December 1994

Data Types for Graph-Based Visual Programming

Michael F. Kleyn
Department of Computer Sciences
The University of Texas at Austin, Austin TX 78712
kleyn@cs.utexas.edu

Abstract

This paper argues the appropriateness of using data types with *sharing* to characterize the underlying data structures of a large category of graphical programming interfaces - those interfaces which include building programs by interactively manipulating graphical elements in graphs as well as editing characters and words in text. The paper examines the difficulties in providing direct formal representations of the definitions and manipulations of such "graph" data types that allow sharing of structure. The problem of formalizing the class is shown to be closely related to similar problems that arise in many different areas including the specification of abstract data types, functional programming, and models of object-oriented and network databases. The paper presents the particular approach used in the context of our work on a high-level specification language for describing interactive graphical programming environments and an associated generator.

Keywords: graph types, recursive types, cyclic data structures, self-referential data structures, recursive instances, recursive data structures, multilinked data structures, abstract data types, shared mutable data structures, graphical program composition, pointer-based data structures.

1.0 Introduction

Graphical user interfaces are now in almost ubiquitous use. One class of application of graphical interfaces that is an active area of research is that of finding effective interactive graphical support for developing programs and specifications [1]. These graphical approaches to program development exploit the continuing performance advances in graphical user interface technology. Languages that are presented graphically are typically referred to by the terms *graphical/visual language*, or *visual formalism* [2], and the interactive interfaces that support programming in them are referred to by the terms *graphical/visual programming interfaces/environments.*

A large majority of existing graphical programming environments can be viewed as interactive systems that support "graph-based" languages. These are systems in which developing a program includes interactively connecting graphical elements into a structure which represents the program. The graphical elements that are connected are usually *nodes* that represent computation units and *links* that represent flows, dependencies, constraints, or transitions between the units. These nodes and links may be of several different types (and are usually iconically identified as such). The nodes and links may themselves be complex objects; they can contain components that further specify the required computation. These components are usually structured text, but in some graph-based languages the nodes or links may also be hierarchical, *i.e.*, a complete subgraph may a component of a node or link. Such hierarchy provides a way of managing the complexity of a program in a way analogous to using subroutines or macros in text-based languages.

When designing an interactive graphical programming environment, careful consideration should be made of the design of the data structure that contains the program, since it is this data structure that ultimately constrains the presentation and interaction of the interface with the user [3]. This paper describes a means for specifying a data structure that can be used to support graphical programming interfaces for the graph-based visual languages. We have developed *Glide* a (textual) specification meta-language similar in spirit to specification meta-languages of programming environment generators such as *Pan* [4], the *Cornell Program Synthesizer* [5], and *Gandalf* [6]. The meta-languages of these generators are used to describe programming environments for textual programming languages. These generators allow programs to be represented as instances of a complex data type that reflects the syntax of the language. The programming environment that is generated is derived from the specification of such a complex type. Our work extends the scope of programming environment generation to encompass graph-based visual languages. In so doing it was necessary to extend the class of complex data type that can be specified in such a meta-language. This problem and our approach used to solving it is the subject of this paper.

The paper is organized as follows. In Section 2.0 we describe graph-based graphical programming systems in more detail, show two typical examples, and identify the essential differences between graph-based and text-based program composition. In Section 3.0 we investigate how the structure and manipulation of complex data types can be expressed. Our presentation is a progressive one. We begin by reviewing how the most trivial class of complex data type can be described and then, step by step, remove restrictions so as to encompass a more general class of complex data type. In Section 4.0 we then argue that the class of data type suitable for systems described in Section 2.0 is a further natural generalization of those described in Section 3.0, and show why formal characterization of such types presents problems. The different solutions that have been proposed to these problems in the context of apparently very different application areas are then discussed in Section 5.0. We conclude by suggesting possible reasons behind this correlation between generalized complex types and graphical programming.

2.0 Graph-based Visual Programming

This section provides context for our arguments by reviewing two typical examples of existing graph based visual programming languages and then discussing some important differences between classical textual languages and these visual languages.

2.1 Example visual programming environments

The following two figures show typical examples of graph-based visual programming environments. The first figure shows CODE2, a system that was developed for task oriented parallel program development. In CODE2 a complete computation is represented by a hierarchical graph of nodes which specify units of computation. These are interconnected by links which represent abstract data dependencies between the units. Figure 1 shows the top-level graph of a parallel laplace solver program written in CODE2. Textual attributes which specify such aspects of the computation as the routing of data and the conditions for a computation to execute are accessible through forms which are displayed when a node or arc is selected.

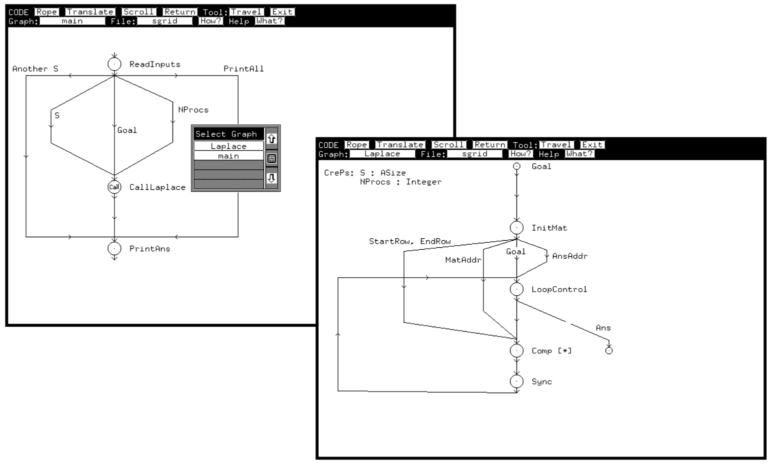

Figure 1. CODE2 Visual Programming Interface

Figure 2 shows a typical program in the AVS® graphical programming system. It is used for rapid development of data visualization programs. The nodes represent input, output, filtering, mapping, and computation modules. These modules act on large structured data sets (arrays) which are passed via the links.

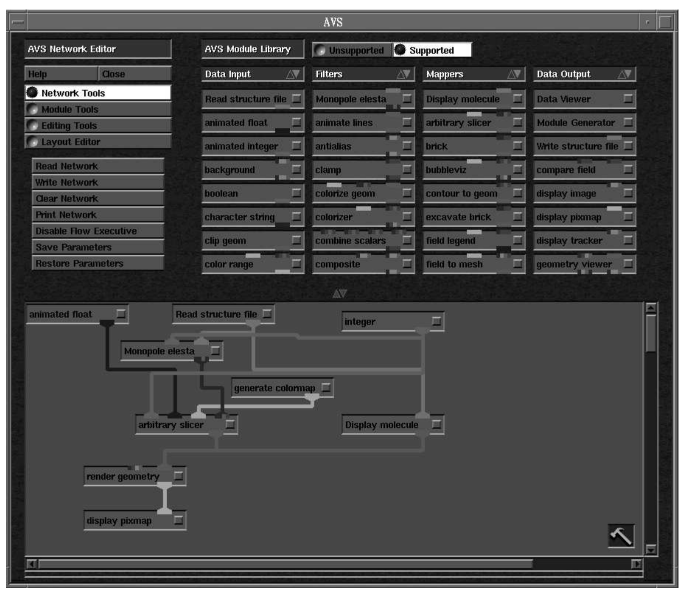

Figure 2. AVS Visual Programming Interface

Though these two systems are visually very different they are both graph-based visual programming environments. In both systems programs are created by interconnecting computation nodes and data arcs and then specifying the computation further by specifying the values of attributes associated with the nodes and arcs.

The distinction between graph-based and non-graph-based visual languages is not always clear cut. Some visual languages use interconnection together with some 2D graphical constraints. For example, there may be a graphical constraint that all inputs to a computation might always enter on the left side of a node, and all outputs leave on the right side of the node. Commercial examples of such "partial" graph-based languages include [7].

2.2 Comparison with textual programming

Making textual programs by composing elements into a sequence is a a very different program-creating mindset from making visual programs by composing elements into a graph. The key difference is that whereas in textual programs tokens (characters or lexical elements) are *concatenated* or *inserted* into a linear sequence, in a graph-based program they are *interconnected.* It is often stated that the basic difference between textual languages and graphical languages is that the former are "one-dimensional" while the latter are "two-dimensional". In the case of graph-based graphical languages however it is perhaps more accurate to characterize the difference as "one-dimensional" vs. "many-dimensional", since in graph-based languages any graphical element can potentially be connected to (become adjacent to) *any number* of neighboring elements - not just one left neighbor and one right neighbor. Note that we are speaking at the level of graphical elements that are manipulated; a programmer connects a "link" graphical element to a "node" graphical element in order to make the link and the node adjacent. A graphical element can even be transitively adjacent to itself through a cycle of adjacent graphical elements - this cannot happen in a text language. Even though the graph is graphically depicted as a two-dimensional geometric object, for most graph based languages the particular location of graphical elements is not meaningful. The graph has the same meaning no matter where the nodes and links are placed - it is only their interconnection that matters. However, just as in text languages with no formatting, it may be difficult to read a poorly laid out graph. The problem of providing good and efficient graph layouts for applications such as graph based graphical programming environments is the subject on-going research, see [8]

A second difference between text-based languages and graph-based languages is that the interactive programming environment that is used to construct programs is even more important. It is much harder to edit and manipulate programs of graph-based languages without machine support; each alteration of the program may require tedious re-arrangement of the positions of nodes, links and labels. Since the construction of graphical programs is nearly always mediated through interactive tools it is important to consider the steps and states of the interface on the way to reaching a complete correct program. In the case of text based programs there are many variations. Basic text editors only provide the ability to manipulate programs as arbitrary sequences of characters. More sophisticated editors that are "language-sensitive" are now increasingly common. These editors provide formatting, coloring, and interactive commands that are based on the structure of the language. Syntax-directed editors of the kind generated by the automatic generators mentioned earlier are a further step - they present the program as a structured data object and constrain the user to construct programs solely out of the component constructs of the language. These variations carry over into editing in graph-based languages.

3.0 A progression in specifying complex data types

Having provided some background on interactive textual and graphical programming environments, we now examine the use and implementation of data types that underlie such systems. This section presents a series of example definitions of complex data types. Each successive example represents a class of data type which is more expressive. The final class is adequate to represent text programming languages, which we then generalize further to represent graph-based languages. The first few examples are simple and well-known.

Basic complex data type

We begin with the simplest form of complex data type, one in which primitive types are composed and can be manipulated and as a unit. For example, the coordinates of objects in 2D-space might be represented through the following composite type definition.

```
COORDINATE == X-coordinate   : INTEGER
              Y-coordinate   : INTEGER
              Label          : STRING ;
```

Such a type can be instantiated for all combinations of values of each of its component primitive types. Thus the set of values (instances) of the type COORDINATE is the set of values of the cartesian product of the domains INTEGER x INTEGER x STRING. Most modern programming languages provide at least this level of type definition facility. The tags X-coordinate, Y-coordinate, and Label name each component. (We use the term “tag” here because other terms such as member, field, slot, symbol, attribute, are associated with a particular discipline)

Recursive types I: lists

A next higher level of expressiveness in complex types is to allow type definitions to be *recursive*. In this case the definition of a new type includes the type being defined. The simplest example of a recursive type is a list. For example the type definition

```
LIST ==       Label          : STRING
              Tail           : LIST ;
```

is a composite type in which one of the fields contains a value also of type LIST. Instances of this type are lists whose elements are strings.

At the implementation level, it is possible to make use of *pointers* to represent instances of this type. A pointer is a memory location which contains a value which identifies another memory location. Instead of assigning the value of a primitive type to the component field *Tail*, a pointer which is the address in memory of the first element of the tail of the list is assigned to the field. A picture of such a list instance might be the following

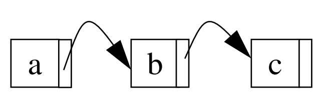

Figure 4. Implemtation of a List

This informal graphical depiction illustrates the concrete pointer-based implementation of an example list. A *dereferencing* operation must be used to access the tail. An imperative language such as C or Pascal requires a special annotation in the type definition (*e.g.*, *LIST) to indicate the presence of a pointer to a given type. Dereferencing and this annotation bring to the surface an aspect of the implementation which would ideally be hidden [9]. Functional languages (such as ML, Miranda, Pure Lisp, etc.) allow only the abstract view of such a type. Functions can be defined for manipulating lists in which the implementation is completely hidden. In the same spirit, the *algebraic* approach to specifying abstract data types (ADTs) allows the semantics of such functions to be defined via algebraic equations interrelating the functions [10]. The intent behind the algebraic specification of abstract data types is to characterize the behavior of a data type without needing to appeal to any particular implementation (pointer-based or otherwise).

A recursive type does not restrict the size (length) of the instances. This introduces the issue of creating structures whose size is only known at run-time and may vary during the course of program execution. This in turn requires the support for dynamic storage allocation and deallocation, but this issue will not be pursued in this paper.

Recursive types II: trees

A simple generalization of the list type is the tree type. Trees can also be conveniently represented using recursive type definitions. For example the following defines a binary tree type

BTREE ==	Label	: STRING
	Lbranch	: BTREE
	Rbranch	: BTREE;

Instances of such a type can similarly be implemented with pointers. An example is illustrated here.

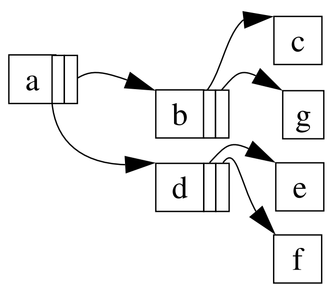

Figure 5. Implementation of a Binary Tree

This is an instance of the concrete implementation of a binary tree. Again, the definition of such a type and the manipulation of its instances can be easily represented in functional languages and as ADTs. These representations can be naturally extended to trees with other branching factors. So called *general trees* in which the branching factor varies can also be handled. In order to represent trees where the branching factor varies, a pointer based implementation would require *lists* of pointers, e.g:

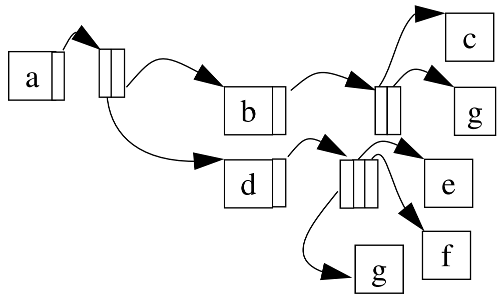

Figure 6. Implementation of a General Tree

(Here the represents some implementation of a list.)

Recursive types III: tree structures with variants (grammars)

A different extension of expressiveness is to allow a composite type to contain different combinations of types. These are known as *variants* or *constructors*:

EXPR ==	Fexpr	:(Flt1:FLOAT Op:OPERATOR Flt2:FLOAT)
|	Iexpr	:(Int1:INTEGER Op:OPERATOR Int2:INTEGER);

In this example an instance of type EXPR can be either the combination of two FLOATs and one OPERATOR *or* two INTEGERs and one OPERATOR. The variants are distinguished by their own tags, Fexpr and Iexpr. This extension provides a type definition language with the same level of expressiveness as context free grammars, the notation being equivalent to BNF. Each type definition corresponds to a production and each direct or mutual recursion between type definitions corresponds to the recursive use of productions in a grammar.

A data type of this class can be used to specify a set of instances which correspond to the parse trees of syntactically correct expressions of a given language. A program thus corresponds to a tree instance of such a type. Indeed, the specification meta-languages for automatic programming environment generators (mentioned in Section 1.0) allow the designer to specify a data type from this class.

Restricted shared structures

A different form of generalization is to allow instances of recursive types in which parts of the instance structure are *shared*. It is possible to take a type definition that is of the same form as the previous one for binary trees,

BDAG ==	Label	: STRING
	Branch1	: BDAG
	Branch2	: BDAG;

but create an instance for the type which is not a tree but an acyclic *graph*. The diagram below illustrates an example (again, at the level of implementation with pointers).

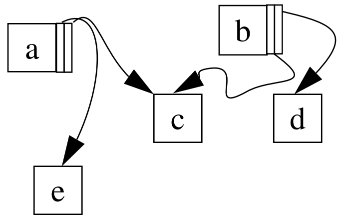

Figure 7. Implementation of a Shared Structure

In admitting such structures to be legal instances of the type, one encounters the first problem for the formal specification of changes to such instances. If the value (in the sense described earlier) of the DAG rooted at **a** is modified by modifying **c** it has the *side effect* of modifying the value of the DAG rooted at **b**. The component labelled **c** is shared by both the instance rooted at **a** and the one rooted at **b**. Admitting side-effects violates the requirement of referential transparency in a declarative formal description. Instances implemented as DAGs encounter the problem of side effects through structure sharing. This problem of structure sharing occurs whenever a component object has more than one "parent".

Cyclic shared structures

A further step in increasing expressiveness is to allow *cyclic* structures. Using the same form of type definition:

BDG ==	Label	: STRING
	Branch1	: BDG
	Branch2	: BDG;

the instance illustrated here could be admitted:

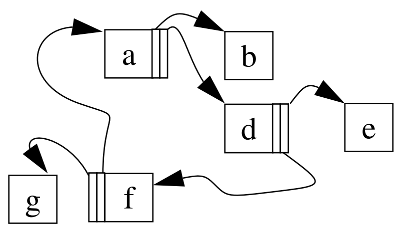

Figure 8. Implementation of a Cyclic Structure

In order to admit the structure that is *implemented* with pointers as illustrated, it is necessary to remove a further restriction. In the previous categories only the type *definition* was recursive. In the case illustrated above the *instance* of a type can also "contain" itself as a component. This is recursion in the instance rather than in the type - a *recursive instance*, though the term *cyclic* or *circular structure* is more commonly used. Circularity can also be thought of as a special case of sharing in which the shared item and sharing item are the same [11]. The cycles in such structures cause further problems for declarative formal descriptions of structure manipulation. This is because they rely on the ability to traverse a well-founded data structure, which a cyclic graph is not. For example, a naive structural equality test of two cyclic instances will not terminate because it will get caught in a loop.

The use of these unrestricted graphs are quite common in interactive systems which need a way to represent interconnection. For example, hypertext systems allow links between components of a document without restriction so that the structure of the document as a whole can be an arbitrary graph. CAD/CAM modelling packages contain data structures for 2-D and 3-D geometric modeling. These models have cyclic pointers to represent the topological properties such as the interconnection between faces, edges, vertices. Graphical user interface toolkits allow an interactive graphical interface to be created by composing small interactive objects (“widgets”) into a complete interface for given application. The neighboring relationships between such objects are often captured with inter-object references which may be cyclic [12].

4.0 Specifying data types for graph based languages

The need for representing connected objects, as we described earlier, also exists in graph based languages. We now present a class of datatype that is suited to characterizing the data structures needed to represent such languages within a programming environment. The extension of expressiveness of data types is obtained by combining variants with shared structures. For example, the following datatype could be used define the form of graphs that represent Petri Nets.

```
PETRINET     ==      Nodes        : NODE**
                     Links        : LINK**;

NODE         ==      Pn           : PLACENODE
                 |   Tn           : TRANSNODE ;

PLACENODE    ==      Label        : STRING
                     NumTokens    : INTEGER
                     Inputs       : PNLINK**
                     Outputs      : PNLINK** ;

TRANSNODE    ==      Label        : STRING
                     State        : STATE
                     Inputs       : PNLINK**
                     Outputs      : PNLINK** ;

PNLINK       ==      Label        : STRING;
                     LinkPn       : PLACENODE
                     LinkTn       : TRANSNODE
```

(Here we have made use of the additional annotation, the “**” indicates a *set* of elements of a given type, taken from our specification language Glide.) A typical instance of this type might be displayed to a user in an interactive graphical programming interface as the box shown here :

LINK : **b, d, f**

TRANSNODE: **c**

PLACENODE: **a, e**

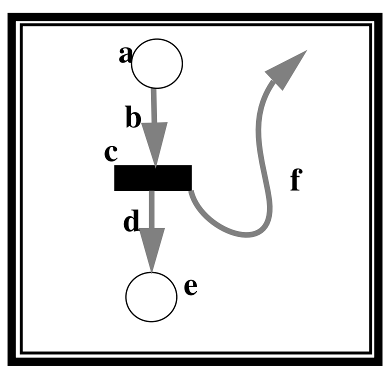

Figure 9. Displayed Petri Net

Note that the LINKs **b, d,** and **f** are graphical elements in the same way as the nodes **a**, **c** and **e**. The interconnection of components is represented through mutual type recursion in the definition of the types PNLINK, TRANSNODE, PLACENODE, and admitting recursive instances of these types. A pointer implementation of the interconnection of the components is shown in this diagram:

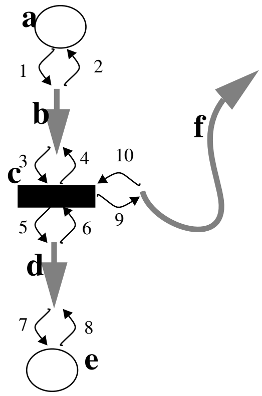

Figure 10. Implementation of a Petri Net

This diagram would not be displayed to a user of the system. It is shown to illustrate how the *interconnectedness* of graphical elements can be implemented through pointers and to show the relationship between the original diagram and the figure below:

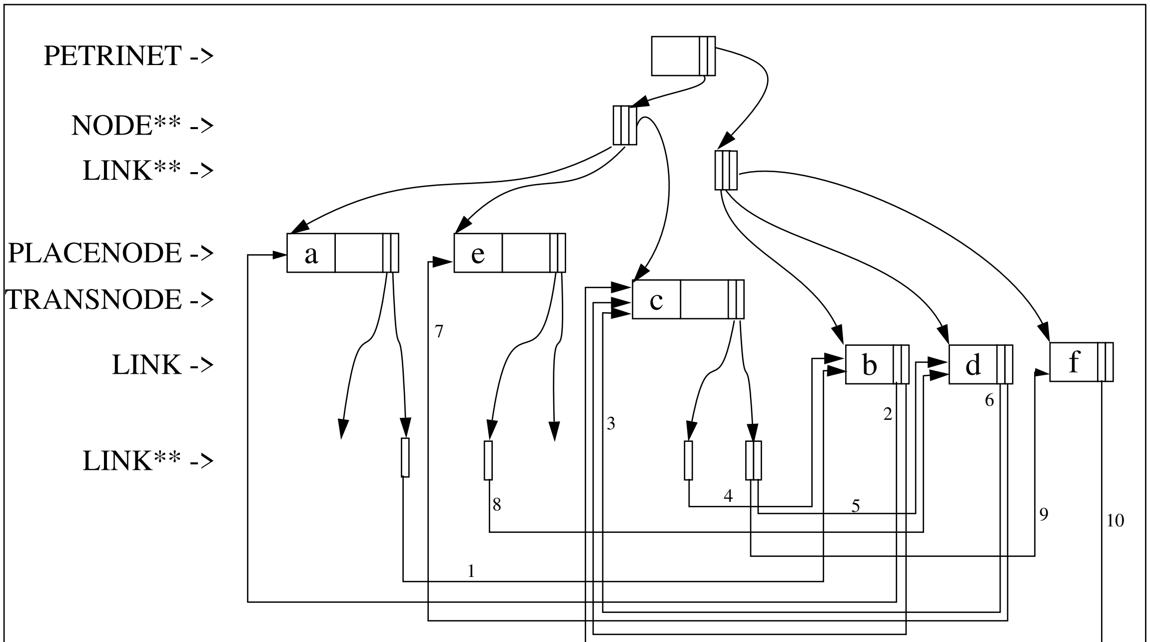

Figure 11. Implementation of a Petri Net

This figure is the direct analogue of the diagrams of lists and trees and graphs in Section 3.0. The numbers label the same pointers. Here the [|] represent some implementation of a set.)

Path Expressions

Portions of this structure can be identified by "path expressions" which walk along one or more paths in the underlying graph. For example, if pn is this instance of the PetriNet type then:

pn.Nodes	is all the nodes in the net
pn.Nodes.Pn	is all the place nodes in the net
pn.Links	is all the links in the net
pn.Links.LinkTn	is all the transition nodes attached to any link in the net.

There is no explicit notion of references or pointers in these path expressions; there is no difference between walking along the straight edges or curved edges in the figure above.

Shape Predicates

There is a price to paid for this elegance. Though the type specification does identify the desired program instances it also admits ones which are not. It is thus necessary to exclude some shared and cyclic instances. For example the following diagram:

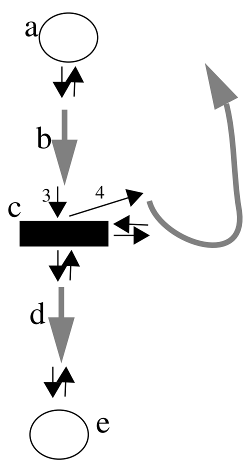

Figure 12. Incorrect structure instance

does not violate any of the type composition prescribed by the type definitions yet it is clearly unwanted because pointers 3 and 4 are not correctly "paired". For this reason it is necessary to provide an external means to characterize the particular sharing and cycles that are legal and separate them from those which are not. The mechanism we use for this we call *shape predicates*. Shape predicates make use of path expressions to identify these desired cyclic/shared instances. The following two shape predicates capture the desired cycles for the PETRINET type ($\oplus$ is exclusive or):

> "For all links in the petri net, a link is either a member of the output links or a member the input links of the transition node it is attached to, but not both"
>
> $\forall$ l:pn.Links [l $\in$ l.LinkTn.Outputs $\oplus$ l $\in$ l.LinkTn.Inputs]

> "For all links in the Petri Net, a link is either a member of the output links or a member the input links of the place node it is attached to, but not both"
>
> $\forall$ l:pn.Links [l $\in$ l.LinkPn.Outputs $\oplus$ l $\in$ l.LinkTn.Inputs]

Again, pn is a variable representing any instance of the complex type PETRINET. The path expressions are used in the predicates to identify components or sets of components. These predicates do not preclude having nodes and links which are yet not attached. This is convenient from a programming environment point of view since they can be added into the structure (as members of pn.Nodes or pn.Links) and appear on the screen for later attachment by a further editing operation.

The predicates for this example are quite simple, but graph-based languages with more complex interconnection structure, for example, ones with nodes which have many “ports”, or ones hyperlinks (links which can connect more than two nodes) require more complex shape predicates, but the path expressions mechanism allows these shapes to be specified in a concise way.

The key reason for wishing to represent the interconnection of components in this way is that by using a direct generalization of the way the text language structure is represented, it is possible to smoothly integrate the representation of graphs and text in one uniform formalism, and hence to view graph-based visual languages as a generalized form of text based languages. Since the types described here subsume the recursive types with variants (III), structured text that can be described by a context free grammar can be naturally included in a definition. For example the current description of Petri Nets can be extended to accommodate, say, a textual condition attribute by simply adding the BNF definition of the condition syntax to the current description:

```
TRANSNODE ==    Label        : STRING
                State        : STATE
                Condition    : CONDEXPR
                Inputs       : PNLINK**
                Outputs      : PNLINK** ;
CONDEXPR ==     A1           : ARITHEXPR
                RO           : RELOP
                A2           : ARITHEXPR
RELOP ==        r            : ‘=’ | ‘>’ | ‘<‘ | ‘=<‘ | ‘>=’
```

5.0 Existing approaches to types with sharing and circularity

Having shown how the need for sharing and circularity arises in generalizing data type definitions so that they include data structures that underlie graphical programming and described our solution, this section shows that the problem of formalizing shared data structures arises in many other areas and indicates how they have been addressed. We specifically examine functional programming, abstract datatype specification, and data base models.

5.1 Functional programming

We begin by examining the problem of shared and cyclic types in functional languages. Two functional languages are used as examples, ML [13] and Miranda [14]. These languages that make a point of being declarative and founded on the well understood semantics of the theory of functions.

ML - As its name suggests (Meta Language), one of the primary purposes of ML is to describe languages, and thus it would seem well-suited to applications centered on language structure. The following is the collection of recursive datatype definitions that corresponds to the Petri Net type that was specified in the previous section in GLIDE notation.

```
datatype   PetriNet        = PETRINET       of Node list
                                             * PNLink list      and

           Node            = P              of PlaceNode
                           | T              of TransNode        and

           PlaceNode       = PLACENODE      of Id
                                             * NmTk
                                             * I
                                             * O                and

           TransNode       = TRANSNODE      of Id
                                             * Ste
                                             * I
                                             * O                and

           Id              = ID             of int              and

           NmTk            = NMTK           of int              and

           I               = INPUTS         of PNLink list      and

           O               = OUTPUTS        of PNLink list      and

           Ste             = STATE          of bool             and

           PNLink          = PNLINK         of PlaceNode
                                             * TransNode            ;
```

(The **and** is the separator in ML that allows a collection of mutually recursive types to be defined. The "type operator" **list** is used because the direct analogy to the Glide ** annotation, *i.e.* **set** is not available as a primitive in ML.) This definition is accepted by the ML compiler. The problem with this complex type is that instances that are cyclic (and would thus correspond to the diagrams of the Petri Net such as the one above) can *not* be created. Only instances which correspond to trees (none of which correspond to actual Petri Nets) can be instantiated.

To preserve referential transparency required in ML, an instance of a type can be created but cannot be changed. A small change to part of an instance must be modeled by constructing a complete new copy of the structure with the small change. The problem of side effect in shared structures does not arise since update-in-place is not available, but the consumption of time and space due to copying structure for every change soon becomes prohibitive. A further consequence of referential transparency is that it is not possible to create cyclic structures since one of the components of a cyclic structure is itself (which, by definition, hasn't been created yet). These problems can also be viewed as a consequence of the strict functional semantics of the language.

It is because cyclic structures are so useful in real programming problems, however, that ML does include an imperative extension, "reference types", which is similar to pointers. The **ref** keyword is used to specify a reference type. The following changes to the previous definition is how the Petri Net type *could* be defined in ML.

```
datatype .......
          I              = INPUTS          of ref PNLink list  and

          O              = OUTPUTS         of ref PNLink list  and
.....
          PNLink         = PNLINK          of ref PlaceNode
                                            * ref TransNode ;
```

The only difference between this definition and the previous one is the addition of the **ref** qualifiers in the definitions of the INTPUTS, OUTPUTS, and PNLINK types. This "reference type" extension to ML can be used in conjunction with destructive assignment (`:=`) to create cyclic structures. This requires a notion of change of state and thus lies outside the referentially transparent functional model. In ML support for imperative programming is closely associated with the use of reference types. It is not possible to create cyclic instances of recursive types without using **`ref`**.

Miranda - Miranda is a lazy functional language with an elegant minimalist syntax. The analogous definition of the Petri Net type in Miranda is shown here.

```
petrinet        ::=     Makepetrinet      [pnnode]
                                          [pnlink]

pnnode          ::=     Makepnnodepl      placenode
                |       Makepnnodetr      transnode

placenode       ::=     Makeplacenode     nodeid
                                          nmtk
                                          inputs
                                          outputs

transnode       ::=     Maketransnode     nodeid
                                          state
                                          inputs
                                          outputs

nodeid          ::=     Makenodeid        [char]

nmtk            ::=     Makenmtk          num

inputs          ::=     Makeinputs        [pnlink]

outputs         ::=     Makeoutputs       [pnlink]

state           ::=     Makestate         bool

pnlink          ::=     Makepnlink        placenode
                                          transnode
```

(Again, we are here using the square bracket, [ ], Miranda list constructor since sets are not available as a primitive.) In contrast to ML, cyclic instances of this type *can* be created in Miranda, without requiring an imperative extension. There are two ways to explain why this works in Miranda. At the abstract level, Miranda is a lazy functional language, so a value can be used before it is created, and hence it is possible to create an object out of itself (the cyclic case). This is achieved at the implementation level, because all values in Miranda are accessed by always dereferencing from a pointer. Thus the pointer to an object can be created and used before the object itself is created. Structures which are implemented as cycles in Miranda are viewed as "infinite" structures, because at the functional level abstraction they can be regarded as such - the underlying pointer structure is not visible and not accessible. Traversal operations (such as printing of an infinite structure) do not terminate.

In contrast to both Miranda and Glide notations, ML breaks symmetry when the **`ref`** construct is used. The representation of PNLINKs and PNNODEs as components of PETRINET is different from that of connected PNLINKs as "components" of a PNODE.

The issues of sharing and cyclic structures are well know problems of functional languages. A proposed solution that avoids "recreating the entire data structure" after any change to a data structure is described in [15]. The proposed solution "hides" the change of state as auxiliary heap data structure which must then be passed as an extra argument between function invocations. It is possible to reduce the amount of copying by carefully sharing structure of the heaps associated with each function invocation. This is still far from and ideal solution since it simply defers the problem of structure re-creation to this auxiliary structure.

5.2 Abstract data types

This section examines work in formal specification of abstract data types which is similar to the work in functional languages in that both strive for concise declarative abstract descriptions of data types and the operations. The problems of shared and cyclic structures have also been investigated here:

Levy [11] - The thesis of M.R. Levy has directly addressed the problem of extending the ideas of algebraic specification of abstract data type data types to incorporate shared and circular structures. Two solutions are proposed to providing an abstract description: (i) providing an explicit algebraic characterization of the use of references and assignment by relating references to a special notion of congruence and (ii) by relating circular structures to infinite objects in continuous algebras. Sets of equations are used to characterize sharing in instances of a type.

Moller [16] - The paper by Moller also addresses the problem of shared and cyclic structures from an abstract mathematical perspective on algebraic specification. He proposes a solution to the inability of algebraic formalisms to describe graph structures by characterizing a graph as a repeating pattern in an infinite tree. This resembles the Miranda view of graph structures as infinite structures. Solutions of Moller and Levy require a high degree of comfort with these algebraic techniques.

Data Abstraction Languages CLU [17] - CLU is a language specifically designed for data abstraction. The language distinguishes between **type**s and **cluster**s, the latter being used to define abstract data types with their of associated operations. Cluster definitions can be recursive, but type definitions can not. This was regarded in hindsight as significant omission in the design of the language.

Graph Types [18] **-** A recent paper by Klarlund and Schwarzbach also examines the problem of cyclic structures. The authors provide an interesting way of formalizing at least a subset of graph structures as abstract data types. The key ideas are to first identify an underlying spanning tree of a graph structure and then to use a simple language based on regular expressions to define the remaining links which span the tree. These are termed “routing expressions”. The authors demonstrate that this technique makes it possible to describe graph structures such as doubly linked lists, threaded trees, and binary trees with the leaves linked to the root. They note however that some structures are not amenable to this form of graph description. The graphs defined by such routing expressions are functions of the underlying tree, i.e. for a given set of routing expressions and a given tree it is not possible to have more than one graph for. A disadvantage that this technique shares with ML references is that symmetry is broken since the one path to a node is on the spanning tree is distinguished from the others described by routing expressions. The general point conveyed by this work is that it is possible to identify subclasses of general graphs which, because of some unique property, can be characterized in a simple way - in this case through regular expressions on paths in a tree.

5.3 Database models

The problem of shared and cyclic structures also arises in providing formal models of network databases. For example, the work Gangopadhyay [19] provides a formal model of network data bases (“DML”) in which the same problem of shared mutable data structures is explicitly considered. Here, the solution used is to distinguish between the concept of *identity* of a data item as distinct from its *value*. The identity of a shared component of a data structure does not change, only the value associated with it. An auxiliary look-up data structure (the “*type-state*”) is used to associate identities with values. This auxiliary structure is described using a state transition model. This solution is thus similar to the Barton’s solution in functional programming.

The issue of cyclic structures also arises in object oriented data models. Klarlund and Schwarzbach note the close relationship between the formal specification of graph types and formal models of object oriented programming because of mutual reference between objects. The following comments by Stroustrup in the preface of [20] also indicates the importance of the issue:

> *“One of the most powerful intellectual tools for managing complexity is hierarchy, that is, organizing related concepts into a tree structure...*
> *...Naturally, this organization has its limits. Sometimes even a directed acylic graph seems insufficient for organizing concepts of a program; some concepts seem to be inherently mutually dependent. If a set of mutually dependent classes is so small that it is easy to understand, then cyclic dependencies need not be a problem.“*

5.4 Other

Graph Grammars - The extensive work on “graph grammars” is also related to the problem of shared and circular structures. The work in this area takes a somewhat different approach to characterizing graph structures. Classes of graphs are described by collections of graph rewriting rules. A graph grammar is a collection of rules to change graphs by first matching and then altering parts of the graph structure. These rules are often graphically depicted. The problem of circular structures is hidden by dealing with pictures of graphs directly, but the difficulties encountered in attempting to find a concise standard textual notation for graph rewriting is an indication of the underlying problem of cyclic structures.[21]

Paralleization Compilers and Complex data structure -The issue of structure sharing is also of concern to designers of parallelisng compilers, where *aliasing,* the anonymous referencing of the same location in memory can preclude paralleliztion. Hummel, Hendern and Nicolau use a scheme that is the dual of Glide shape predicates: They use predicates to define where sharing that does *not* occur. The reason for this is that in paralelizing programs the goal is to identify those parts of a structure which are not shared, do not carry dependencies, and therefore the code accessing them can be safely executed in parallel.[22]

6.0 Conclusions

This paper has argued that there is a strong relationship between graph-based visual programming and data structures with sharing and cycles. Programming languages can be viewed not just as structured strings, but also as complex data objects. This view of programs as complex objects was shown to be even more apparent in graphical programming. The paper argued that data types can be used to characterize graph-based visual programming languages if instances with sharing and cycles are admitted. This approach then requires and an external means for fully specifying these structures and the notion shape predicates with path expressions was introduced for this purpose. The paper demonstrated how widespread the issue of sharing and cycles in data structures is in various areas of computer science research, and compared the approaches in each of these areas to the one proposed for specifying graph-based visual languages.

In closing, it is interesting to note that the reverse correlation of our argument is readily apparent. The computing literature is full of graph-based diagrams and computer scientists make heavy use of graph-based sketches because in order to communicate the complex structures involved in the discipline, these explicit diagrams of the structures are more comprehensible than their textual or verbal descriptions.

7.0 References

1. *Proceedings of the IIEE Workshop on Visual Languages*: 1990, *October Skokie Illinois; Proceedings of the IIEE Workshop on Visual Languages* 1991, October Kobe Japan; 1994 (10th) *IEEE/CS International Symposium on Visual Languages*, October 1994.

2. D. Harel. On visual formalisms. *Communications of the ACM*. May 1988

3. R. Akscyn, E. Yoder, D. MacCracken. The data model is the heart of interface design. *Human Factors in Computing Systems (CHI)* 1988.

4. M. van de Vanter, S. Graham, R. Ballance. Coherent user interfaces for language-based editing systems. *Int. J. of Man-Machine Studies*. Number 37 1992.

5. T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM*. September 1981.

6. D. Notkin et al. The gandalf project. *The Journal of Systems and Software*. Volume 5, Number 2, May 1985.

7. Interactive Development Environment's "Software Through Pictures". 595 Market Street San Fransisco CA94105. System reviewed by P. D. Stotts in Tools Review: 'Software through Pictures' from IDE Inc. in *Journal of Visual Languages and Computing* (1993) 4, 201-209.

8. *Graph Drawing '93*. ALCOM International Workshop on Graph Drawing and Toplogical Graph Algorithms. September 1993. Paris.

9. C.A.R. Hoare. Recursive data structures. *Intl. Journal of Computing and Information Sciences*. Vol. 4, No. 2 1975.

10. J. Guttag and J. Horning. *The Algebraic Specification of Abstract Data Types*. *Acta. Inform.*, vol. 10. 1978.

11. M.R. Levy Data types with sharing and circularity *Ph.D. Thesis* Dept. of Computer Science, University of Waterloo 1978.

12. A. Weinand, E. Gamma, R. Marty. Design and implementation of ET++ a seamless object-oriented application framework. *Structured Programming*, Vol1 No. 2 1989.

13. L. Paulson. *ML for the Working Programmer*. Cambridge University Press.

14. R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall 1988.

15. W Burton, HK Yang. Manipulating multilinked data structures in a pure functional language. *Software Practice and Experience* Vol 20 (11) 1167-1185 Nov 1990.

16. B. Moller On the algebraic specification of infinite objects - ordered and continuous models of algebraic types in *Acta Informatica* 22 p.537-578 1985

17. B. Liskov. A history of CLU *Proceedings of the Second ACM SIGPLAN History of Programming Languages Conference (HOPL-II)*. ACM SIGPLAN Notices. V28, Number 3, March 1993.

18. N. Klarlund, M. Schwartzbach. Graph Types. *POPL* 1993.

19. Dipayan Gangopadhyay. *A formal system for network databases and its applications to integrity based issues. Ph.D. Thesis* University of Texas, Dept. of Computer Sciences 1983.

20. B. Stroustrup. *The C++ programming language 2nd ed.* Addison-Wesley, 1991.

21. *Graph Grammar Workshops*: Springer-Verlag, 1990 Ehrig, Kreowski, Rozenburg eds LNCS 532; 1982 Goos, Hartmanis eds LNCS 153; 1978 Goos, Hartmanis eds LNCS 73.

22. J. Hummel, L. Hendren, A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. *International Conference on Parallel Processing* 1992.

23. B. Courcelle Graph rewriting: an algebraic and logic approach, in *Handbook of Theoretical Computer Science*. 1990

24. A. Poulovassilis, M. Levine. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Informations Systems* Vol 12 No. 1 Jan 1994 pages35-68.