ACTIVE TEMPLATES: MANIPULATING POINTERS WITH PICTURES

P.J. Lyons
(P.Lyons@massey.ac.nz)
Computer Science Dept.
Massey University
Private Bag 11-222
Palmerston North
New Zealand

M.D. Apperley
(M.Apperley@waikato.ac.nz)
Computer Science Dept
Waikato University
Private Bag 3105
Hamilton
New Zealand

A.G. Bishop
(A.G.Bishop@massey.ac.nz)
Computing Services
Massey University
Private Bag 11-222
Palmerston North
New Zealand

G.S. Moretti
(G.Moretti@massey.ac.nz)
Computer Science Dept.
Massey University
Private Bag 11-222
Palmerston North
New Zealand

ABSTRACT

Active templates are a semi-automatic visual mechanism for generating algorithms for manipulating pointer-based data structures. The programmer creates a picture showing the affected part of a data structure before and after a general-case manipulation. Code for the operation is compiled directly from the picture, which also provides the development environment with enough information to generate, automatically, a series of templates for other similar pictures, each describing a different configuration which the data structure may possess. The programmer completes the algorithm by creating matching after-pictures for each of these cases.

At every stage, most of the picture-generation is automatic. Much of the tedious detail of conventional pointer-based data-structure manipulation, such as maintenance of current pointers, is unnecessary in a system based on active templates.

KEYWORDS

Active templates, visual programming language, HyperPascal, data structures

INTRODUCTION

Active templates are the interface to a graphical representation for algorithms to manipulate pointer-based data structures, and to a system for generating the algorithms semi-automatically. The algorithms comprise a set of "before-pictures" of possible configurations of the data structure and corresponding "after-pictures" showing them after they have been altered appropriately. The pictures provide a more abstract representation than textual algorithms. The programmer draws a picture of the general case first, and the structure of this picture enables the system to draw most of the other before-pictures automatically. This considerably reduces the labour involved in generating a complete algorithm, and the probability of logic errors.

Active template are applicable, or at least adaptable, to a variety of languages; the particular syntax presented here was designed for HyperPascal [7]. To give focus to the discussion, we start with an overview of HyperPascal.

This general-purpose visual programming language has semantics broadly equivalent to Pascal's. Its development has been fuelled by a desire to exploit the capabilities of common computer technology to improve representations of conventional programming operations.

Many systems exploit Graphic User Interfaces, and the high-speed processing now available in any PC, to provide the programmer with browsers, debuggers and so forth. Others [1], [4], [8] have used the technology to support new programming paradigms. HyperPascal, by contrast, has been developed as a testbed for refining ideas about better representations for conventional programming constructs, with a view to their future use in other languages.

HyperPascal uses three types of picture (cf. [9], [10]):

> The scope of variables is represented in the Scope Tree, a simple tree in which each node contains the declarations of a subprogram, its variables and types.
>
> The manipulation of data is represented in the Action Tree, in which flow of control is mapped onto a tree whose leaf nodes contain specifications of assignment sequences. Each subprogram has its own Action Tree which occupies its own window.
>
> The appearance of I/O is specified in the Forms Windows. These simple WYSIWYG editors are linked to I/O assignments in the Action Tree. The use of I/O assignments is one area in which HyperPascal's semantics depart somewhat from those of Pascal.

Flow of control in HyperPascal is specified in the Action Tree by a diagrammatic syntax developed from Doran and Tate's [2], [3] structure diagram notation.

The tree's root is a subprogram header; intermediate nodes are flow of control icons, and leaf nodes specify data manipulations (assignments, i/o, and subprogram calls).

Figure 1 shows a coathanger-shaped flow-of-control icon with four children representing arbitrary subtrees. Execution follows one of the paths through the icon; a child attached to it by a simple connection (no diamond) is executed unconditionally if the flow of control reaches its attachment point; a child attached by a conditional attach-

ment (a diamond) is conditionally executed. If the test fails, execution continues along the bottom path through the loop; if it succeeds, the alternative path (exit from the flow icon, *via* the exit diamond, or premature start of the next iteration, *via* the go-back diamond, and respectively) is chosen. Thus, in Figure 1, the first and fourth children will cause the loop to terminate immediately if they succeed; the second will execute if the flow of control reaches it. The third, if successful, will jump to the next iteration.

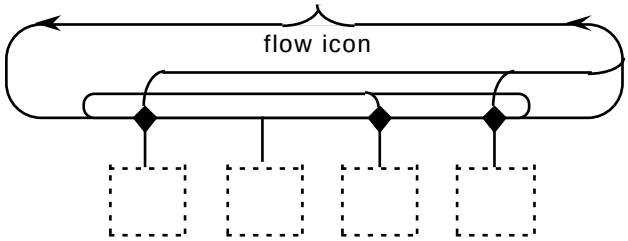

Figure 1: Flow-of-control icon with four children

The pointer-manipulation mechanisms available *via* the Action Trees of our preliminary version of HyperPascal were semantically similar to those in Pascal. This has been a cause of frustration, not least amongst the language's designers, whose initial goals in developing HyperPascal included allowing programmers to specify dynamic data structure manipulations by drawing pictures like the general-case illustrations so often included in text books. Figure 2 shows a typical example, the insertion of a node containing a new value, x, into L, an ordered list. The dashed pointer indicates that the pictured part of the data structure need not be at the head of the list.

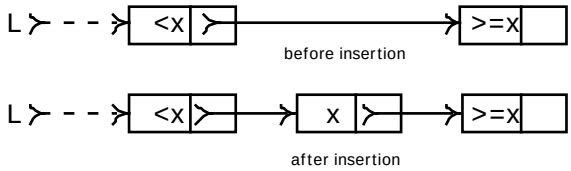

Figure 2: Inserting x, a new value, into L, an ordered list

Implementing these intuitively obvious pictures as a conventional program involves writing code to check that the data structure is in the configuration shown in the before-picture, and to convert it to the configuration shown in the after-picture. The complete insertion algorithm involves a number of other cases. Coding these individually is painstaking and error-prone work. However, the difficulties derive not from nature of data structures, but from the low level of abstraction of the textual tools for manipulating them.

Personal experience suggests that most expert programmers develop a mental library of stereotypes for such algorithms to circumvent the programming tools' close focus. However novice programmers easily lose sight of "the big picture" when coding pointer manipulations. Fix, Wiedenbeck, and Scholtz[5] have identified this (they call it "labelling complex code segments with a plan label") as a significant difference between novice and expert programmers.

GENERAL ASPECTS OF ACTIVE TEMPLATES

The Active Template system, a semi-automatic generator for pictures defining data structure manipulations, has been added to the definition of HyperPascal.

A data structure manipulation generated using active templates comprises a set of before/after-picture pairs executed in a loop. The list insertion operation shown in Figure 2 would be represented by the pictures in Figure 3. The labels B-P1 and A-P1 have been added to Before-Picture-1 and After-Picture-1 to facilitate discussion. (The white node disposal arrow is described below.)

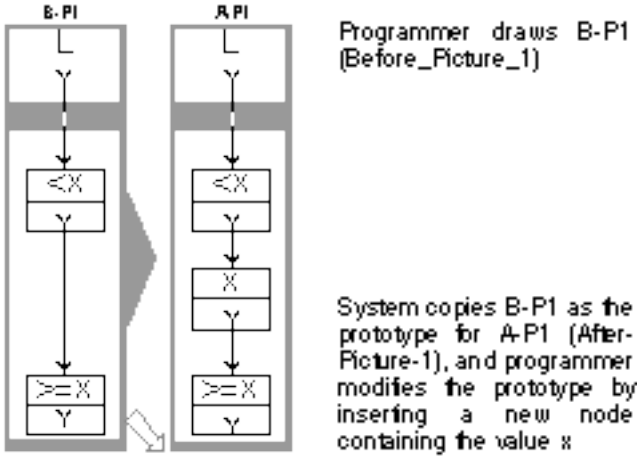

Figure 3: Before/after-picture for list insertion

Semantics

Each pair of pictures is compiled into code which could be rendered informally as: "if the data structure matches the before-picture, then make it match the after-picture".

Differentiation between templates

A before-picture has a broad arrowhead on the boundary adjacent to the after-picture, pointing at the after-picture.

Template editing

Pointer fields initially contain an undifferentiated pointer (a pointer tail, with no arrowhead,), which may or may not be nil . However, the programmer may extend the shaft of the pointer, whereupon a node of the type it references appears at its head. Deleting the node makes the pointer nil.

As a type-check, an undifferentiated pointer appears beside any pointer variables in a HyperPascal program. Dragging an arrowhead out of such a pointer fires up the active-template editor, and the templates can then be constructed.

Figure 4 shows how the value of pointer B can be copied into A. The programmer effects this by dragging an arrowhead from A onto the tail of B, so that A ends up pointing at wherever B was pointing. The diagrammatic syntax for the copy operation is identical whether B is undifferentiated, nil, or pointing at a node.

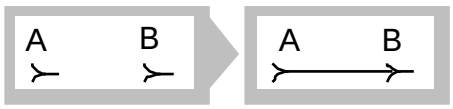

Figure 4: Making A point to the same place as B generates "A := B"

Pictorial declarations

The system knows how to draw the node at the end of a differentiated pointer because records are declared diagrammatically, with a drawing tool. Figure 5 shows declarations of linked list and binary tree nodes. The dashed arrows are "pointers-to-more-of-the-same". In an earlier version of the system, we represented a node which points to another of the same type using arrows pointing back at the parent node, but the declarations looked like instantiations of a structure in which a node points to itself. The current version, while it needs to be explained, seems clearer thereafter. As Graf [6] has pointed out, visual language designers should avoid developing a visual metaphor for every possible language component.

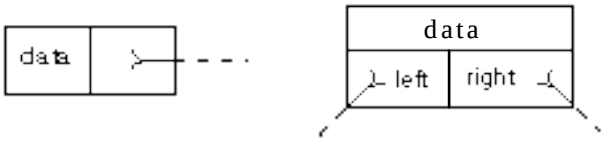

Figure 5: Pictorial declarations

Windows in templates, and structure traversal

A template need not show all of a data structure. The white areas surrounded by grey are windows onto parts of the data structure, and the grey area represents the parts of the data structure which are not relevant to the current operation. In textual programming languages, temporary pointers are used for keeping track of the nodes of interest, and to traverse a structure, new values are assigned to the temporary pointers. Figure 6 shows how updating a window in an after-picture achieves this more intuitively.

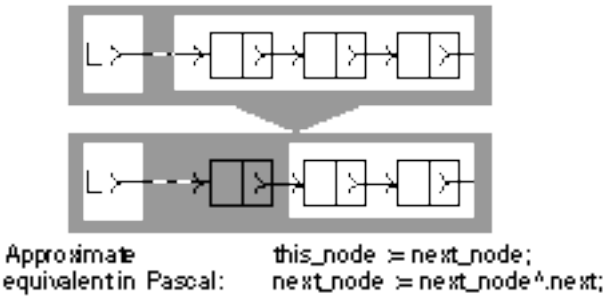

Figure 6: Moving a window to traverse a structure

Of course, a traversal step can include alterations to data fields. This would be necessary if, for example, all the data fields in a structure were to be set to zero. The after-pictures would show an updated data-field assignment as well as an updated window.

Connections between windows

A dashed arrow crossing a grey area represents sequence of one or more pointers between two parts of the data structure *via* parts of the data structure which are not currently of interest. The programmer can drag a section of the surrounding grey border across a white window to create such a representation, whereas dragging one window till it abuts another indicates that no intermediate pointers exist between them.

Template linking

All nodes in a before-picture also appear in the associated after-picture. Repositioning a node in the after-picture automatically moves it the same amount in the before-picture. This allows an association-by-position so temporary pointers are not required. Nodes added to an after-picture are not added to its associated before-picture.

Node disposal

It was mentioned earlier that the white arrow shown in Figure 3 is a node-disposal tool. Clones of the arrow can be dragged into the after-picture and attached to nodes. This causes the compiler to emit code to `dispose` of those nodes after the other manipulations specified in the after-picture have been performed.

Field testing and assignment

Any data field in a before-picture may be labelled with a value range; the system generates code to test that the field has a value in the specified range as part of the state test associated with the before-picture. Unlabelled fields are not tested. Expressions may be included in the fields of after-pictures, and the code for evaluating them and for storing the result in the field is generated as part of the code for generating the configuration shown in the after-picture.

Multiple structure references

The pictures may contain more than one data structure.

Automatic generation of before-pictures

The static system described so far could be used for programming data structure manipulations, and would be a modest improvement on the simple textual representation of conventional Pascal pointer-based algorithms. The programmer would construct a complete set of before-after-picture-pairs composed of self-drawing nodes and pointers. No previous_node, current_node or next_node pointers would have to be maintained. The compiler would generate code for a loop in which, in general, the structure-traversal case would repeatedly match until a configuration occurred in which the desired operation could take place; that manipulation would then be performed, and the loop would terminate.

However, it is feasible to provide active support for the generation of both before- and after-pictures, automating a large proportion of their production. Consider a Pascal version of the code which the system would generate for the first general-case before-picture (Figure 3). It maintains `this`, a pointer to the top node in the picture. The code checks for the existence of the nodes in the diagram, and for the data values specified in the fields. It needs to perform the tests in a strict order, from the node closest to the entry pointer to the node furthest from it, to avoid invalid pointer references. Specifically, it should be:

```
if this <> nil then
  if this^.data < x then
    if this^.next <> nil then
      if this^.next^.data >= x
        then terminate(A_P1)
```

Note that the procedure `terminate` changes the data structure to look like A-P_1, and it also sets a housekeeping variable, `searching`, to `false` in order to terminate the data structure traversal.

The series of `if`s in this algorithm can be matched to a series of `else`s, each corresponding to a possible configuration of the data structure. Furthermore, the system has enough information to draw pictures of these cases. Consider the innermost `else` clause. It would be executed if `this<>nil and this^.data<x and this^.next <> nil and this^.next^.data < x.` The editor can automatically construct the template shown in Figure 7 for a before-picture incorporating this whole test.

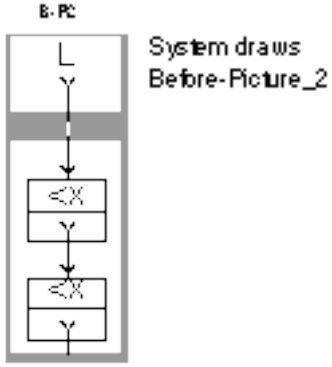

Figure 7: This system-drawn template for a Before-Picture needs specialisation

The programmer needs to do very little editing to turn this into a before-picture representing one of the other cases which have to be tested for. If the data structure has the configuration shown in Figure 7, the operation to be performed on it depends on the value of the undifferentiated pointer in the lower node. If it is nil, the new data value just needs to be added after that node. If it is non-nil, more of the data structure must be examined, to locate the insertion point for the new data value. The programmer therefore edits the template by extending the undifferentiated pointer to point to a node. Note that this doesn't alter the data structure; it only specifies a more extensive test. The editor then automatically generates a template for the after-picture, which the programmer also edits to show the desired alteration to the data structure. Figure 8 shows the result.

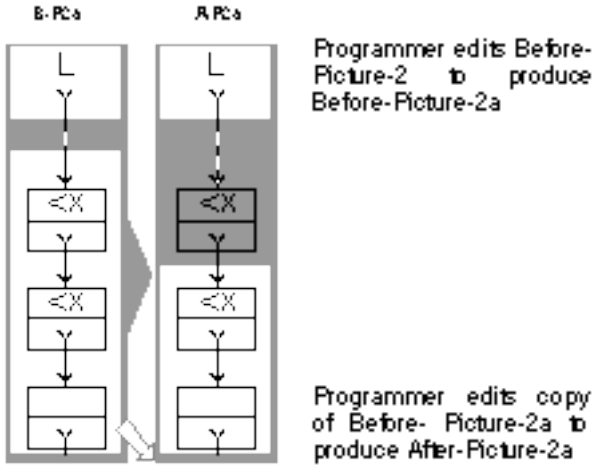

Figure 8: Updating the window before the next iteration

In fact, the after-picture shows the data structure unchanged with respect to the before-picture, because the before-picture shows a configuration in which the location for the new data item has not yet been found, so another iteration of the search loop is necessary. However, the window has been updated to show which part of the data structure will be examined next time round the loop. The development environment software only has to retain a pointer to the first node in the window in order to keep track of the part of the data structure in view. On the next iteration of the loop, it will use this, in conjunction with the before-pictures in the program, to generate the necessary tests.

As a result of completion of the active templates shown in Figure 7, the algorithm which the system has been generating has been augmented by a nested `if`-statement. The system can therefore automatically generate templates for the before- and after-pictures, corresponding to the `else` of the nested `if`, before continuing with the surrounding `if`. The programmer only needs to add a new node to complete the after-picture shown in Figure 9.

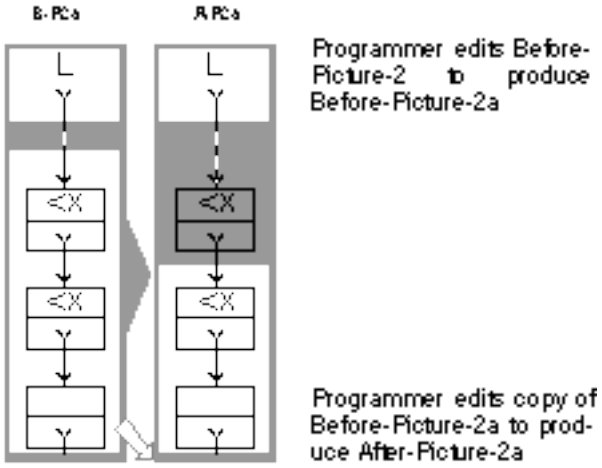

Figure 9: Completing the nested case

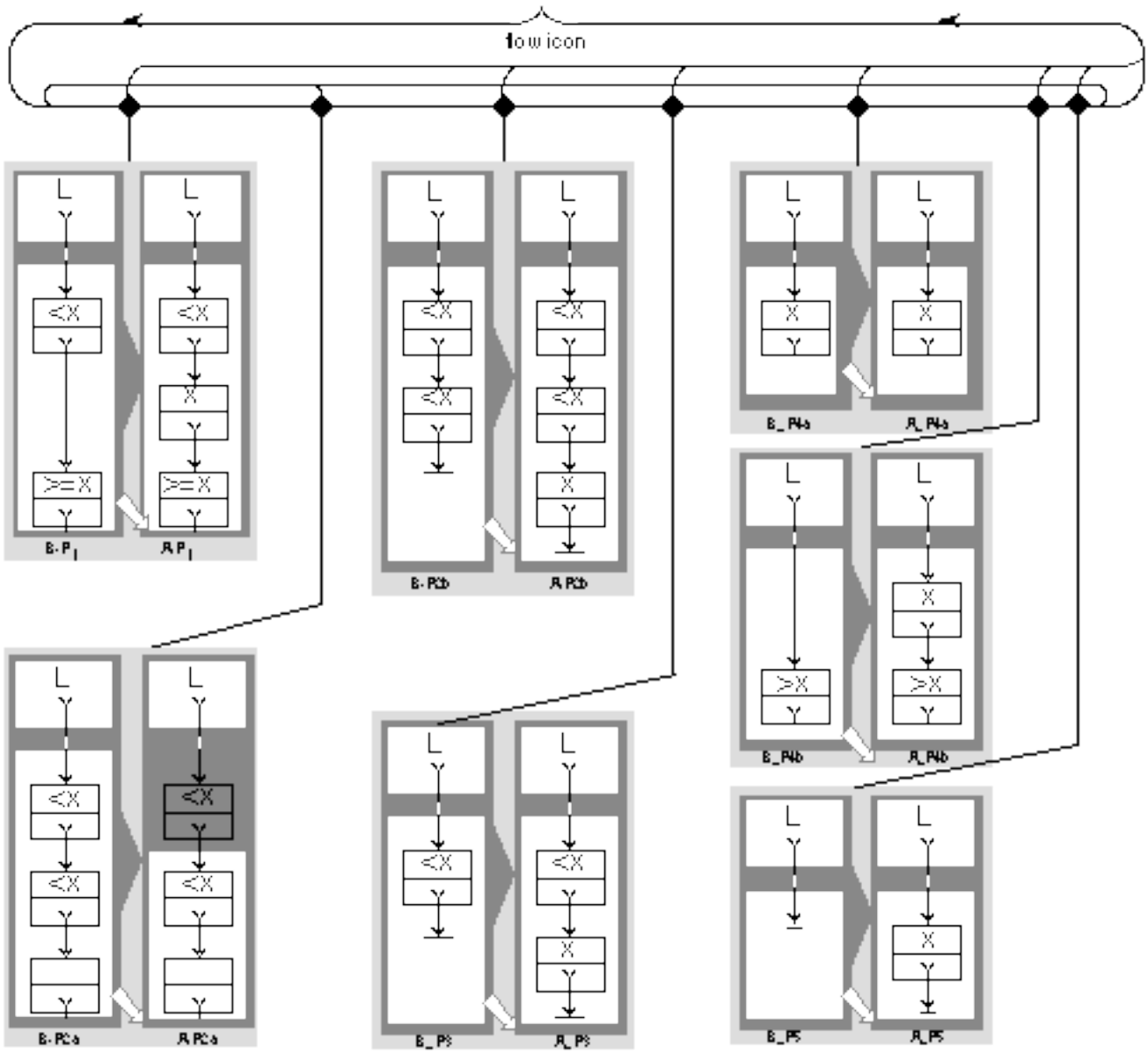

Figure 10: Complete specification of the insertion of a new value into an ordered list

At this stage, the system has enough information to generate code for the following part of the search algorithm:

```
if this <> nil then
   if this^.data < x then
      if this^.next <> nil then
         if this^.next^.data >= x
            then terminate(A_P1)
            else if this^.next^.next <> nil
                    then continue(A_P2a)
                    else terminate(A_P2b)
```

The process of building up the cases continues, with the system supplying most of the pictures. Of the remaining four before-pictures included in Figure 10, (the complete pictorial algorithm), three are completely automatically drawn by the development environment software, and one ($B\text{-}P_{4a}$) needs a minor specialisation similar to the one described above. The automatically-drawn templates for the after-pictures also require very little modification to prepare the algorithm shown in Figure 10, which also includes the parent loop icon (which has been omitted so far).

The algorithm which is generated for the whole list insertion becomes:

```
this := L;
searching := true;
while searching do
if this <> nil
   then if this^.data < x
      then if this^.next <> nil
         then if this^.next^.data >= x
               then terminate(A_P1)
               else if this^.next^.next <> nil
                     then continue(A_P2a)
                     else terminate(A_P2b)
      else terminate (A_P3)
   else if this^.data = x
         then terminate(A_P4a)
         else terminate(A_P4b)
```

```
else terminate (A_P5);
```

The automatic generation of cases works most satisfactorily if the first before-picture the programmer draws is complex, because a complex before-picture produces a skeleton traversal algorithm containing a large number of `if`s, and a correspondingly large number of `else`s. Even if the programmer has to specialise some of those `else`s, the complex initial case will have automated much of the work of generating the other before-pictures in the algorithm.

Although generating programs for structure manipulations using this sort of diagrammatic notation seems, at first blush, prone to run into screen space problems, consider the type of diagrammatic notation contained in Figure 1. This was presented as an instantly recognisable representation of an insertion into a linked list. The programmer can, having produced the complete algorithm, stack the pictures it contains so that only the evocative general case is visible at the top of the stack. In this way, space is saved, and detail is suppressed, without any comprehensibility penalty.

Conclusion

Active Templates are a visual language component for programming data structure manipulations. Although specifying an operation such as insertion of a new value into an ordered list (see Figure 10) may involve a number of different cases, each with its own before- and after-picture, it has been shown that once the first picture has been produced, most of the other before-pictures can be generated automatically, and each of the after-pictures can be produced by editing the before-picture with which it is associated.

Such an approach allows programmers to keep sight of the wood in spite of the trees. It frees them from much of the detail usually involved in constructing data structure manipulations such as maintaining pointers to nodes of interest, and ensuring that all configurations of the data structure have been allowed for.

The practical effects of programming using Active Templates will be investigated by incorporating them into HyperPascal. However, in common with most other aspects of HyperPascal, the active template paradigm is applicable to a wide variety of current programming languages.

REFERENCES

[1] Cox, P.T., and Pietrzykowski, T. 'Using a Pictorial Representation to Combine Dataflow and Object-orientation in a Language-independent Programming Mechanism', *Proc. Intl. Comp. Sci. Conf.*, 1988, 695-704.

[2] Doran, B., and Tate, G, 1972a 'An Approach to Structured Programming, Part I', *Massey University Department of Computer Science Publication no 6.*

[3] Doran, B., and Tate, G, 1972b, 'An Approach to Structured Programming, Part II', *Massey University Department of Computer Science Publication no 9.*

[4] Edel, M. 'The TinkerToy Graphical Programming Environment', *IEEE proceedings COMPSAC*, 1986, 466-471.

[5] Fix, V., Wiedenbeck, S., and Scholtz, J., 1993: 'Mental Representations of Programs by Novices and Experts', *INTERCHI '93.*

[6] Graf, M., 1990: 'Visual Programming and Visual Languages: Lessons learned in the Trenches', in *Visual Programming Environments: Applications and Issues*, ed. Ephraim P. Glinert, 452-455.

[7] Lyons, P., Simmons, C., and Apperley. M. 'HyperPascal: A Visual Language to Model Idea Space', *Proc. 13th NZ Computer Society Conference* **2**, August 1993, 492-508.

[8] Reiss, S.P., 1985 'PECAN: Program Development Systems that Support Multiple Views', *IEEE TRANS Softw. Eng*, **SE-11**, 3, 1985, 276-285.

[9] Shu, N.C., 1986 'Visual Programming Languages: A Perspective and a Dimensional Analysis', in *Visual Languages*, ed. Shi-Kuo Chang, Plenum Publishing, NY 1986.

[10] Williams, C.C., and Rasure, J.R., 1990 'A Visual Language for Image Processing', *1990 IEEE Computer Society Workshop on Visual Languages.*