Prograph: a step towards liberating programming from textual conditioning

P.T. Cox § F.R. Giles * T. Pietrzykowski §

§ Technical University of Nova Scotia, Halifax, Canada
* Acadia University, Wolfville, Canada

Abstract

We present a critique of textual programming languages and software development environments, linking them to the development of hardware and discussing their connection with natural languages and mathematical formalisms. We then outline criteria for modern integrated programming languages and environments based on the use of graphics. These principles are illustrated by a description of the pictorial, dataflow, object-oriented language Prograph and its implementation. Finally, possibilities for further use of pictures in programming environments are discussed.

1. Introduction

Our purpose is to investigate the contributions that visuality can make to the programming environment as a whole, rather than just to programming languages. We must divide the uses of visuality into two categories: general uses such as windows and menus, and specific ones, such as representing algorithms and data with pictures rather than text. To this end we will first investigate the origins of textual programming, and attempt to identify appropriate and inappropriate uses of text. We will propose pictorial alternatives to the latter, illustrated by the pictorial programming language Prograph, implemented in a rich software development environment.

The development of textual programming languages and environments has paralleled that of computer hardware, and therefore has been influenced by the organisation of this hardware. As a consequence, languages tend to be oriented towards simple, character-based input and output, and to be sequential in structure. They also rely on a combination of mathematical formalisms and natural languages, with the result that their syntax is complex and unforgiving, enforcing restrictive structure on algorithms. In recent years, user interfaces have been revolutionised by the advent of sophisticated high resolution graphics, providing possibilities for rich interactions with the user. Existing textual languages and environments are deficient for such hardware for two reasons: first, they do not facilitate the design and programming of pictorial interfaces, and second, they do not take advantage of pictures to represent algorithms and data. On the other hand, for some uses text is superior to pictures: namely, for comments, identification of program elements and certain algebraic formulae.

2. Evolution of programming environments

2.1 Textual programming

Textual programming languages developed from a desire to make the representation of algorithms as precise and readable. Therefore they have been influenced by languages used for other purposes, namely natural languages and mathematical formalisms such as algebra, first order predicate calculus, and lambda calculus. A further influence on the development of textual programming languages stemmed from the computing hardware and operating environments available for executing algorithms. The hardware imposed a certain structure on languages because of its von Neumann architecture, while operating environments were very crude, providing only the simplest character-oriented input and output facilities.

The natural language component of programming languages originated in attempts to describe the functions of the hardware. For example, variables were used to symbolise addresses, keywords such as goto and if-then were used to describe unconditional and conditional branches. Later, some higher level languages discarded natural language structures in favour of mathematical formalisms, like lambda calculus in Lisp and predicate calculus in Prolog.

This historical development of programming languages has several unfortunate consequences. The facilities these languages provide for describing algorithms correspond more closely to how computers operate than to the cognitive or perceptual processes of the programmer. Because the medium of expression is text, which is inherently one-dimensional, algorithms are forced to be sequential. This sequentiality unnecessarily constrains the thinking of the programmer by forcing him or her to linearly organise every program, whether or not the algorithm requires it. The linearity of text has a similar detrimental effect on the programmer's conceptions of data, since even multidimensional structures such as graphs or arrays have to be represented in some linear fashion. These restrictions were appropriate for machines with single main processors, but become an obstacle in new parallel architectures.

Textual programming, because of its natural language ancestry, has inherited a complex syntax. Unlike their natural counterparts, however, the syntax of programming languages is inflexible and unforgiving, forcing the programmer to deal with small syntactic details, rather than the important concepts of algorithms.

The first attempts to describe control in textual languages relied on the free use of gotos, which were soon discovered to produce unreadable and error-prone programs. As a result of negative experiences with these primitive controls, structured programming was introduced, enforcing a deeply nested organisation of programs. Unfortunately, the accompanying design methodology, based on step-wise refinement of pseudocode, can be applied only to a certain extent since early decisions about control structures can be invalidated by deeper design issues. For example, when a while loop is introduced, its control variables may be unknown since they originate in the body of the loop which has yet to be designed.

Textual programming languages are based on coding information in one-dimensional strings to be understood by computers. Of course, these strings should also be readily understood by humans and should therefore imitate natural languages, in which only a small number of the many combinations of symbols are meaningful. Consequently the information density of programs in such languages is very low [1].

Another well known and extensively discussed defect of textual programming languages is their use of variables [2]. Variables originated as symbolic addresses, and have since been cast into various conflicting roles. They are used both as transmitters of local data within procedures, and as global data repositories, leading to the notion of variable scope, which adds to the confusion and unreadability of textual programs. A further problem arises from the fact that the name of a variable is used for two purposes, to identify the variable and to provide mnemonic information. These two uses are in conflict, since for mnemonic purposes names should be long, but to minimise typographical errors they should be short. Finally, the concept of variable is alien to languages based on ideograms, such as Chinese [3].

Textual languages are also deficient in expressing the concepts of object orientation [5, 7, 10, 13]. For example the class hierarchy, a two dimensional graphical construct central to this programming paradigm, can be represented in text only by some kind of linear coding which detracts from the simplicity of this concept. Furthermore, referring to objects requires variables, creating a misconception that objects need names, and that the name of the variable is the name of the object. Objects, however, are
anonymous by nature, their identification depending on the values of their attributes.

Describing data in textual languages has shortcomings too. Simple values such as numbers and strings are naturally represented by text and therefore cause no problems. However, structured data built from records and pointers can be represented only indirectly in text, by some linear coding. Most languages allow the simple component of structures to be viewed, while others provide coding for a limited range of structures, such as lists and terms in Lisp, Prolog and Smalltalk. Even in cases where such codings are available, however, they usually rely on mathematical formalisms such as functional expressions, and do not necessarily represent the underlying semantics. For example, a data structure which can be displayed as a directed acyclic graph, may be meant to represent the structure of some physical object, such as a mechanical device. Hence, even though a language provides a representation, it may be removed by many levels of abstraction from the intended semantics.

As mentioned above, the evolution of programming languages has been heavily influenced by the simple character-oriented input and output facilities of early hardware. This form of communication has remained essentially unchanged through the evolution from primitive punched card and tape devices to time-sharing via remote ASCII terminals. A fundamental change over time, with the single-user computer systems, has been in the resolution graphics and pointing devices, leading to a revolution in the design of operating systems which now have interfaces relying on windows, icons and menus. This originated from research at Xerox PARC.

### 2.2 The role of pictoriality in programming

Visuality has a long association with computers and programming. Before the advent of digital computers, analogue machines were widely used for a variety of tasks such as the solution of differential equations. These machines were programmed in a pictorial fashion by designing diagrams to specify control structures. Terms and commands were used as part of the programming process to control the machine's actions. With the development of the digital computer, a new method of programming was introduced, which was based on the use of textual languages. However, with the advent of graphical interfaces, the use of graphical languages began to increase.

It is important that the development of pictorial languages and programming environments should not be hampered by the traditional approaches outlined in section 2.1. We will therefore describe in general terms the characteristics that pictorial programming environments should have.

It is clear that the components of a programming language should be a natural extension of the environment provided by the host computer. Consequently, a pictorial language should employ all the usual interface devices such as windows, menus and icons. Similarly, in the programming environment provided by the implementation of the language, the user's actions should evoke a "predictable" response. For example, if a double click on an icon in the host operating system opens a window, this convention should be followed in the programming environment, so that double-clicking an icon representing a program element should open a window revealing the internal structure of that element.

A pictorial programming language should represent the underlying concepts in a natural way as possible. For example, representing a class hierarchy should be accomplished by displaying it as a tree or graph. Also, devices necessary to accomplish certain goals in textual languages should not be propagated into pictorial ones: for example, linearly coded nested structures, keywords, punctuation, sequentiality, and data transmission by local variables.

In textual languages, program elements have no physical manifestation, but are referred to only by abstract identifying names. For example, the name of a procedure used in a call does not itself carry any features to identify the reference as a procedure. This information is usually declared somewhere else in the code. The context and the name of the caller determine the context of this particular occurrence of the name. Even worse is the situation in some languages which use context only without declarations to determine the characteristics of the entity referred to by a name, FORTRAN for example. In pictorial languages, however, the icon for a program element can communicate these characteristics, while the name is used only to distinguish it from others of the same category.

Pictoriality can also play an important role in the representation of data. As mentioned above, most languages can directly display only simple data. Using pictures, however, complex data may be represented at any level of abstraction, from arrays of machine addresses to any picture that captures the semantics appropriate to the particular application. For example, an instance of a class "Window" could be represented as a window on the screen, or an instance of a class "bicycle" by a picture of a bicycle.

## 3. Prograph

In this section we describe the pictorial programming environment Prograph which was designed according to the above principles. We will describe how Prograph represents the main concepts of object-oriented programming, classes, attributes and methods and how a method's representation defines the semantics of its dataflow, data driven execution. In particular, we indicate how parallelism, sequencing, iteration and conditional constructs are modelled in Prograph. A more complete description of the language features of Prograph is given in [8]. The design of Prograph developed from research on the role of pictures in describing functional programs [19]. In further research it was discovered that pictoriality could be used to replace the familiar nested control structures by multiplexes and Prolog-like case structures, and to naturally represent object-orientation with single inheritance [8, 9].

### 3.1 Classes

A class in object-oriented programming provides a mechanism for defining a data type with attributes and methods which apply to objects of that type [9]. Classes in Prograph are organized in hierarchies: the class structure of a Prograph program is displayed in the Classes window. This window contains a visual representation of the current forest of classes. Each class is represented by an icon and each parent-child class relationship is represented by a line from the parent's icon down to the child's.

There are two types of classes in Prograph: system classes and user classes. Prograph has a predefined hierarchy of system classes which provide user-interface features such as windows, menus, dialog boxes and lists for a Prograph application. An instance of a system class will have system attributes and an appropriate behavior when the Prograph application is running. Also, Prograph has primitive methods which apply to instances of particular system classes, such as methods to cut, copy and paste strings of text between a text instance and the Macintosh "clipboard". System classes are distinguished pictorially from user classes by a double bar at the bottom of the class icon.

The example we use throughout this paper is a Prograph application which allows a user to browse a list of literature reviews such as those of the ACM Computing Reviews. The user can import reviews from a text file, insert, delete and modify review contents and examine individual reviews. The
A Prograph method is either a class method or a universal method belonging to no class. Universal methods include built-in Prograph primitives. Class methods are represented as named icons in a methods window for the class and user-defined universal methods are represented by icons in the Universal window. A method consists of a sequence of cases, where each case is a dataflow structure, consisting of data inputs, data outputs, a set of operations and connections between them.

The definition of a case is illustrated graphically within a window for the class. Input into the case is indicated by roots on an input bar at the top of the case's window, and output by terminals on a bottom output bar. The input and output arities of a case are its numbers of roots and terminals respectively, and all cases of a method have the same input and output arities.

An operation within a case is represented by an icon containing the name of the operation. Input and output data for an operation are defined by terminals and roots at the top and bottom of the operation's icon. Operations within the case are connected by data links which carry data from the roots of operations to terminals of other operations. An operation is analogous to a procedure call and terminals and roots for an operation act as parameters of a procedure. However, unlike many textual programming languages, data values are untyped, and a terminal or root of an operation indicates the action of copying a data value between the calling operation and the associated method.

A root may be attached to several data links, but a terminal can be connected to at most one. The arity of an operation must be the same as that of the definition for the associated method.

The following table summarizes the different operations available in Prograph. Note that the type of each operation is indicated by the shape of its icon.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sample Call</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td>Copy value from terminal of calling operation</td>
</tr>
<tr>
<td>Output</td>
<td></td>
<td>Copy value to root of calling operation</td>
</tr>
<tr>
<td>Simple</td>
<td>Quicksort</td>
<td>Call user method Quicksort</td>
</tr>
<tr>
<td>Constant</td>
<td>256</td>
<td>Output constant 256 on root</td>
</tr>
<tr>
<td>Match</td>
<td>NULL</td>
<td>Next case if value on terminal is not NULL</td>
</tr>
<tr>
<td>Persistent</td>
<td>Reviews</td>
<td>Output value of persistent Reviews in root</td>
</tr>
<tr>
<td>Instance</td>
<td>Index Entry</td>
<td>Output new Index Entry instance on root</td>
</tr>
<tr>
<td>Get Attribute</td>
<td></td>
<td>Output value of attribute key of input instance on right root</td>
</tr>
<tr>
<td>Set Attribute</td>
<td></td>
<td>For left input instance, set value of attribute review to right input and output instance</td>
</tr>
<tr>
<td>Local</td>
<td></td>
<td>Call local user method check</td>
</tr>
</tbody>
</table>

The input operation of a method's case copies data values from the terminals of the calling operation to the corresponding roots of the input bar and the output operation copies data values from the terminals of the output bar to the roots of the operation.

A simple operation may be a call to a user-defined method, a Prograph primitive, or a method from the Macintosh toolbox. A call to a Prograph primitive is indicated by a bar on the bottom of the operation's icon and a call to Macintosh toolbox routine by bars on the top and bottom of the icon.

The constant operation is labelled by a constant which is produced as the value of its output. The constant's type must be one of the built-in Prograph types such as Boolean, integer, list and string.

Prograph has a mechanism for the permanent storage of data in objects called persistent. A persistent may contain data of any type and is retained during and between executions of Prograph methods. In our sample application, the list of reviews, Reviews, is a persistent. The value of a persistent is accessed within a method through the persistent operation which is represented by an oval icon labelled by the name of the persistent. If the persistent operation has a terminal, then, during execution, the value on the terminal will be assigned to the corresponding persistent. Similarly, if it has a root, the value of the persistent will appear on this root during execution.

A new instance of a class is generated by the instance operation which is labelled by the name of the class and outputs the new instance. Normally, the attribute values of the new instance will be the default values specified in the class attributes window for the class.

Attribute values of a class instance are accessed through the get attribute and set attribute operations. Each of these operations inputs a class instance and is labelled by the name of an attribute. The get attribute operation outputs the instance and the value of the named attribute for the instance. The second input of the set attribute operation is a data value and the operation outputs a copy of the instance with the named attribute set to the given value.

Finally, a local operation is a call to an inner, locally defined method which cannot be called by other operations.
Figure 3a depicts methods of class Index and the details of method Index/Sort, a method for sorting attribute entries of an instance of class Index. Note that prefixing the name of a method with the name of a class indicates that the method belongs to class. Here, Index/Sort is a method of class Index.

Execution of a Prograph method begins with a call to the method and the passing of input data to the input roots. The first case in the method’s case sequence then begins execution. Execution of a case follows the data-driven, dataflow paradigm determined from the graphical representation of the case. That is, an operation with input is not called within an executing case until it receives all its input data, and otherwise, operations for which there are no data interdependencies are considered to logically execute in parallel. Output from a case is available only when the case’s execution terminates. Normally, a case does not terminate until all operations within the case have executed.

Method Index/Sort of Figure 3a sorts a list of index entries and displays the sorted values. In the execution of case 1 of Index/Sort, the first operation is input, which copies the data value from the terminal of the calling operation. The expected input data is an instance of class Index. The next operation is a get attribute which inputs the Index instance and outputs it together with the value of the named attribute, entries, which in our sample application is a list of Index entry instances. The value of entries is then passed to method Quicksort which outputs the elements of entries sorted on attribute key. This output is passed to the set attribute operation which updates the value of entries. Next the class instance is passed to /Build value list, shown in Figure 3b, which constructs the new display list of entries. Finally, the case and therefore the method Index/Sort, terminates with no output.

Data values in Prograph are untyped. When case 1 of Index/Sort is defined, it is unnecessary to specify that the input is an instance of class Index. Indeed, in order for the execution of this method to successfully terminate, it is necessary only that the get attribute, set attribute, Quicksort and /Build value list operations succeed. The get attribute and set attribute operations will succeed on input of an instance of any class which has an attribute named entries. This is an example of late binding in Prograph.

Simple polymorphism in Prograph is illustrated by the selection of a method called by an operation. If there is no prefix on an operation's name, the method called is a universal one. Thus the operation Quicksort refers to a universal method. If the name is prefixed by /, then the operation is polymorphic, calling a method applicable to the class of the instance arriving on the leftmost input terminal. This class can be determined only at run-time. In Figure 3a, operation /Build value list is a call to method Build value list of class Index, provided that at run-time the leftmost input is indeed an instance of class Index. For the sake of completeness, we note that if an operation's name is prefixed by $\sim$, then the called method is in the same class as the method containing the calling operation.

A comment may be attached to any icon or program element and serves only to provide additional information for the viewer. Thus the comment "Index" attached to the input root for the first case of method Index/Sort in Figure 3a reinforces the interpretation that the input to Index/Sort should be an instance of class Index.

An initialisation method for a class overlays the standard instance generation for the class in that the initialisation method is automatically called whenever a new instance of the class is generated. In our example, whenever a new Book instance is created by the instance operation, the initialisation method Book::< is called and adds the new book to the persistent list Reviews. This is shown in Figure 4.

Similarly, there may be extraction and assignment class methods which overlay the standard get attribute and set attribute respectively. Inputs for such overshadowing methods are copied from the terminals of the associated operation, the method executes, and then the method’s outputs are copied to the roots of the operation. Figure 4 also illustrates that there is an extraction method for attribute Review and an assignment method for attribute Review. Note that the icon shapes for initialisation, extraction, and assignment class methods indicate the functions of these methods.

The value of any data is shown within a value window by the primitive display. In Figure 5, execution of the display operation opens the value window on the right for a view of the persistent Reviews, an instance of built-in class list. In this example, the user can toggle between textual and graphic representations of Reviews by clicking on the "Graphic" checkbox of the value window.

Lists, one of the fundamental data types in Prograph, can be processed using several different several multiplexes. Method Index/Build value list, shown in Figure 3b, illustrates how multiplexing provides a mechanism for parallel processing of lists. After being called with input of an instance of class Index, such as in method Index/Sort, case 1 of the method gets the value of attribute entries. The value of attribute entries, a list of instances of class Index Entry, is passed to the get attribute multiplex which has a list annotation, $\sim$, on its input terminal and an output root. This indicates that the multiplex expects a list as input, applies the operation to each element of the list, and assembles the results into a list on the list root. The list elements may be processed in parallel. In our application, the output should be a list of key values, which is then set as the value list of the input Index instance.

Another multiplex associated with list processing is the partition multiplex, which applies a Boolean operation to each element of an input list. The left output is the list of elements for which the Boolean operation produces TRUE and the right is the list of elements which produce FALSE. The partition multiplex is used in case 2 of method Quicksort, shown in Figure 7a.

We next describe a technique for terminating the execution of a multiplex. When an operation executes, it may succeed, fail or generate an error. Normally an operation succeeds after the associated method is called, but failure can be generated within the method. For example, a match operation will succeed or fail. When a terminate-on-failure control is associated with a match operation, failure of the match terminates the multiplex which calls the method containing the match operation. Figure 6a illustrates a repeat multiplex. When executed, case 1 of method File/Import inputs a system file number and repeatedly reads review data from the file and adds reviews to the database through calls to Review/Get Review. Primitive read-line outputs an error number of zero if and only if the file read is successful. In
operation, ends the execution of the multiplex. This is an example of a white
end-of-file do loop in Prograph.

As we mentioned previously, a method consists of a sequence of cases and a
call to a method initiates the execution of its first case. Controls can be used
to halt execution of the current case of a method and begin execution of the
method's next case. To illustrate this, we next describe a Prograph
implementation of the well-known recursive quicksort algorithm to sort a list
of index entries by key value. The method Quicksort is shown in Figure 7a
and contains two cases.

The first case of Quicksort handles the input of an empty list and the
second case handles a nonempty list. Case 1 matches its input with the
empty list, (). If the match succeeds, then case 1 outputs the empty list to
the root of the call. If the match fails, then the next-on-failure control,
indicated by the symbol on the operation, dictates that case 2 is to begin
execution. Case 2 of Quicksort calls the system primitive detach-l,
which inputs a list and outputs the left (first) element of the list and a list of
the remaining elements. The left element is used as the "pivot" element for
the quicksort algorithm. The next operation in case 2 is a partition multiplex
which partitions the rest of the list into two sublists. The partition is
determined by the Boolean output of the call to method I belonging to the
class of the left input of the operation. In our sample application, the left
input of I should be an instance of class Review, so I should be a method of class Review, as shown in Figure 7b. The two sublists are then
sorted by recursive calls to Quicksort. Finally, the sorted lists and pivot
element are joined together to give the complete sorted list which is then
output by method Quicksort.

Figure 7.

Generally, operations between which there are no input/output dependencies
are considered to execute in parallel. Thus, for example, there is parallelism in
the two recursive calls to Quicksort within case 2 of method Quicksort. When it is necessary to specify that one operation must execute
before another the user can sequence operations with the synchro mechanism
which is indicated pictorially by a succession of rounded arrowsheads from
the operation which must execute first to the second operation. Figure 6b
illustrates sequencing of the importing of review data from an input file,
followed by the closing of the file.

4. Environment

The Prograph environment has three main components, the editor,
interpreter, and application builder. The editor is used for program design and
construction, the interpreter executes a program and provides debugging
facilities, and the application builder simplifies the task of constructing a
graphical user interface for a program.

4.1 Editor

The editor is the key component of the Prograph programming environment
and serves the user in several tasks and contexts. The editor is used to define
any data attached to Prograph elements. Each major component of a program
is displayed and edited within an appropriate edit window. There are edit
windows for classes, attributes, persistent, methods, cases and instances.

The user interacts with the editor through mouse and keyboard actions which
are uniformly interpreted by the editor according to the context of the action.
As mentioned in section 2, the behavior of the editor is consistent with other
Macintosh applications. For example, the different cases of a method can be
accessed by clicking on boxes on the right side of the title bar of a window
for one of the method's cases, or from the Cases Pane, as shown in Figure 8.

Clicking on the "Show Case Pane" box toggles between showing and hiding
the Case Pane. Clicking on the "Prior Case" or "Next Case" box opens a
window for the prior or next case in the method's sequence of cases.
Manipulation of cases in the Case Pane is consistent with the general mouse
operations of the Prograph environment. For example, a new case between
the current cases 1 and 2 of Quicksort's case sequence is added by clicking in
the background of the Case Pane between the icons for cases 1 and 2.

The editor provides mechanisms to help prevent the creation of syntactic and
logical errors. For example, the user is not permitted to make incorrect
connections, such as a root to a root. One area that is open to
misconstruction is the relationship between the numbers of terminals and
roots of an operation and the numbers of input bar roots and output bar
calls of the cases of the operation. In order to assist the programmer with arities, the editor has features to ensure that newly created
or modified methods, cases and operations have arities which match their
corresponding Prograph components.

4.2 Interpreter

The interpreter contains many features which facilitate the edit-execute cycle
of program development and debugging. Debugging involves detecting and
correcting syntax and logical errors. When an error in an executing program
is detected by the interpreter, it immediately localizes the error, provides
access to the editor for modification of the program's components, including
active data values, and may intelligently suggest a correction of the error.

The programmer can use other debugging facilities of the interpreter to eliminate
logical program errors.

A program execution will be suspended when the interpreter detects an error.
For example, a case may call a primitive operation with input that is outside
the acceptable range for the primitive. If such an error occurs, the interpreter
suspends execution at the operation call, opens an execution window for the
case, and presents an error message.

An execution window for a case illustrates the current execution state of the
case and is based on its edit window. Operations which have already executed
are shown normally, the suspended operation call is flashing, operations
which have not yet executed are dimmed, and the background is dotted to
distinguish this window from the case's edit window. Note that there may be
several occurrences of a case on the execution stack, each with an associated
execution window. Figure 9a shows an execution window for case 1 of
method Book/ke, with execution suspended at the call to primitive attach-
r.

In order to assist the determination and correction of logical errors, the
programmer can control when the interpreter will suspend a running program.
For example, the programmer can set breakpoints on operations so that the
interpreter will suspend execution of the case just before calling an operation
with a breakpoint. When a case is suspended because an operation breakpoint
is reached by the interpreter, the system opens an execution window in which
the operation with the breakpoint is highlighted.

The Prograph interpreter provides a powerful tool for the top down
development of a program, automatic run-time creation of methods. During
top down refinement of a program, operations may be included for which
there are no corresponding method definition. When attempting to execute
such an operation, the interpreter generates a message indicating that the
programmer can control when the interpreter will suspend a running program.
For example, the programmer can set breakpoints on operations so that the
interpreter will suspend execution of the case just before calling an operation
with a breakpoint. When a case is suspended because an operation breakpoint
is reached by the interpreter, the system opens an execution window in which
the operation with the breakpoint is highlighted.

4.3 Application Builder

The application builder provides a tool for the construction of a program's user
interface. The interface may be defined in a graphical editor, and the
application builder will convert the interface definition into the corresponding
Prograph application.

5. Conclusion

The Prograph environment is a powerful tool for program development and
deployment. It provides a graphical programming environment which
facilitates the creation and debugging of programs. The environment is
integrated with a programming language that provides a rich set of
constructs for program design and construction. The Prograph environment
provides a comprehensive set of facilities for program development, making
it a valuable tool for programmers.
execute window for any case in the stack window can be opened by double
interpreter adds calls, the stack in this window grows from top to bottom and
by clicking on the case’s icon. Figure 9b illustrates the stack window during
sort of the index entries.

While a program is suspended, the programmer has access to
occurs, the interpreter performs
The programmer may also trace execution through dynamic views of program
Myers defines
the stack window is open during program execution, the window is updated
dynamically
application, and specify
other things, name the application, add windows and menus to the
typical of most other UIMS’s. For example, Smalltalk-80 [13], is a
textual language, edited and executed on a graphics workstation. The Interface
builder of the NEXT Computer [23], is a graphical system for generating
written in Objective-C. HyperCard is a tool for developing certain types of
applications with pictorial interfaces, programmed with the textual language
HyperTalk [14]. In each case, the programmer works with two types of
environment; one for the task of writing and editing programs in a textual
language, and the other for constructing and testing the user interface.
Prograph’s application builder is a seamless UIMS. Using graphical editors,
the programmer constructs menus, window, and window item objects which
form the graphical interface of an application written in a visual, object-
oriented programming language. In this section we discuss the editing of the
graphical interface components of an application and the behavior of a
running application.

Prograph’s system classes define the objects for building an interface based on
menus, windows, and events. A system class has system attributes which
include static properties of instances and properties for the run-time, event-
driven behavior of class instances. For example, static attributes of a
Window instance include the name of the window and the list of window
item instances for the window. Run-time attributes include the names of
methods to call when, at run-time, particular events occur. For example,
system attribute key method of a Window instance is the name of a
method to call if a keyboard event occurs while the window is the front
window of the running application.

The application builder has editors for the different system classes. Figure 10
illustrates the application editor, with which the programmer can, among
other things, name the application, add windows and menus to the
application, and specify a method to call when the user selects the “About…”
menu command while the running application is running.

Each application window can be edited via the window editor with which the
user specifies a window’s size, type, and run-time behavior. The user can add
items such as buttons, lists and text to a window and determine the static and
run-time behavior of these items. Figure 11 illustrates the edit window for a
window titled “Index” and the dialog that opens when the programmer double-
the “Go To” button. Here, the programmer has specified that a method
Go To is to be called when, at run-time, the user clicks the “Go To” button.

The menu editor for a menu is a dialog which depicts the menu as it will
appear when the user pulls it down. The programmer can add and delete menu
items, specify their appearance, and give the names of methods to call at
run-time when the user selects menu items.

The user starts execution of the application by an appropriate menu
command. The interpreter then switches context, displays the application’s
active menus and windows, and enters an event-handling loop for the
application. The interpreter receives and processes all events generated by the
operating system. While the application is running, the user can give menu
commands to suspend execution and then access program components
through the editors.

A graphical UIMS is an excellent prototyping tool. In the case of Prograph,
the application builder is more than a prototyper. The application builder
provides instant access to the user interface of a program during all phases of
application development, design, implementation, testing and maintenance,
and is therefore an integral part of software development in Prograph. This
style of interactive execution and editing is discussed in [17].

5. Conclusions and Future developments

In the above we have explored the use of pictoriality in developing an
integrated programming environment not influenced by traditional
programming language design. It is important, however, to identify
appropriate roles for text in the programming environment, since pictures
clearly have drawbacks in certain situations. A prime example of the
superiority of text is in expressing well-known and compact concepts which
are inherently textual, such as algebraic formulae. Text is also useful for
naming program elements which are pictorial, to distinguish individuals of
the same kind: for example in Prograph, textual names are used to identify
methods of the same class. Finally, comments obviously require text, since
the information they convey is informal and explanatory. However, when the
formal structure of the programming language is pictorial, comments do not
become confused with the logical elements as in textual programming
languages. In general, pictures provide a better representation of most
complex structures, such as algorithms, since they correspond more closely
with the mental model of these structures. Expressing such things in text
introduces an unnecessary layer of abstraction which must be removed by
parsing the text to mentally construct a picture.

The Prograph language can be extended to include some useful mechanisms
which are not well expressed in text. Multiple inheritance, in which classes
can inherit methods and attributes from more than one parent can clearly be
added by allowing the graph that specifies the class hierarchy to be a directed
acyclic graph rather than a tree. A well-known problem of multiple
inheritance arises when a class has several parents to which distinct methods
of the same name apply. Pictorial devices can be used in resolving such
conflicts. First, classes with conflicts can be visually distinguished from
those without. Second, a window can be opened on to a class with a conflict, displaying a grid with one row for each ambiguous method name, and one column for each parent class. The user can then resolve the conflict for each of the listed methods by selecting the appropriate square in the grid. In contrast, in textual languages, conflicts are resolved either by some defaults, which are not necessarily appropriate in every application, or by complex textual specifications.

In section 2 we discussed the use of pictures for representing data in whatever form is appropriate to the application. Prograph currently supports such representations for system classes, for example, representing an instance of class Window as a window. Future extensions [7] will allow any representation for instances of classes to be programmed by the user.

Another planned extension [7] is to allow methods to be attached directly to instances rather than just to classes. This will reduce the necessity for creating many special classes simply to capture slight differences in behaviour.

References