# MOTION-PICTURE DEBUGGING IN A DATAFLOW LANGUAGE

Stanislaw Matwin
University of Ottawa
Ottawa, Ontario

Tomasz Pietrzykowski
Acadia University
Wolfville, Nova Scotia

ABSTRACT

The paper describes how graphics are used in an experimentral programming language PROGRAPH. Particular emphasis is on application of simple graphics for debugging of highly concurrent and distributed programs. The system described in the paper has been implemented.

RESUME

Le mémoire décrit comment les graphiques sont utilisés dans un langage expérimental, PROGRAPH. On présente ici des applications de graphiques simples pour le debugging des programmes hautement concurrents et distribués. Le système présenté dans le mémoire a été réalisé sur l'ordinateur.

## Summary

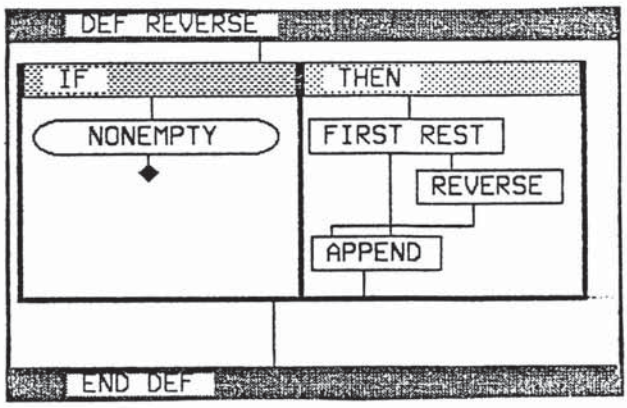
Programming languages for non-von Neumann computers are gaining more and more interest among researchers in Computer Science. Nevertheless, in order to be recognized as being useful in non-academic environment, two problems related to their use will have to be solved. First problem is notation.

Two approaches exist here.

On one hand, the natural text representation of the program is being used [VAL]. However, because of the linear character of the text, there is a danger that the programmer may unconsciously eliminate certain concepts, like parallelism, from his solution. Therefore, an alternative notation, based on dataflow programs, has been advocated by the group from the University of Utah [GPL]. Our proposal for the dataflow programming language [PROGRAPH] goes along their ideas. Although PROGRAPH and GPL have a number of features in common, PROGRAPH introduces a number of concepts which are, to our knowledge, novel in functional languages (e.g. database apparatus, and tools for explicit synchronization of parallelism).

On the other hand, in order to become practical tools, dataflow programming languages require an effective programming environemnt. Two important components of such an environment are: a graphics editor to create and modify program graphs ("prographs"), and an adequate debugger. In this paper we identify some problems stemming from the design of such a debugger and present our solution of these problems.

Our discussion of these features will be illustrated on an example of a simple prograph definition, which reverses a list submitted as its only argument. This definition, as given below, contains a bug. The reader should be able to identify it easily without being fluent in PROGRAPH, since PROGRAPH notation is highly intuitive and clear to anyone with basic training in programming:



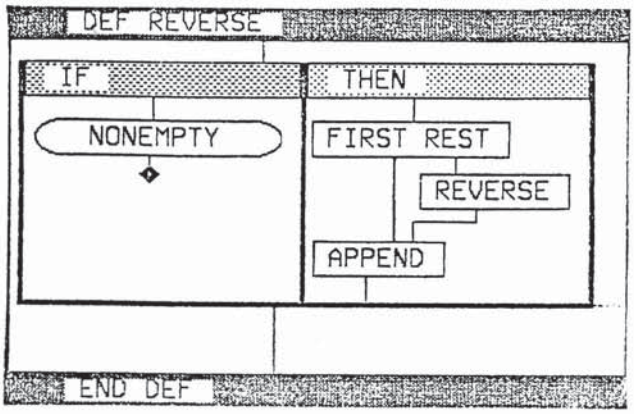
When execution of this user-defined operation starts, the user wants to follow the flow of data through the "compartments" of IF-THEN complex. The next request from the user is to be able to see the actual values flowing through the wires. Moreover, the user wants to see in what order the boxes are "armed". Arming a box means satisfying all its input; a box is then fired, its output flow through the wires to other boxes, which in turn become armed, etc.

Finally, the user wants to follow the recursion and be able to do it in a stop-and-go mode. This gives him a chance to compare the events taking place in his prograph with his original intentions.

The debugging mechanism of PROGRAPH meets all the demands described above. First of all, when the IF-THEN complex is entered and the logical compartment with IF at its top is executed, the top bar starts flashing, identifying the currently executing compartment. The fact that the data flows through the wire is represented by tokens, moving on the screen along the wire (this explains the "motion-picture" name coined for our approach). Furthermore, the user has the option of seeing the actual values flowing through the wires (this is possible only for atomic values, i.e. numbers and characters). If, during the prograph execution process, a user-definition is called, then the prograph of this definition is displayed. Execution of this prograph is then visually traced using the methods described above. This mechanism obviously allows for tracing of recursive calls.

Returning to our example of REVERSE, suppose that it is tested on the following list:

(1, 2, 3)




The user will have a chance to observe three recursive calls. As a matter of fact, he will be able to spot the error with the first call, when FIRST REST will send 1 as the first argument of APPEND, which means that the result of REVERSE will be a list starting with value 1. If not spotted at this level, the error will persist in the next recursive call, etc. This should allow the user to understand that the problem is in the ordering of arguments of APPEND. A quick fix with the editor gives the preceding, correct prograph.

In conclusion, it may be noted that the proposed method, besides being feasible (it has been implemented on a PERQ) is also aesthetically attractive. Moreover, it seems to have the potential to be used as a tool to debug distributed programs [Basili].

References

[Basili] Basili, V., Private communication, 1983

[GPL] GPL Programming Manual, Research Report, Department of Computer Science, University of Utah, July, 1981.

[PROGRAPH] Pietrzykowski, T., Matwin, S., Muldner, T., Programming Language PROGRAPH - Yet Another Application of Graphics, Procs, of Graphics Interface' 83, Edmonton, Alta, pp. 143-145.

[VAL] McGraw, J., The VAL Language: Description and Analysis, ACM TOPLAS, Jan. 1982, pp. 44-82.