A Visual Language for Data Structures Programming

Richard Wodtli and Paul Cull
Department of Computer Science
Oregon State University
Corvallis OR 97331
{wodtli,pc}@cs.orst.edu

Abstract

Data structures are more easily understood when they are presented visually rather than textually. We have developed a system, Calypso, to allow the visual definition of data structures programs using pictorial pattern/action pairs in an imperative setting. We present several examples including rebalancing an AVL tree and sorting an array using the Quicksort algorithm. These examples demonstrate the superiority of this visually-based approach over textual specifications. Calypso is based on a general framework for building and combining visual notations in various domains. This framework permits Calypso to be easily extended with new data structures and abstractions.

1. Introduction

The human brain can process certain kinds of visual information virtually instantaneously, whereas language is processed sequentially. Designing programming formalisms that effectively engage the brain's fast visual processing capabilities could lead to programs that are significantly easier to understand and maintain. Many researchers are exploring ways of incorporating pictorial and other kinds of information into programming on many levels [1, 2].

The principal task of many programs is creating and transforming data structures, and many data structures and associated operations have natural pictorial representations familiar to most working programmers. Many operations on data structures are difficult to understand when presented in a purely textual form. A data structure can only be viewed in a very local way in a textual program because each operation refers only to one or two components of the structure. Using pictorial representations in which individual transformations can readily be seen in the context of the entire data structure aids substantially in understanding programs.

The data structures programming literature has been a rich source of ideas and inspiration, suggesting among other things *pictorial patterns* and *pattern transformations*, the key concepts on which Calypso is based. The pictures generally used to show AVL-tree rebalancing following the insertion of a new node form an excellent example. Figure 1, adapted from [3], is taken from a typical presentation of this operation:

The top patterns represent the AVL tree after insertion but before rebalancing, and the bottom patterns represent the AVL tree after rebalancing. These patterns are simple and general, naturally subordinating unnecessary detail, yet a brief perusal is sufficient to garner an extraordinary amount of information. They are unquestionably easier to understand than a purely textual description.

Using patterns also fits well with the well-known and powerful "divide-and-conquer" problem solving approach: finding and solving subproblems, then combining the subproblem solutions to get the solution to the original problem. In data structures programming the subproblems are associated with substructures such as the subtrees in Figure 1. This correspondence of patterns to the divide-and-conquer technique suggests that such patterns will find natural use in a wide range of applications.

This paper presents an overview of a visual programming language Calypso designed for specifying algorithms on data structures. Calypso uses compile-time type checking with an ML-style [4] type system with parametric polymorphism. Calypso is an imperative language because destructive manipulations on data structures are readily and intuitively represented visually, because current practice and education in data structures programming is still principally imperative, and because the imperative approach still results in more efficient programs on conventional architectures using well-understood and tested compilation techniques. Many efficiency issues simply cannot be addressed in a functional language. For instance, Quicksort is more

useful than other "fast" sort algorithms because it can be done wholly "in-place," that is, without copying the structure being sorted. This characteristic is critically dependent on how the Partition phase is implemented. In functional programming languages the distinction between copying and non-copying algorithms cannot be expressed, the Partition phase becomes trivial, and the point of Quicksort is lost.

Calypso is based on a general framework for building picture editors. These editors allow interactive creation of representations of objects in various domains, such as data objects, pictorial patterns, and programs. Translators attached to the picture editors then process these representations.

Calypso was implemented in X2 [5, 6], which was designed by David Sandberg at Oregon State University in the 1980's.

2. Motivating Examples: What Can be Done

2.1 AVL-Tree Rotations with Pictorial Patterns

A good example of the use of pictorial patterns, and the one that provided the initial impetus for the work presented here, is the collection of pictures that most data structures textbooks use in some form to show how to rebalance an AVL tree after inserting a new node.

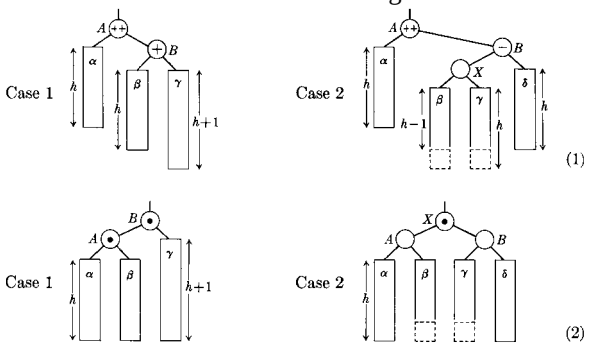

Figure 1: AVL-Tree Rebalancing

The set of pictures in Figure 1 is from Knuth's *Sorting and Searching* [3], but all texts that discuss AVL trees present the rebalancing algorithm in basically the same way.

Figure 1 is interesting in many ways. First, it is *general*. There are two *patterns* and a specification of a small set of *unconditional* transformations on those patterns. These pictures cover two of the four possible cases that can arise — the others occur when the directions of imbalance are opposite from the ones shown here, and the pictures are just reflections of the ones shown. By elaborating this notation a little, and not being too literal about left-to-right orientation, all the possible cases in this situation can be represented with just these two pictures.

The *bars* α, β, etc., represent arbitrary subtrees. These bars represent actual run-time objects that have *attributes* and can be *referred to* by other objects. For instance, the subtrees in Figure 1 have an attribute *height* and have pointers from nodes A, B, and X.

The height attribute is used together with some textual expressions in the variable *h* to specify assertions about the relative heights of the subtrees when the directions and amounts of imbalance are as shown in the nodes. In these pictures the information provided by the height attribute notation is just intended to provide useful information to the human reader, but it suggests a way to pictorially represent a class of functions on data structures and usefully incorporate invocations of those functions into pictorial patterns.

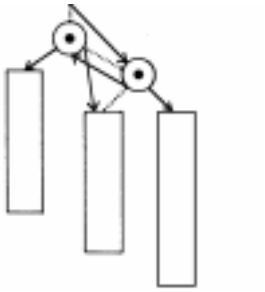

Figure 2: AVL-Tree Detail

Figure 1 shows the transformation by labeling the various components and using before-after pairs, with the labels showing the correspondence of objects. There are other ways to do this — for instance, as shown in Figure 2, "before" positions of pointers could be represented with dotted lines, "after" with solid lines, and the nodes and subtrees left where they are. Viewed this way, it is easy to see that the transformation can be represented as a set of direct manipulations, and the object labels are no longer needed. On the other hand, this representation would rapidly become confusing for more complex transformations. A complete system on the lines of Calypso would offer multiple views of transformations.

Another interesting thing about this transformation is that none of the unit transformations is dependent on the outcome of any of the other unit transformations, so they can be done in any order or even simultaneously.

The final point of interest relates to the node X in Case 2. It is analogous to the height attribute specifications in that — in terms of tests implied by the pattern — it is redundant: the imbalance information on the two nodes imply its existence. On the other hand, it is needed in the transformation specification: it must be referred to, and its components must be modified. It is necessary to be able to distinguish "essential" components of a pattern — those things meant to imply run-time tests

— from components whose existence follows from the fact that the run-time structure being tested matches the "essential" pattern.

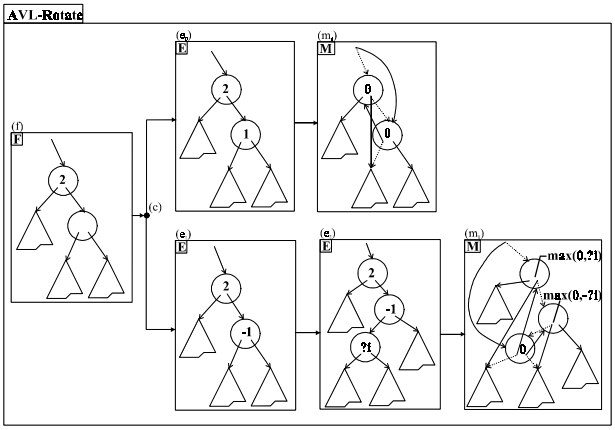

Figure 3: AVL-Tree Rotation Procedure

Figure 3 is what the AVL tree example looks like as a procedure in Calypso. This picture has two levels. The top level consists of a pattern formation box, a case object, pattern elaboration boxes, pattern manipulation boxes, and directed arcs linking everything together. This level essentially specifies flow of control. The bottom level, which consists of the pictures inside the individual boxes, represents pictorial patterns and operations on them.

The pattern formation box *f* specifies the formal parameters and the argument structure, and provides the basis for an initial binding of pattern components to run-time structure components. All the components of the argument structure are assigned internal names, and run-time values are assigned (bound) to these names when the procedure is invoked. All the values bound to the names are functions of the formal parameters. In this example, the formal parameter is the pointer at the top; all the bindings are expressed in terms of the run-time value of this pointer. The pattern formation box is also an assertion about the structure of the argument — in the example it consists of a pointer and a tree with at least two nodes. If the argument does not satisfy this assertion, there is a run-time error, and the result is undefined.

The Case object *c* indicates that the pattern elaboration boxes following it are to be considered tests of the argument structure. In this case there are two, and they test the values of the imbalance fields of the two nodes. If the argument structure happens to match none of the patterns in the following elaboration boxes, it is again considered a run-time error, and the result is undefined.

In the top case, the elaboration box e_0 is followed directly by a pattern manipulation box m_0, in which the "rotation" shown in Knuth's case 1 is specified. In the bottom case, the elaboration box e_1 after the Case object is followed by another pattern elaboration box e_2 in which the left subtree of the node with the "-1" imbalance value is elaborated to a non-nil value. Because this second elaboration box does not directly follow a Case object, it is an assertion about the argument structure; if this assertion is not satisfied, there is a run-time error. Code is generated to bind the appropriate components of the argument structure to internal names for the new pictorial pattern components. Finally, the pattern manipulation box m_1 connected to this elaboration box specifies the manipulation shown in Knuth's Case 2. The *pattern variable* ?*i* communicates imbalance information to text expressions in m_1.

2.2 Quicksort with Pictorial Patterns

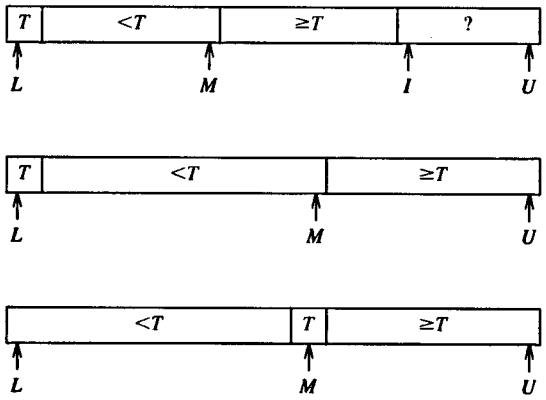

Figure 4: Quicksort Patterns

Recursive structure objects appear again, this time as arbitrary sub-arrays, in the discussion of Quicksort found in Jon Bentley's book *Programming Pearls* [7] (Figure 4). This use of pictorial patterns is more abstract than the ones seen earlier. The first pattern pictorially represents an *invariant*, a set of relationships between data structure components that must hold if another iteration of an unspecified process is to be performed. The second pattern represents the state of the data structure when the loop terminates — that is, when the invariant represented by the first pattern no longer holds. The third pattern simply shows the result of exchanging two elements of the second pattern. Such an exchange is a fixed-length, non-conditional transformation that can be specified by direct manipulation of the second pictorial pattern. The patterns shown in Figure 4 were originally intended for use by humans and had to be adapted for use in Calypso, but also inspired some enhancements in the Calypso model and notations.

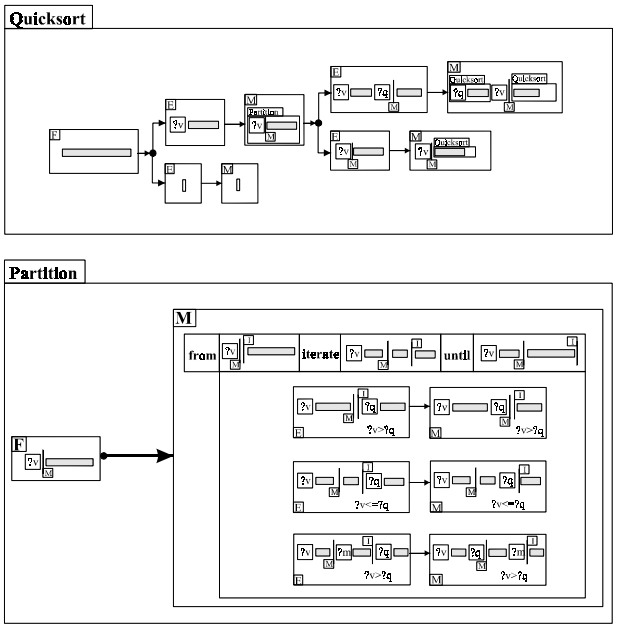

Figure 5: Quicksort Program

The Quicksort definition shown in Figure 5 has two components. The top picture defines the Quicksort procedure itself; the bottom picture defines the partition procedure used by Quicksort. Quicksort is defined on an arbitrary array. Partition is defined on an argument that consists of an array with at least one element and a special "marker" unit which is basically an integer variable whose value marks a "current location" in an array. The Partition procedure uses the first element of its array argument as the partition element and partitions the *rest* of the array using that partition element. Thus, the first element of the array passed to Partition is not in its proper location following the execution of Partition.

These two definitions have basically the same structure displayed in the AVL tree example. One extension is that a pattern manipulation, for instance where the Partition procedure is invoked, results in a pattern that can be further elaborated and manipulated. Another extension is in the Partition definition, where a loop construct, **from-iterate-until**, has been used. This construct starts with a structure that matches an initial pattern and repeatedly performs some manipulations on it until it matches some terminating pattern. In the Partition definition, the initial pattern is an array with two markers, the manipulations are moving markers and interchanging array values, and the terminating pattern has the *I* marker at the end of the array.

As exemplified by the use of the markers, an argument structure can be augmented with additional components in pattern manipulation boxes. These additional components are local to the branch of the program beginning at the box where they are defined.

Pattern variables ?*v*, ?*q*, and ?*m* are used to communicate between textual and graphical parts of the Quicksort specification. Values for these variables are bound at run time using expressions derived by the picture processor at edit/compile time.

3. System Architecture

It became clear early in the development of Calypso that a number of kinds of small picture editors would be needed, so a significant initial effort was put into developing a framework for building general picture editors. The procedure definition editor alone consists of four sub-editors, one for the upper definition level representing control flow and one for each of the three kinds of pattern transformation editors. Picture editors have translators that can be invoked at any time after an edit session with that picture is completed.

Picture editors are tools for building external representations of objects in specific domains. These representations can then be translated into an internal representation suitable for a given application. The term "picture editor" is somewhat misleading because it seems to imply that the representation in question is necessarily graphical. In fact, the representation could be textual or three-dimensional. The source code that constitutes the external representation of program in textual programming languages also fits this paradigm.

Because it is built on this framework, Calypso is readily augmented with new abstractions and constructs. In particular, an interface object template editor could be added, permitting new kinds of pattern objects to be created and given meaning in terms of existing static structure and procedure definitions. New abstractions could then be defined entirely within Calypso, whereas now interface object templates must be built in the host environment.

4. Procedure Definitions: An Example

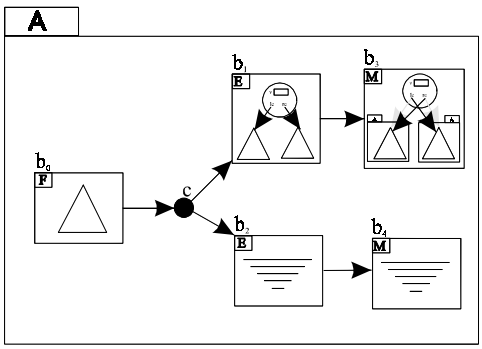

Figure 6: A Procedure Definition

Figure 6 is the definition of a procedure *A* in Calypso. This recursively-defined procedure takes a binary tree as its argument and interchanges left and right sub-trees throughout the tree.

The argument structure is represented in the *pattern formation box* b_0. The symbol in the box b_0 is a *recursive-structure template* for binary trees. It represents the disjunction of two possibilities, that the argument is a binary tree node or it is the **nil** object.

The box b_0 is connected to the *case object c*, which is a conditional construct. A case object is connected via outgoing arrows to *pattern elaboration boxes*, which represent tests for branches of the conditional statement. Here there are two cases, one where the argument tree is non-**nil** and the other where it is **nil**, tested in the pattern elaboration boxes b_1 and b_2, respectively. The patterns in b_1 and b_2 are elaborations of the pattern in b_0.

In the case where the tree is non-**nil**, the procedure swaps the children pointers so that the left child field points to the right sub-tree and the right child field points to the left sub-tree and invokes itself recursively on the two sub-trees. These operations are represented by the contents of the *pattern manipulation box* b_3. Reassignment of pointer values is represented by moving the destination feature of the arrow representing the pointer to the object representing the value it is to be assigned. In this case, the small boxes enclosing the sub-trees in b_3 and tagged with "*A*" are instances of the call form for the procedure *A* being defined.

When the argument tree is **nil**, *A* simply returns without performing any operations. This null operation is represented by the picture in the pattern manipulation box b_4 being identical to the picture in the pattern elaboration box b_3.

```
procedure A(i0:BTreeP);                    {argument
                  definition and binding from b0}
var  i1,i2:BTreeP;                         {from b1}
begin
     if i0 <> nil then
          {from c and b1}
          i1 := i0^.lc; i2 := i0^.rc;
                    {i1, i2 bindings from b1}
          A(i1); i0^.lc := i2; A(i2); i0^.rc := i1
                    {manipulations from b3}
     elsif i0 = nil then skip    {from c, b2 and b4}
     else signal_error                     {from c}
     end if                                {from c}
   end proc;
```

Figure 7: Generated Code

The code generated by this picture would be similar to the Pascal-like procedure definition given in Figure 7.

The variables whose names begin with "i" in that figure are internal variables generated by the system. The values of internal variables are not modified after the initial assignment. They provide a way, invariant over all definable manipulations, of accessing the run-time objects represented by pictorial pattern components. The scope of internal variables in the target language code does not necessarily correspond to the scope of the corresponding pictorial pattern component. For instance, the internal variables i1, i2 correspond to the child tree recursive-structure objects bound in the elaboration box b_1, whose scope is over the boxes b_1 and b_2, but the scope of the internal variable names i1 and i2 is over the entire procedure definition in the target language.

```
type
     BTreeP = ^BTree;
     BTree = record
          v:integer; lc,rc:BTreeP
     end;
```

Figure 8: Structure Definitions

The Pascal type definitions needed for this definition are associated with the binary tree recursive-structure template and the binary tree node record template, and are shown in Figure 8.

The following figures are a series of snapshots of the process of constructing the procedure definition shown in Figure 6.

Before this series of snapshots, a new procedure definition form pictorial object has been instantiated (this would look like Figure 6 with no internal structure), and the user has specified the procedure name (*A*) and double-clicked on the interior of the form. This action invokes the *clickOp* for the interior feature, which activates the procedure definition Picture editor.

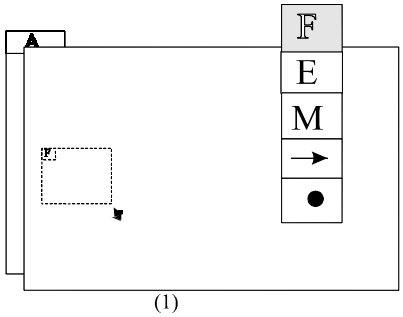

Figure 9: Snapshot 1

In snapshot 1, the pattern formation box pictorial object template has been selected from the palette representing the pictorial syntax environment and a pattern formation box is being placed on the left side of the procedure definition picture.

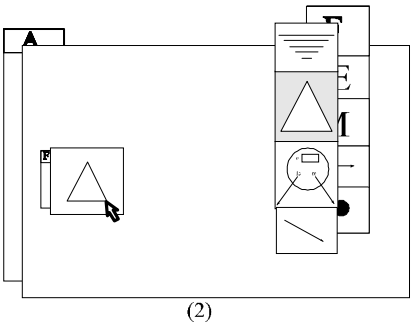

Figure 10: Snapshot 2

In snapshot 2, the user has double-clicked on the interior of the pattern formation box, activating the editor for a pattern formation picture. Note the presence of a new picture window and associated pictorial syntax environment palette. The binary tree recursive-structure template has been selected from the palette and the recursive-structure object is being placed in the picture.

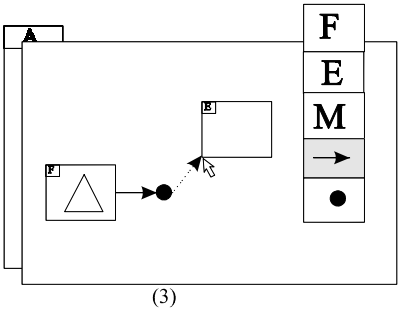

Figure 11: Snapshot 3

In snapshot 3, the pattern formation picture has been closed, a case pictorial object, an arrow pictorial object, and a pattern elaboration box have been added to the main procedure formation picture, and an arrow is being placed between the case object and the pattern elaboration box.

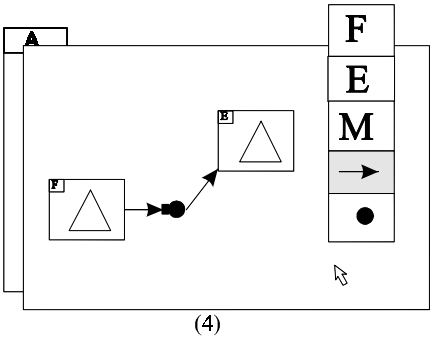

Figure 12: Snapshot 4

In snapshot 4, the arrow between the case object and the pattern elaboration box has been placed, and the system has copied the binary tree recursive structure object in the pattern formation box to the pattern elaboration box. Since the pattern in the pattern elaboration box must be an elaboration of the pattern in the pattern formation box, the system provides the pattern in the pattern formation box as a starting place for the elaboration.

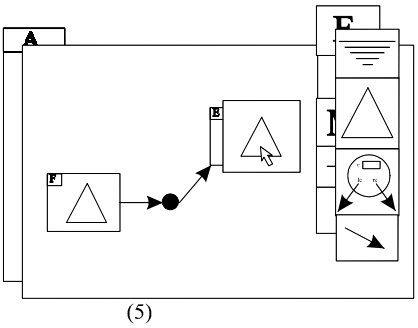

Figure 13: Snapshot 5

In snapshot 5, a second pattern elaboration box and an arrow connecting it to the case object have been added to the main picture, and the editor for pattern elaboration picture has been activated.

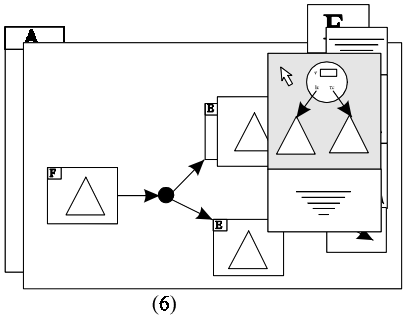

Figure 14: Snapshot 6

In snapshot 6, the user has double-clicked on the binary tree recursive structure object, bringing up a palette from which the non-**nil** alternative is being selected for incorporation into the elaboration picture.

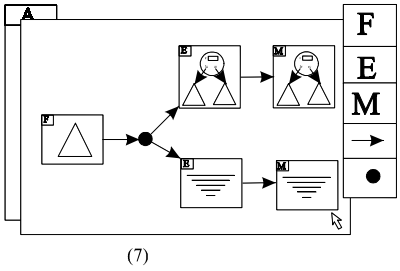

Figure 15: Snapshot 7

In snapshot 7, the pattern elaboration box has been edited to specify that the recursive structure object in it must represent the value **nil**, and the two pattern manipulation boxes have been added and connected with arrows to the elaboration boxes. Calypso has copied the patterns from the elaboration boxes to the connected manipulation boxes. The manipulation boxes, as they stand in this snapshot, both represent the **Null** transformation.

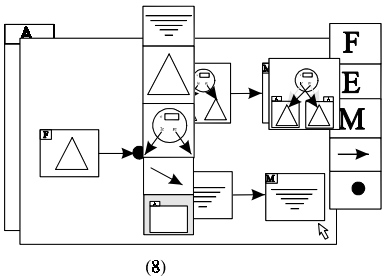

Figure 16: Snapshot 8

In snapshot 8, the top pattern manipulation box picture is being edited. Note that the pictorial syntax environment palette for this picture includes a pictorial object template, procedure *A*'s *call form*, from which are generated pictorial objects representing the invocation of procedure *A*.

5. Related Work

Calypso is related to a number of systems described in the visual programming literature. It is a significant generalization of these systems because it is based on a framework that permits the formation of complex representations of objects in arbitrary domains, allowing new kinds of abstractions to be created at all levels. Because this framework does not distinguish between pictorial and textual representations, Calypso can integrate pictures and text more flexibly and with an unprecedented fineness of granularity.

AMBIT/G [8] is the earliest system that clearly addresses the same general issues as Calypso. AMBIT/G is based on pattern matching and on representing data structure transformations as before/after pairs, with succeed/fail arcs connecting the patterns to indicate flow of control. The notion of recursive-structure templates is not used. The principal problem with AMBIT/G is that it has no abstraction mechanisms, so the pictures quickly become large and intricate

Pygmalion [9] is a graphical programming-by-example system that permits the definition of operations on data structures. It permits the formation, graphical representation, and composition of objects in diverse domains, for instance electrical circuit components. Pygmalion uses concrete examples and dynamic representations of programs exclusively.

PiP [10] is a pictorial programming-by-demonstration system that permits operations on data structures to be specified by direct manipulation of graphical representations of those structures. PiP is based on the functional programming model. PiP does not have recursive-structure templates. Transformations are specified and shown in individual windows that provide only local views.

ThinkPad [11] uses direct manipulation of pictorial representations of record structures to specify data structure transformations. The graphically specified programs are translated into Prolog. Pointer relationships between nodes are not shown, and ThinkPad does not have recursive-structure templates. The data structure pictures in ThinkPad are only patterns in the sense that they impose type constraints on function arguments. Their use is to allow record structures and record structure components to be referenced via mouse-clicking rather than using textual names in forming conditions and expressions.

The stated objectives for DataLab [12] are similar to those for Calypso, and within its range of expression the programs look very similar to the ones created using Calypso. However, DataLab was not designed with extendibility in mind and only operates on data structures composed of records and pointers. DataLab has something that looks like a recursive-structure template, but it has no real functionality — it cannot be elaborated from a fixed set of choices, and it cannot be used to express constraints.

GRClass [13] is a system for pictorial specification of algorithms on graph data structures. These structures are represented as relations, in the database sense. Many-one relationships are permitted, but not many-many. GRClass provides a static representation of programs, but like ThinkPad and other systems this representation is spread over several windows.

Pfeiffer [14,15] describes a visual language using pattern/action pairs for manipulating graph data structures. The language has a notion of type declaration, but does not seem to otherwise provide for data abstraction. Since all data structures are represented as graphs, with different kinds of relationships denoted by differently labeled arcs, there is inadequate differentiation among pattern components.

ChemTrains [16] is a graphical simulation language that uses pattern/result picture pairs. ChemTrains was developed for non-programmers, and does not use procedural abstractions. Its only control structure is the repeated application of elements of a set of pattern/result pairs to a picture constituting a data structure until none of the pattern/result pairs is applicable to the resulting picture. ChemTrains does not have the notion of recursive-structure templates; parts of a data structure not relevant to a pattern are simply left out of that pattern.

DEAL [17] is a visual language addressing many of the same issues and using a very similar approach to Calypso. It is based on the functional programming model, though it provides some constructs, such as iteration, that give it an imperative flavor. It permits implicit iteration, and as a consequence of the

representation used for that construct, the pictorial patterns do not necessarily *mean* all that they might seem to *imply*. Two distinct elements in a pattern might refer to the same entity, and the fact that one element in a pattern representing an array is to the left of another element in the array does not necessarily mean that its index in the array is smaller. This could be confusing in some circumstances.

A general system for integrating visual and textual programming languages is proposed in [18]. As an example the paper presents an AVL tree maintenance procedure formed in a language combining Prolog and pictorial data structure patterns. Since pictures and text are treated as fundamentally different entities, a rich interaction and blending of the two forms is not possible.

6. Conclusions

We have presented Calypso, a system for visually specifying algorithms on data structures. Calypso is applicable to the entire range of data structures and can be readily extended with pictorial representations of new abstract data types and control structures. Using pictorial patterns permits the effect of local actions on the complete structure to be readily grasped. Recursive-structure templates enlarge the class of specifiable constraints. The breakdown of effectiveness that many visual languages suffer when applied to large and complex problems should not be an issue with Calypso because of its support for multiple levels of abstraction and variety of pictorial relationships. The approach used in Calypso promises to be a significant leap forward in the interaction of programmers with computers.

7. References

1. K. Kahn (1996), Drawings on napkins, videogame animation, and other ways to program computers, Communications of the ACM, 39(8) (August), 49-59.
2. A. Cypher (ed.) (1993) Watch What I Do: Programming by Demonstration, MIT Press, 652 pp.
3. D. E. Knuth (1973) The Art of Computer Programming: Sorting and Searching, Addison-Wesley, 723 pp.
4. R. Milner (1978) A theory of type polymorphism in programming. Journal of Computer and System Sciences 17, 348-375.
5. D. Sandberg (1985) The design of the programming language X2. Technical Report 85-60-1 Dept. of Computer Science, Oregon State University.
6. D. Sandberg (1986) An alternative to subclassing. Object-Oriented Programming Systems, Languages and Applications Conference Proceedings, pp. 424-428.
7. J. Bentley (1986) Programming Pearls, Addison-Wesley, 195 pp.
8. C. Christensen (1968) An example of the manipulation of directed graphs in the AMBIT/G programming language. In: Interactive Systems for Experimental Applied Mathematics, (M. Klerer and J. Reinfelds, ed.) Academic Press, pp. 423-435.
9. D. C. Smith (1975) Pygmalion: a creative programming environment. PhD Dissertation, Stanford University, 234 pp.
10. G. Raeder (1984) Programming in pictures. PhD Dissertation, University of Southern California, 181 pp.
11. R. Rubin, R., E. Golin,, and S. Reiss (1985) ThinkPad: a graphical system for programming by demonstration. IEEE Software 2(2), 73-79
12. M. Al-Mulhem (1990). DataLab: a graphical system for specifying and synthesizing abstract data types. PhD Dissertation, Oregon State University, 153 pp.
13. G. Rogers (1990) The GRClass visual programming system. 1990 IEEE Workshop on Visual Languages, pp. 48-53.
14. J. Pfieffer (1990) Using graph grammars for data structure manipulation. 1990 IEEE Workshop on Visual Languages, pp. 42-47.
15. J. Pfeiffer (1995) Ludwig2: decoupling program representations from processing models. Proceedings, 11th IEEE Symposium on Visual Languages, pp. 133-139.
16. B. Bell and C. Lewis (1993) ChemTrains: a language for creating behaving pictures. Proceedings, 1993 IEEE Symposium on Visual Languages, pp. 188-195.
17. M. Erwig (1994) DEAL - A language for depicting algorithms. Proceedings, 10th IEEE Symposium on Visual Languages, pp. 184-185.
18. M. Erwig and B. Meyer (1995) Heterogeneous visual languages — integrating visual and textual programming. Proceedings, 11th IEEE Symposium on Visual Languages, pp. 318-325.