Designing new Programming Constructs in a Data Flow VL

Elena Ghittori - Mauro Mosconi - Marco Porta

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

mauro@vision.unipv.it, porta@vision.unipv.it

Abstract

A powerful and useful Data-Flow Visual Programming Language (DFVPL) must provide the necessary programming constructs to deal with complex problems. The main purpose of this paper is to give a contribution to the debate on DFVPL constructs, by presenting the solutions we devised for the VIPERS language.

1. Introduction

Data-flow is one of the most popular computational models for visual programming languages (VPL). One of the most important features which characterizes the power of a data-flow VPL is the availability of a rich library of predefined functions to be used as elementary building blocks [1]. Moreover, a powerful and useful data-flow VPL must provide the necessary programming constructs to deal with complex problems (in the language's application domain). Our experience with the VIPERS system [2], developed at the University of Pavia, confirms once more that the pure data-flow model needs to be enriched with some forms of control flow constructs in order to tackle non-trivial applications. Iteration, for instance, has been provided in different ways in several languages. Nevertheless, we feel that satisfactory solutions are very difficult to achieve: sometimes, these solutions use a notation which is not consistent with the data-flow paradigm. The purpose of this paper is to give a contribution to the debate on data-flow VPL constructs.

2. Loop Control Structures in VIPERS

VIPERS allows the programmer to freely choose whether to explicitly construct parallel iterations [3] , by introducing cycles into the program graph, or to use compact forms simulating loop behaviors.

Usability issues regarding the loop structures which will be discussed in the next subparagraphs can be found in [4], where a new testing methodology is also presented.

2.1 The FOREACH control 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. Figure 1 shows the explicit structure of a *Foreach* construct, which, given a generic list (L), sequentially emits its elements (E). Input ports are on the left side, while output ports are on the right side. To achieve a correct synchronization, VIPERS exploits control signals (thin arcs without arrows) connecting blocks' control ports (those with the lightning symbol). If there exists a signal 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 can not be executed before execution of block A. More than one signal may arrive at the same input port: for the correspondent block to be enabled, it is sufficient that at least one of them is active.

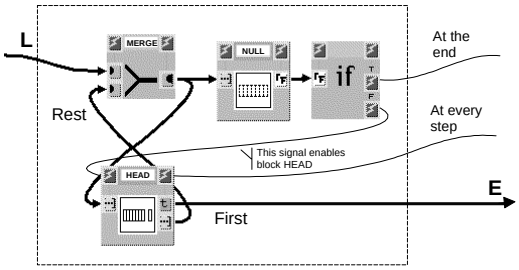

Figure 1: implementation of the *Foreach* structure in VIPERS

The MERGE block fires when either of its two input ports receives a new data item, which is then emitted as an output. Such block is mostly used as the entry point of loop structures, thanks to its ability to accept both an initial input value and successive updatings of it.

As can be easily seen from the figure, the initial list enters block MERGE, leaving 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 gives "true" off) or greater than zero (in which case gives "false" off). If the list is not null, block IF activates the signal relative to its "false" output control port (F), 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 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 execution of possible other blocks which do not directly need data contained in the list being inspected. Likewise, when the list traversal is finished, the "true" control signal (T) given off by block IF can be utilized to activate new blocks and hence allow the computational process to go on.

If one prefers to deal with a more compact structure, the whole dotted portion of the figure can be embodied into a single block (a library block or a macro) [2], as shown in

Figure 2. Such block has as input the list to be scanned and as output the current element, besides the control signals.

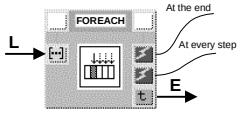

Figure 2: compact form for the *Foreach* control structure

2.2 The FOR control structure

The situation for the *For* loop construct, whose explicit and compact forms are shown in Figure 3, is analogous. n represents the number of times the subgraph activated by the "false" (F) control signal of block IF is to be executed. Block DECR, enabled as long as the iteration process is not completed, decrements its input data by one and posts them to block MERGE, out of which they arrive unchanged. Block EQUAL compares such data with zero and emits "true" or "false" according to the comparison result. It is to be noted that the subgraph (the loop body) controlled by the *For* iterative structure (or, in the previous case, by the *Foreach* structure) may in turn contain cycles, thus implementing a temporally dependent iteration [3] (Figure 4). In this case, however, it might be necessary for block DECR (Figure 3) to be activated by a block inside the loop body instead of being enabled by the "false" control signal of block IF, in order to achieve correct synchronization. In fact, it may happen that execution of complex and long loop bodies has still not finished when a new activation signal is emitted by block FOR. When it is necessary to wait for the end of an iteration step before starting with the next one, a synchronizing signal must be added to connect the end of the cycle with the FOR block. For this reason, the compact *For* structure, in the case of sequential iteration, may also require an input control port. Similar considerations are valid for the *Foreach* structure.

2.3 The WHILE control structure

In *While* constructs, execution of the loop body depends on one or more boolean conditions which are to be satisfied. In Figure 5 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 body is executed as long as x > y. As in the case of the *For* and *Foreach* constructs, the repeated subgraph is allowed to contain cycles, to carry out temporally dependent iterations. Figure 6 shows the compact form for the *While* scheme of Figure 5.

References

[1] Hils, D. D., "Visual Languages and Computing Survey: Data Flow Visual Programming Languages", *Journal of Visual Languages and Computing*, vol. 3, 1992, pp. 69-101.

[2] Bernini, M., Mosconi, M. "Vipers: a data Flow Visual Programming Environment Based on the Tcl Language", in *Proc.AVI'94*, ACM Press , 1994

[3] Ambler, A. L., Burnett, M. M., "Visual Forms of Iteration that Preserve Single Assignment", Journal of Visual Languages and Computing, vol. 1, 1990, pp. 159-181.

[4] Ghittori, E., Mosconi, M., Porta, M., "Designing and Testing new Programming Constructs in a Data-Flow VL", Technical Report, DIS University of Pavia, Italy, 1998. URL: http: //iride.unipv.it/research/papers/98tr-dataflow.html.

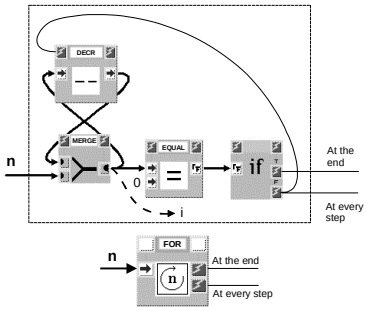

Figure 3: explicit (above) and compact (below) forms for the *For* iterative construct in VIPERS

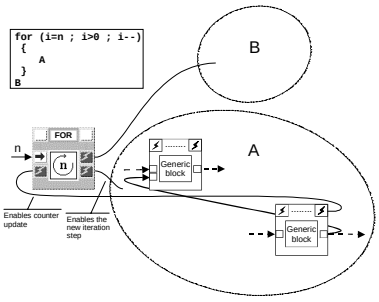

Figure 4: example scheme for a *For*-based temporally dependent iteration in VIPERS

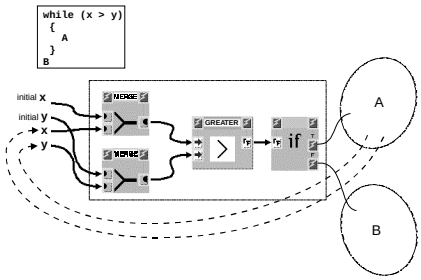

Figure 5: example scheme for an explicit *While* construct in VIPERS

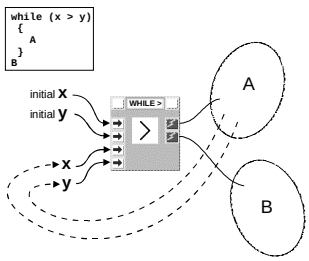

Figure 6: compact form for the *While* scheme of figure 5