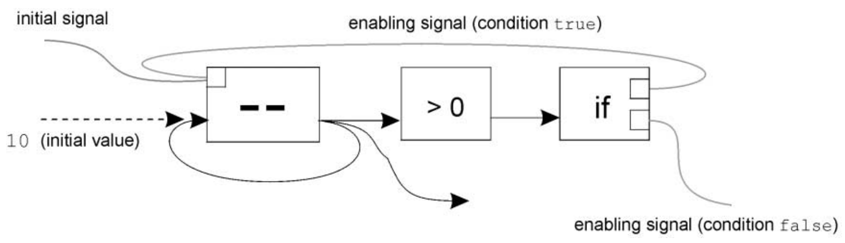

initial signal
enabling signal (condition true)
--
> 0
if
10 (initial value)
enabling signal (condition false)

Fig. 24. A visual control mechanism with an '`if`' element.

input to be zero, it *activates* its output signal, whose effect is to prevent function '`--`' from firing. In other words, the signal may be exploited as the mechanism to communicate to function '`--`' that a condition has become true (the input of function '`> 0`' has reached zero). An analogy with inhibitory arcs in Petri nets should be evident.

However, with such a control mechanism, there may be synchronization problems. In fact, consider again Fig. 23a. Since the output of function '`--`' is simultaneously posted both to its input and to the input of function '`> 0`', when it reaches zero, function '`--`' may fire before receiving the inhibitory signal.

A way to overcome this problem is to use *enabling signals* instead of inhibitory signals. In Fig. 23b, the signal is activated each time function '`> 0`', when firing, evaluates its input to be greater than zero. Function '`--`', on the other hand, fires only if it has received a new enabling signal instance. Hence, now, there is no longer any synchronization problem, although an initial signal is necessary to cause function '`--`' to fire the first time.

By comparing Fig. 23b to Fig. 22a, their similarity is clear. And actually, an enabling signal may also be seen, in a certain sense, as an ordinary link transporting boolean data, where '`true`' corresponds to signal activation. However, enabling signals can be much less restrictive than boolean input links, if they submit to the following useful rule: a function with an input control port to which no control signals are connected is allowed to fire. That is, signals have an effect only if they are present.

It is also interesting to note that if function '`> 0`' had another output control port, it could exploit it, for example, to enable another function once the iteration process is ended. And actually, it would be convenient for every *predicate*, such as function '`> 0`', to have a couple of output control ports, one to be activated when the condition is true and one to be activated when the condition is false. Equivalently, a special element with two output control ports could be used which, receiving a boolean input, activates either of them according to the input value. This way, a predicate could behave as an ordinary function emitting data (a boolean output). This, in turn, would be used as an input by the above-mentioned special element to activate the proper output signal. Fig. 24 shows the subgraph of Fig. 23b implemented through such a mechanism, where we call the special element '`if`'.

Although the two implementations of Figs. 24 and 23b are equivalent, the solution in Fig. 24 seems to be more general, because it allows predicates to be implemented as ordinary functions and delegates the management of signal activations to a single special element (element '`if`').

4.1.3. *Data-flow VPLs with enabling signals*

For the signal mechanism described in the previous section to be more general, it would be a good thing if every function had an input control port, so that any function can receive enabling signals and therefore be used within iterative subgraphs. Moreover, for synchronization purposes which are beyond the scope of this paper, if every function also had an output control port, control links connected to such a port could be used to exert a control over the execution of other functions. This would increase the gap between the data-flow VPL with respect to a pure data-flow VPL, but would also allow programs to be more easily managed and controlled. For all these reasons, it is worth giving the following:

Definition 4. A *data-flow VPL with enabling signals* is a pure data-flow VPL, or a data-flow VPL not pure due to the only presence of the special element of Corollary 1, to which the following properties have been added: (1) Each function has, besides possible input data ports, an input control port; (2) Each function has, besides possible output data ports, an output control port; (3) An enabling signal is a special link connecting an output control port of a function to the input control port of another function; (4) When a function fires, it activates all the possible enabling signals connected to its output control port; (5) If to the input control port of a function at least one enabling signal is connected, then the function can fire only if at least one of the signals has been activated; (6) Once a function has fired, all the possible enabling signals connected to its input control port are "reset", that is, for the function to fire again, it is necessary for at least one of the signals to be activated again; (7) If a function has no input data ports but at least one signal is connected to its input control port, then it fires each time at least one of the signals is activated; (8) Besides functions, a special element with two output control ports is present in the language, which, upon receiving a boolean input, activates either of them according to the input value.

4.1.4. *Iteration body*

So far, we have dealt with iterative subgraphs as abstract structures, without considering what an iteration is used for. As already stated, iteration means repeating several times a body of code, but a question comes to the fore: in practical terms, what really constitutes an iteration body in a data-flow VPL?

For an answer, consider, for example, Fig. 25.

It is obvious that functions '`--`' and '`> 0`', which are inside the iterative subgraph, are parts of the body. However, the output of these functions may be an input for another function within another program subgraph. Since such an output is emitted for each step of the iteration, the subgraph will also be executed the same number of times. Moreover, in a data-flow VPL with enabling signals, signals too can be used to activate functions within other program subgraphs. In Fig. 25, for example, the signal activated at each iteration step by the special element '`if`' may be exploited for such a purpose. These considerations lead to the following:

Definition 5. In a data-flow VPL with enabling signals, given an iterative subgraph, an *iteration body* is composed of: (1) The functions within the iterative subgraph; (2) The functions within possible program subgraphs whose inputs derive from the outputs of functions within the iterative subgraph; (3) The functions within possible program subgraphs which, directly or indirectly, are

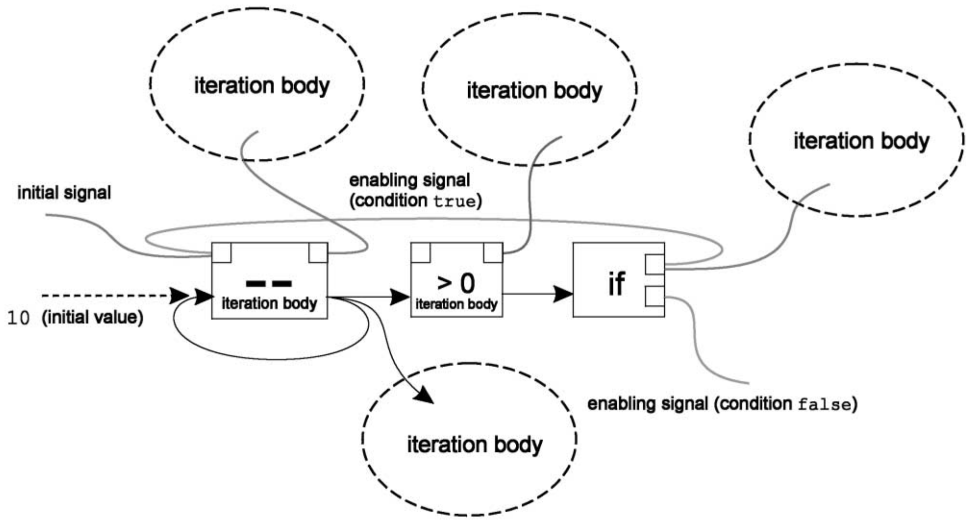


Fig. 25. Example of iteration bodies in a data-flow VPL with enabling signals.

activated by functions within the iterative subgraph or by the special element ‘`if`’ (which checks the iteration condition).

4.2. The VIPERS visual language

The language we will use to describe real data-flow implementations of various iteration constructs is VIPERS [7,8], a data-driven data-flow VPL with enabling signals developed at the University of Pavia.

VIPERS uses a single interpretive language (Tcl [16]) to define the elementary functional blocks (the nodes of the data-flow graph). Each block corresponds to a Tcl command (or procedure). According to the data-flow model, VIPERS programs are graphs in which data tokens travel along arcs between nodes (modules) which transform the data tokens themselves. VIPERS elementary modules have a square shape and present connection points, or ports, on their lateral sides; programs are assembled in a direct manipulation fashion, by positioning and properly connecting the available modules. Entire programs may therefore be built without typing any line of code: after having found in a certain window the functions needed (the nodes of the graph), represented by special icons, the programmer drags and drops them into the main workspace.

4.3. Data-flow iteration structures with enabling signals

In the following subsections, various real iteration structures, implemented through VIPERS, will be presented and analyzed.

4.3.1. The FOR iteration structure

Fig. 26 shows a FOR loop structure implemented through VIPERS.

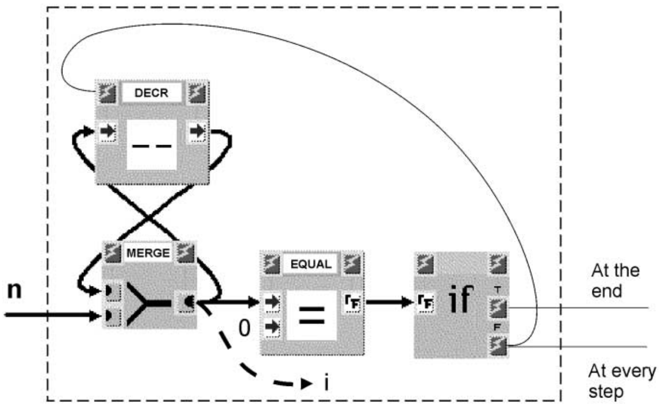


Fig. 26. A FOR loop structure implemented through VIPERS.

Each input/output port in VIPERS is characterized by a special icon indicating the corresponding data type. In our examples, the following symbols will be used: '[...]' for lists, '→' for integer numbers, 't' for textual strings and 'TF' for boolean values.

To achieve correct synchronization, VIPERS exploits enabling signals (thin arcs without arrows) connecting block control ports (those with the lightning symbol). In accordance with Definition 4 in Section 4.1.3, if a signal exists between an output control port (on the right) of block A and the input control port (on the left) of block B, then block B cannot be executed before block A. Signals are also useful for branching program execution through the IF special block (the special element of Property 8 in Definition 4), which, according to a boolean value it receives as an input, activates only one of its two output signals ('T' and 'F', for *true* and *false*).

Block MERGE, like the JCT object in VEE, is the special element of Corollary 1 in Section 4.1 and fires when either of its two input ports receives a new data item, then emitted as an output. Block MERGE, therefore, is suitable for use as the entry point of loop structures, thanks to its ability to accept both an initial input and successive updatings.

In Fig. 26, 'n' represents the number of times the subgraph activated by the *false* (F) control signal of block IF is to be executed. This number initially enters block MERGE, which simply emits it, and then is posted at the inputs both of block EQUAL and block DECR. Block EQUAL compares it with constant '0' and yields *true* or *false* according to the comparison result. The output of block DECR, whose function is to decrease its input value by one, is another input for block MERGE and represents the feedback link for the iteration. Practically, block DECR is activated 'n' times, for as long as the output of block MERGE is greater than '0'. Should it be necessary to have access to the loop index, data exiting block MERGE (i) can be used for such a purpose. The signal connecting block IF's *false* output control port with block DECR's input control port prevents the iteration process from continuing when the loop index has reached zero.

If the FOR loop index is required to go from one to 'n' instead of from 'n' down to one, the structure in Fig. 26 has to be modified a little. Practically, block DECR must be substituted with a block increasing its input by one and block EQUAL has to compare its first input with 'n' instead of zero.

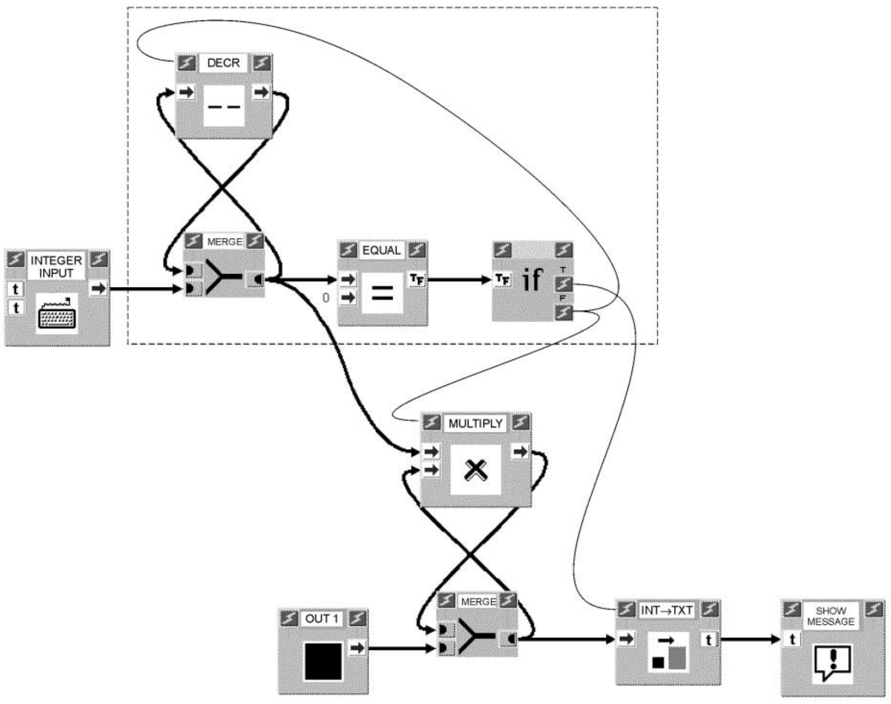
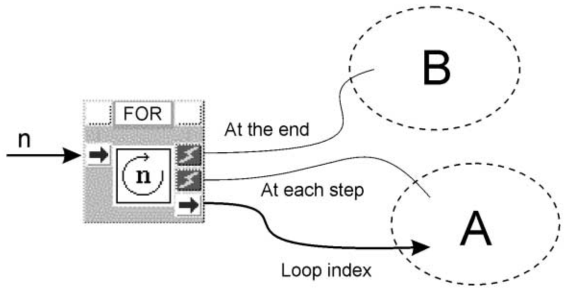


Fig. 27. Factorial calculus in VIPERS.

Fig. 27 shows a VIPERS solution to the factorial problem, in which the FOR structure of Fig. 26 is used. The corresponding textual (C) transposition is the same as that shown for Show and Tell in Section 3.1.

When the program execution starts, block INTEGER INPUT requests the user for an integer number and provides it to block MERGE, as an initial value for the iteration. Paralelly, block OUT 1 emits constant one, which is the initial value for the actual factorial calculation loop. Then, at each iteration step, the FOR loop index (data exiting the upper MERGE block) is supplied to block MULTIPLY, as a first input. The second input, instead, is the output of the lower MERGE block, which is the partial factorial. Block MULTIPLY can fire as long as block IF activates its *false* output control port, that is, as long as the (FOR) iteration condition (first input of block EQUAL equals zero) is not verified. When the condition becomes true, neither block DECR nor block MULTIPLY can fire any more. Then, block INT → TXT, activated by the signal coming from the *true* output control port of block IF, converts its input (the factorial just calculated) into a string, which, lastly, is displayed by block SHOW MESSAGE.

The analysis of the previous example easily leads to the following observation: in a data-flow VPL with enabling signals, sequential iteration implemented through the FOR structure always requires at least one more iterative subgraph to be present in the program graph besides that of the FOR structure itself. In fact, for data produced during an iteration step to be used during the next one, there must be at least one feedback loop to transport it. In the example of Fig. 27, such a loop is implemented by the link connecting the output of the lower MERGE block with the second input of block MULTIPLY.



```
for (i=n; i>0; i--) {
  A
}
B
```

Fig. 28. The “compressed” version for the FOR loop structure and its textual (C) transposition.

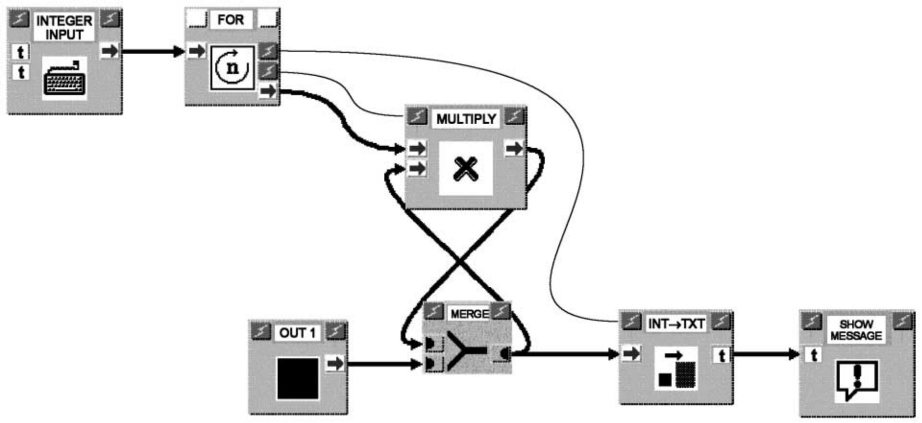


Fig. 29. Factorial calculus using a “compressed” FOR structure.

VIPERS allows groups of blocks to be embodied within another block. Then, it is interesting to note that, since its internal activities do not interfere with the loop body’s blocks, the whole FOR structure could be incorporated within a single block, as shown in Fig. 28.

A block of this type has an input, specifying the number of times the iteration body has to be executed, an output, through which the loop index is given out, and two output control ports, one activated at each step of the iteration and one at its end.

Fig. 29 shows the program of Fig. 27 implemented by means of such a “compressed” version of the FOR loop structure. By comparing it with the VEE version in Fig. 12 (Section 3.5.1), their similarity is obvious.

Indeed, the *sequence out pin* and the *sequence in pin* of VEE objects appear similar to the output control port(s) and the input control port of VIPERS blocks. However, there are two main differences between the two systems. In the first place, a VEE *sequence in pin* cannot receive more than one activation link, while VIPERS allows more than one signal to be connected to the same input control port. Secondly, VEE allows cycles to be created within the program graph only if a *For Count* or an *Until Break* object is involved, prohibiting, for example, an explicit implementation such as in Fig. 27. These restrictions, probably, have the purpose of limiting graph complexity.

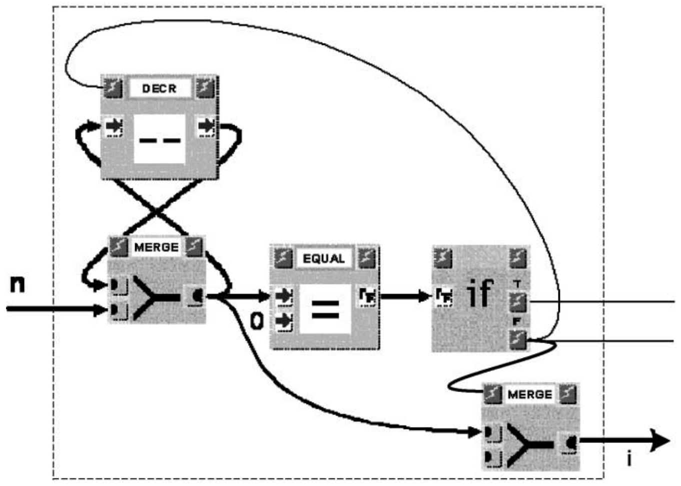


Fig. 30. The FOR structure with controlled index.

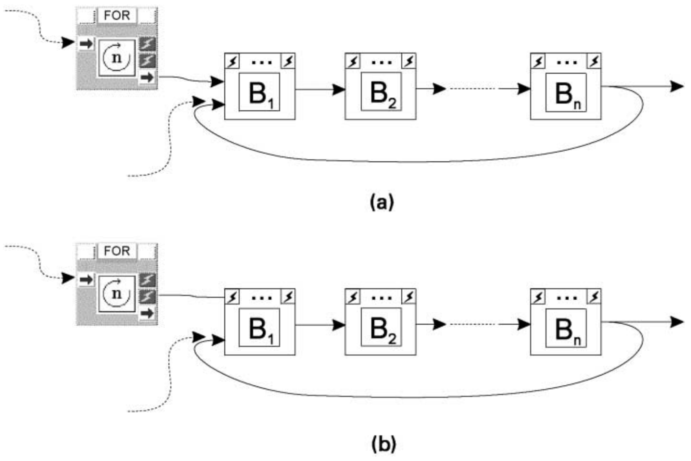


Fig. 31. Potential synchronization problems with the FOR structure.

Observing Figs. 27 and 29, a question may arise: is the signal connecting either the *false* output control port of block IF (Fig. 27) or block FOR (Fig. 29) to the input control port of block MULTIPLY really necessary? The answer is yes, because of synchronization problems. Let us suppose there is no such signal. Then, block MULTIPLY will correctly fire for as long as the loop index is greater than zero. However, when zero is reached, this block must not fire, because otherwise, the factorial just calculated would be multiplied by zero and consequently the computation result would be lost. A solution to this problem may consist in introducing a control over the output of the MERGE block within the FOR structure, for example, through another MERGE block activated by a signal connected to the *false* output control port of block IF, as shown in Fig. 30.

Another important synchronization issue has to be taken into account when using the FOR iterative structure. Consider, for example, Fig. 31a, in which the loop index is provided, as an input, to block B_1, also receiving a feedback input.

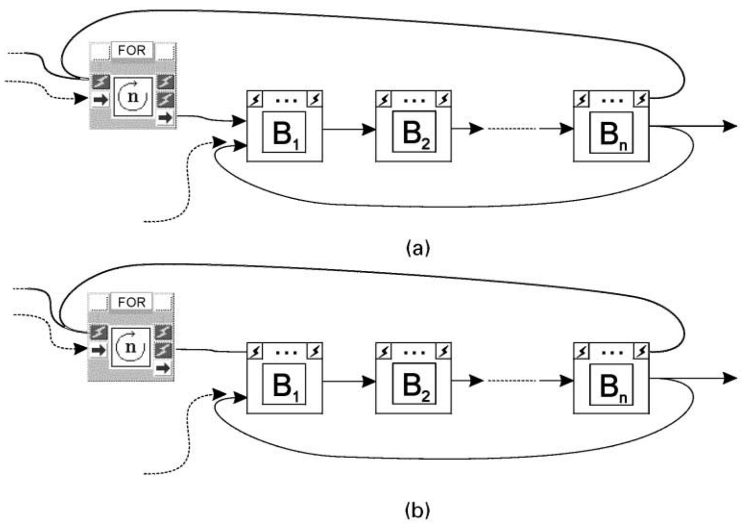


Fig. 32. Solution to the synchronization problems of Fig. 31.

Observing Fig. 26, it can be noted that the FOR structure emits the loop index every two “firing steps”. That is, after initial data (‘`n`’) has been provided to and emitted by block MERGE, it produces the next output (and therefore fires) after block DECR has fired. Since the subgraph in Fig. 31a is composed of more than two blocks, the FOR structure will yield a new value for the loop index before block B_n has consumed the previous one, therefore causing some indexes to be lost. Analogous considerations are valid if block B_1 does not receive the loop index but is subordinated to the FOR control signal activated at each iteration step. In this case too, there will be a loss of signal activations (Fig. 31b).

To prevent this kind of problems from occurring, block FOR must have an input control port as well, so that signals can be used to control it. Fig. 32 shows this new solution for the sample programs of Fig. 31. Now, in both Figs. 32a and b, the iteration process can go on to the next step only after block B_n has fired and synchronization problems no longer exist (although an “initial” signal is necessary to allow the first step of the iteration to occur). Practically, the input control port of block FOR corresponds to the input control port of block MERGE in the explicit structure of Fig. 26.

4.3.2. The FOREACH iteration structure

A typical case of parallel iteration is sequential access to all the elements of a data structure, so that certain operations can be performed on them. Fig. 33 shows the explicit structure for a FOREACH construct, which, given a list (L), sequentially emits its elements (E).

In this example, the list contains strings. The underlying mechanism being conceptually the same, the list could be replaced with any other sequential data structure, such as, for example, a file.

As can be seen from the figure, the initial list enters block MERGE, which leaves it unchanged, then is both posted at the input of block HEAD and analyzed by the boolean block NULL. This block verifies whether the list’s length is zero (in which case it gives out ‘`true`’) or greater than zero (in which case it gives out ‘`false`’).

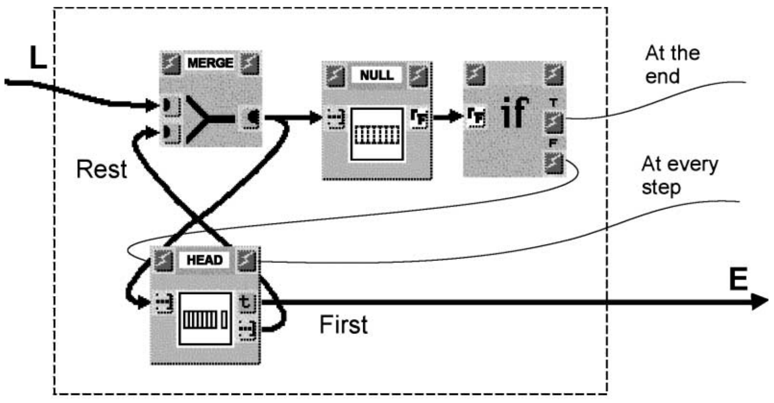


Fig. 33. Implementation of the FOREACH structure in VIPERS.

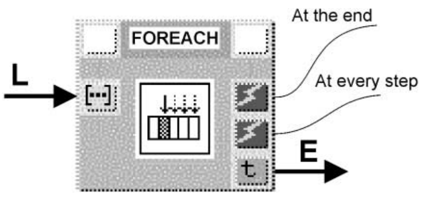


Fig. 34. Compact form for the FOREACH iterative structure.

If the list is not null, block IF activates the signal relative to its *false* output control port, thus enabling block HEAD. This block separates the input list's first element (First), made available, from the remainder (Rest). The "beheaded" list then re-enters block MERGE and this process is repeated until all the list's elements have been scanned.

The signal emitted by block HEAD every time it is activated can be employed to create a synchronism with the execution of any other blocks which do not directly need data contained in the list being inspected. Likewise, when the list traversal is finished, the *true* output control signal of block IF can be utilized to activate new blocks and hence allow the computational process to go on. In the implementation of the FOREACH construct of Fig. 33 the graph is made cyclic by the arc going from the output of block HEAD (Rest) to the input of block MERGE.

If one prefers to deal with a more compact structure, the whole dotted portion of the figure can be embodied into a single block, as shown in Fig. 34. Such a block has the list to be scanned as input and the current element as output, besides the control signals.

As an example of how the FOREACH structure can be used, Fig. 35 shows a program calculating the sum of all the numbers contained in a list. Block OUT 0 emits constant zero once the program starts. Block SHOW INTEGER displays the result when the calculation is finished.

Like in the case of the FOR iterative structure, the FOREACH construct also requires at least one more iterative subgraph to be present in the program graph besides that of the FOREACH structure itself.

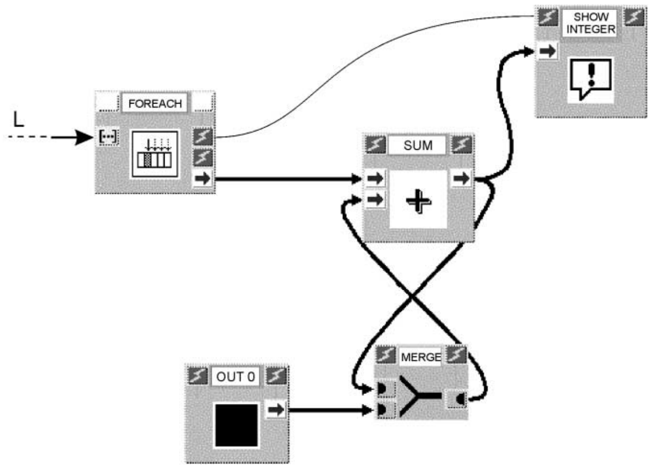
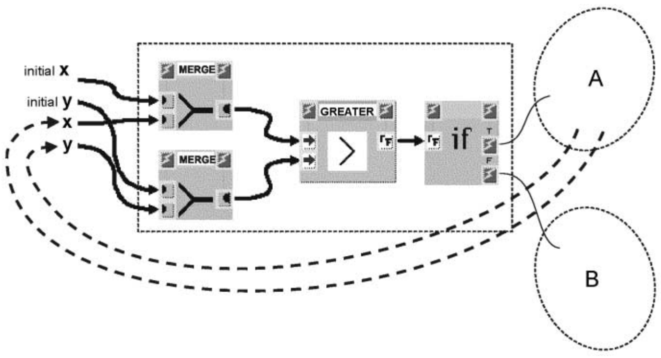


Fig. 35. A VIPERS program using the FOREACH iterative structure.



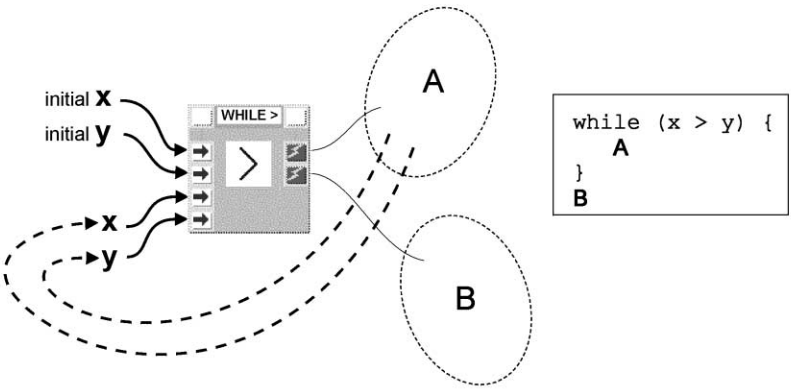
```
while (x > y) {
    A
}
B
```

Fig. 36. Example scheme for an explicit WHILE construct in VIPERS and corresponding textual (C) transposition.

Moreover, synchronization problems analogous to those already discussed for the FOR loop in Section 4.3.1 may arise when the output elements (E) are inputs for a block which also receives a feedback input or when such a block is controlled by the FOREACH output control port activated at each iteration step (see Figs. 31 and 32). In this case too, the solution consists in providing the FOREACH block with an input control port, so that signals can be used to command it. Practically, this port corresponds to the input control port of block MERGE in Fig. 33.

4.3.3. The WHILE iteration structure

In WHILE constructs, execution of the loop body depends on one or more boolean conditions which are to be satisfied. In Fig. 36 the implementation of a WHILE structure depending on two integer "variables" x and y is shown. The variables are used by the test block GREATER and the iteration body (subgraph A) is executed as long as $x > y$. The two MERGE blocks are necessary to allow initial values to be provided for the variables. As in the case of the FOR and FOREACH



```
while (x > y) {
    A
}
B
```

Fig. 37. Compact form for the WHILE scheme of Fig. 36.

constructs, the repeated subgraph is allowed to contain cycles, to carry out temporally dependent iterations.

Compacting the WHILE structure (Fig. 37) does not eliminate any cycle and generates a control block which, by depending directly on the loop condition, appears less general, though equally expressive, compared with those discussed previously.

In other words, while for the FOR loop, and the FOREACH construct for a specific data structure, it is possible to use a standard scheme in any situation, the WHILE loop depends on the particular condition to be tested to decide whether to continue with the iteration or to stop.

4.4. Implementing the concept of 'Variable' through iterations in a data-flow VPL with enabling signals

Iterative subgraphs can also be used to implement the concept of variable in a data-flow VPL with enabling signals such as VIPERS.

A variable, essentially, is "something" to which values can be assigned (stored in) and from which values can be read. Fig. 38 shows a possible structure for a variable in VIPERS.

Since only one of its inputs is used, the lower MERGE block could be replaced with a block which simply emits its input as an output (actually, this block is only necessary because VIPERS does not allow a block's output to be connected directly to one of its inputs; if there was no such restriction, the 'Data out' link in Fig. 38 could be directly joined to the second input port of the upper MERGE block and there would not be any need for the lower one).

The cyclic structure of the variable construct ensures values are not lost once they have been read. A value is automatically assigned to the variable when it is posted to its input (link 'Data to be stored' in Fig. 38). To read the variable's current value, instead, a signal ('Read signal' in the figure) must be activated, so that an output can be given out. However, an important point should be made here: for the variable mechanism to function correctly, it must be assumed that when a MERGE block fires because a signal connected to its input control port is activated, it always chooses the newest input between the two which may be available. Since each time the variable is read its value is re-posted to the upper MERGE block, if this was not verified, a new value to be written, posted after a read operation, would not be taken into account.

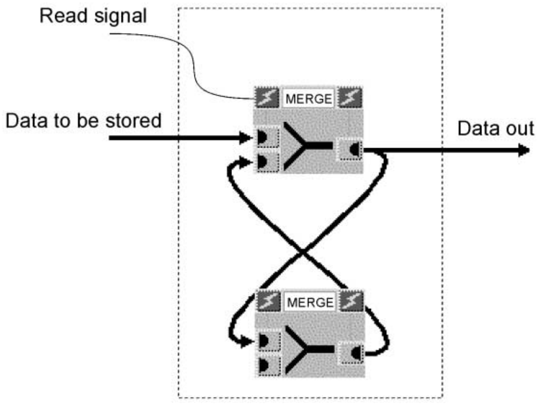


Fig. 38. Possible structure for a variable in VIPERS.

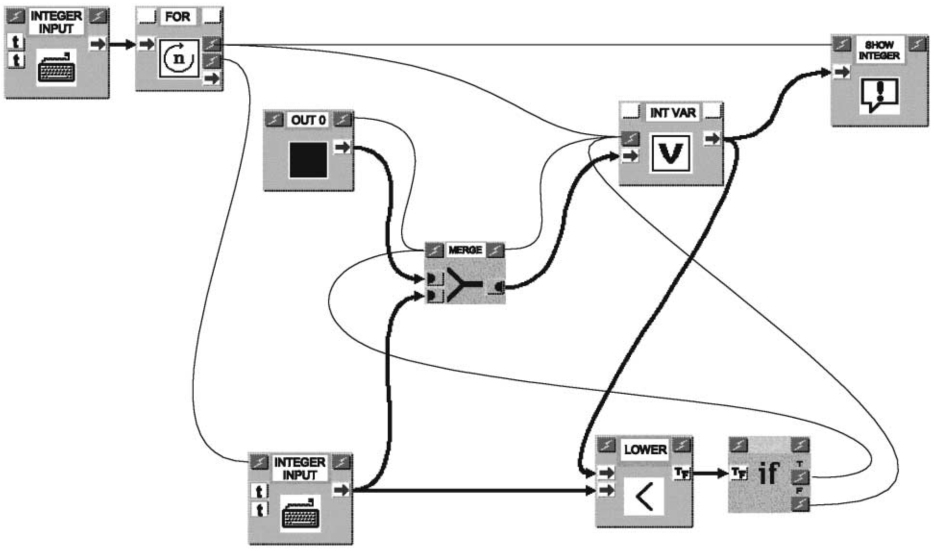


Fig. 39. An example of a VIPERS program using a variable.

Fig. 39 shows an example program using a compact form for the variable structure (block INT VAR embodies the whole dotted portion of Fig. 38).

The program asks the user for an integer number ‘n’ times, ‘n’ being specified through the upper INTEGER INPUT block, and, lastly, displays the maximum value inserted. Clearly, such an operation requires a form of memory to store the partial maximums. When the program starts, block OUT 0 emits constant zero and enables block MERGE, which, in turn, stores the constant in the variable (block INT VAR). Block MERGE, through its output signal, also causes the variable’s value to be made available (i.e. to be read), so that it can be used by block LOWER. For each step of the FOR iteration, the lower INTEGER INPUT block is activated and asks the user for a number, which is posted to both block MERGE and block LOWER. The block then compares this value to the one (the current maximum) just read from the variable and, according to whether it is greater or lower, emits *true* or *false*. If *true* is emitted (which means that the number inserted by the user is greater than the current maximum), block MERGE is enabled (block IF's *true* path selected) and the new value is written into the variable (and then read). Otherwise (block IF's *false* path selected), the old value is simply read. Therefore, at each FOR iteration step, comparison is actually made with the current maximum. When the iteration is finished, a last 'read' operation for the variable occurs and the program result is displayed by block SHOW INTEGER.

5. Conclusions

Among the various classes of visual programming languages (VPLs), data-flow ones have shown a very high diffusion, certainly owing to their simple and intuitive functioning mechanism.

In particular, since the concept of data being transformed by functions is common to many scientific and engineering areas, the data-flow paradigm turns out to be well suited to the nature of many kinds of problems. Yet, often, a (visual) programming language is useful only if it provides the necessary programming constructs to deal with complex problems (within the language's application domain).

Iteration is undoubtedly an important aspect of programming. However, though simple problems may be solved through *pure* data-flow VPLs, in general, ways to facilitate specification of iterative behaviors are needed. Most existing data-flow VPLs adopt a variety of methods to overcome the limitations of the pure data-flow model in implementing iterations. These methods, sometimes, rely on paradigms which are far from data-flow.

In the first part of this paper, we surveyed the ways iterations are carried out by some representative data-flow VPLs. Then, in the second part, we semi-formally investigated what the minimum set of characteristics should be, which, added to a pure data-flow VPL, allow iterative behaviors to be implemented. Lastly, in the third part, we presented some real iterative constructs relying on concepts coming out in the second part and implemented through the VIPERS visual language.

The discussion in this paper, however, *was not aimed at demonstrating that such data-flow solutions are better than solutions adopted by other data-flow VPLs* (in particular, those presented in the survey). It has only wanted to explore whether, how and when (i.e. under what conditions) iterations can be more explicitly represented in a pure data-flow VPL.

References

[1] Larkin JH, Simon HA. Why a diagram is (sometimes) worth ten thousand words. Cognitive Science 1987;11(1): 65–99.

[2] Hils DD. Visual languages and computing survey: data-flow visual programming languages. Journal of Visual Languages and Computing 1992;3:69–101.

[3] Vose GM, Williams G. Labview: laboratory virtual instrument engineering workbench. Byte 1986;11:84–92.

[4] National Instruments Corporation. Labview. User Manual, 1992.

[5] Helsel R. Cutting your test development time with HP VEE, HP professional books. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[6] Hewlett-Packard Co. VEE, Version 5.01. Online help manual, 1998.

[7] Bernini M, Mosconi M. VIPERS: A data-flow visual programming environment based on the Tcl language. Proceedings of the second Conference on Advanced Visual Interfaces (AVI'94), 1994.
[8] Ghittori E, Mosconi M, Porta, M. Designing new programming constructs in a data-flow VL. Proceedings of the 14th IEEE Conference on Visual Languages (VL'98), Nova Scotia, Canada, 1–4 September 1998.
[9] Ambler AL, Burnett MM. Visual forms of iteration that preserve single assignment. Journal of Visual Languages and Computing 1990;1:159–81.
[10] Schürr A. BDL—A nondeterministic data-flow programming language with backtracking. Proceedings of the 13th IEEE Conference on Visual Languages (VL'97), Capri, Italy, 1997.
[11] Kimura TD. A visual language for keyboardless programming. Technical Report WUCS-86-6, 1986, Department of Computer Science, Washington University, St. Louis, Missouri 63130.
[12] Cox PT, Giles FR, Pietrzykowski T. PROGRAPH: A step towards liberating programming from textual conditioning. Proceedings of the IEEE Workshop on Visual Languages (VL'89), Rome, Italy, 4–6 October 1989, p. 150–6.
[13] Pictorius, Inc. Prograph for windows, Version 1.2. Online help manual, 1998.
[14] Rasure J, Williams CS. An integrated data-flow language and software development environment. Journal of Visual Languages and Computing 1991;2:217–46.
[15] Khoral Research, Inc. Visual programming: the Cantata user's guide. Khoros manuals, online web manual, 1997.
[16] Ousterhout J. Tcl and the Tk Toolkit. Reading, MA: Addison-Wesley, 1994.

Mauro Mosconi obtained his Ph.D. degree in Electronic and Computer Engineering from the University of Pavia (Italy) in 1993. He is currently a member of the faculty of Engineering at the same University. His research interests include visual interfaces, usability and Web applications.

Marco Porta obtained his Ph.D. degree in Electronic and Computer Engineering from the University of Pavia (Italy) in 1999. At the same University, he is currently a post-doctoral researcher. His interests include visual languages, human-computer interfaces and machine vision.