DEMOB - An Object Oriented Application Generator for Image Processing

N. Bryson, D.H.Cooper, J.G.Graham, D.P.Pycock, C.J.Taylor, P.W.Woods

Wolfson Image Analysis Unit, Department of Medical Biophysics
The Victoria University of Manchester, Manchester M13 9PT

This paper describes an Object Oriented program generator for image processing applications. Control is represented by a dataflow graph and interaction by "view" objects which update displays and modify domain objects. Progress so far also indicates that Object Oriented Programming for user-defined image processing requires a rich programming support environment.

Many groups have developed libraries of image processing (IP) modules for applications in the medical and industrial fields. The use of these libraries requires considerable programming effort and expert knowledge, which limits the economic viability of the technology. To develop an application, the user must not only have expert knowledge of image processing algorithms, but must also contend with the messy details of programming. Because of the commercial pressure to quickly demonstrate the feasibility of program designs, there is a need for an application generator. This paper describes our attempts to develop such a tool (DEMOB). The primary aim of this tool is not to help with the "art" of image processing, but to help make the construction of an application as direct as possible.

Objectives of DEMOB

One of the objectives in designing DEMOB was to investigate the problems involved in designing an application generator. Prototyping tools implemented using conventional structured programming suffer from having the structure of domain objects, interactions, and control programmed in. For example the implicit control structure may be a pipeline, along which a work image flows, with the user being offered (via menus) a choice of the operations to be performed on the image. Interactions occur in a pre-programmed manner, and tedious re-coding is required when new operations are added.

Our design goals included the requirement that control be separated out explicitly from the operation of the tool and from the domain objects. Interactions should also be factored out, so that interactive tools for different domains could be rapidly built from a set of building blocks. The program generator should be open, so that further domain objects and interactions could be added easily. The user should not be constrained to follow a particular design cycle, but instead should be able to interact with subcomponents of the problem in a relatively unstructured way. The Object Oriented Programming (OOP) technique[1], with its desirable properties of information hiding and run-time binding seemed to offer a viable approach to implementing such a tool. One of the aims of the project was to explore the uses of OOP for IP. Because of the nature of IP, with the need to display and interact with raw and processed images, the tool should have a mouse - driven, window based graphical interface. A brief description of the OOP paradigm is given below. For a fuller description, the reader is referred to reference 2.

APPLICATION REPRESENTATION

Separation of Control

A prototype application is represented internally by a dataflow graph[3]. This representation was chosen because the explicit recording of dependencies between data items provides the potential for automated reasoning about applications. A simple graph, representing "blob" extraction from an image, is shown in figure 1. The dataflow graph consists of: nodes, representing actions applied to data objects; tokens, representing references to data objects; and directed arcs, which transfer references to tokens between nodes.

A node is ready to fire when it has received all its tokens from its input arcs. New tokens are created by the node, and are transferred to further nodes via the

This work is supported by an SERC grant and is part of ALVEY project MMI-093: "Techniques for User Programmable Image Processing (TUPIP)"

AVC 1988 doi:10.5244/C.2.7

output arcs, making them ready to fire. After execution only the "dangling" output arcs of the graph contain references to tokens, all intermediate data objects (and their tokens) having been consumed. Each token contains a counter for references to the data object. When references to tokens are created or consumed, the counter is incremented or decremented, respectively. When the counter reaches zero, the data object, and token, can be deleted.

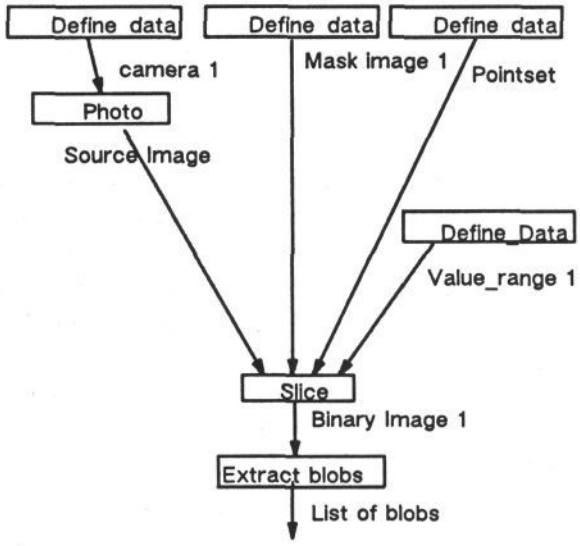

Figure 1. Graph representing blob extraction.

The User Model

DEMOB is designed to present the application and data objects graphically to the user. The user may select any action or data object in his application by selecting the appropriate node or arc of the graph, which will respond by offering an interacton with the appropriate object. Each interaction takes place within a window, which contains a menu, a display of the object, and a prompt line. Selection within the window leads to a more specialised interaction with the object, or an interaction with a new object. Initially the user is offered a general interaction with an empty list object. Typically the user then chooses to load a graph from a disc file, or to create a new example. An interaction with the new graph is entered and the user is presented with a diagram such as that shown in figure 1, and a menu of available interactions. For example, if the user selects a dangling output arc he will enter an interaction with the data object on the arc. The arc itself can be selected, and modified so that, during execution, it will display the data objects passing through it.

During execution the arcs and nodes are highlighted as they become active, and windows open on the screen to show data objects as they are created. The user modifies his program by cutting arcs, creating new nodes, and adding them to the graph. Graphs and objects can also be saved and retrieved from disc file.

IMPLEMENTATION

The Object Oriented Programming Environment

The key features of OOP are data encapsulation and inheritance. The OOP technique consists of identifying objects which are to be manipulated in an application, and in defining the data structures and operations needed for each object. The private data structure representing the object is protected from direct manipulation by the user, and operations on it may only be carried out by sending a message to the object, which uses it to select an appropriate operation. It is important to note that the message only conveys "what" the programmer wants done, but the object itself decides "how" it is done. For example the message AREA sent to a shape object may cause different operations to be carried out, depending on whether the shape is a circle, rectangle, etc. Because the internal representation is hidden, it can be changed without affecting the user, who need only know the messages to which the object responds.

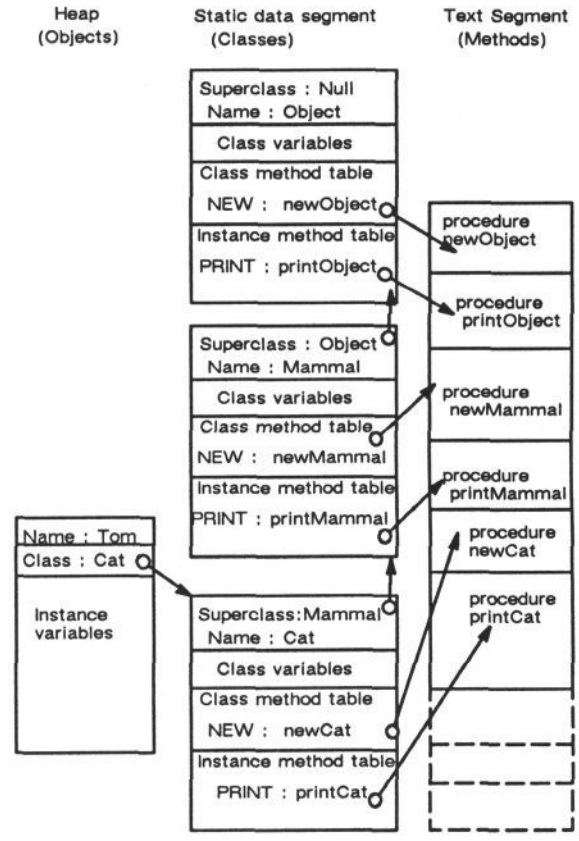

Figure 2. Schematic memory map of OOP system

An individual object is regarded as an instance of a particular class e.g. Tom is an instance of the class

Cat. The class definition defines the data structure ("instance variables") of its instances and the messages (with corresponding operations or "methods") to which it will respond, in the form of a lookup table, indexed by the message selectors. Each object contains a pointer to its class, which is itself an object, with corresponding class methods and "class variables". In particular, each class can respond to the message NEW by returning a new instance (by invoking the newCat method). Figure 2 shows a schematic memory map showing the relationship between objects, classes and methods.

The use of encapsulation helps to protect code from the effects of changes. The use of inheritance allows code to be re-used. The classes are arranged in an inheritance hierarchy, with more general operations and instance variables being defined in classes high in the hierarchy. For simplicity, each class can inherit from one "superclass" only, although in other implementations, multiple inheritance is a useful feature. Each class contains a pointer to its superclass (except for Object, the class at the root of the hierarchy). Methods high in the hierarchy may only manipulate instance variables defined at that level or higher in the hierarchy.

We have developed an OOP environment in-house in which to implement DEMOB. This consists of C with some preprocessor tools. The programmer causes messages to be sent to objects by inserting code of the form:-

```
Tom = NEW$(Cat);
PRINT$(Tom);
```

where Tom is of type "ob_ptr" i.e. a pointer to an object. A special global object pointer, "self", represents the receiver of the current message, and is used within methods to access the instance variables of the object. A preprocessor tool checks that the message selector is represented in a file of valid messages, and then converts this string into a call to a message handling routine:

```
Tom = msg(NEW,Cat);
msg(PRINT,Tom);
```

The message handling routine expects the message selector and the receiver of the message as the first two parameters. Upon execution, the message handling routine searches the method lookup table of the class of the object for the corresponding method (it recognises a class by the fact that its class pointer instance variable is null). The stack is adjusted to simulate a standard C procedure call, and control is passed to the method.

The programmer may also specify that the search for a method should begin in the superclass, rather than in the class, by writing:-

```
PRINT$super(Tom);
```

This powerful facility allows the chaining together of methods. For example, the PRINT message causes the printing of the values of all instance variables of the object, and is implemented at each level of the class hierarchy by methods of the form:-

```
PRINT$super(self);
printf("Instance variables at this level");
```

This causes the instance variables, starting at the root of the hierarchy, to be printed.

The low level image processing routines of our system are microcoded to make use of particular data structures, optimised for IP operations and executed on a slave co-processor. Since a large effort had already been expended in developing existing software and hardware,we were constrained to include these existing data structures in DEMOB. One of the objectives in developing DEMOB was to investigate the effective use of mixed typed data structures and objects.

Image Processing Objects

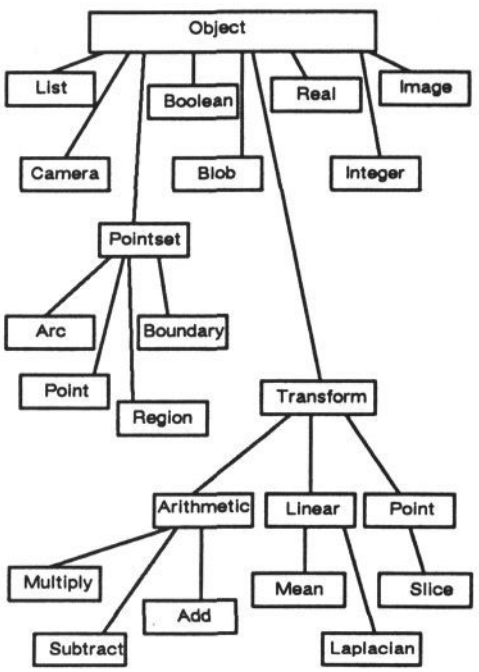

Figure 3. Image Processing Objects Class Hierarchy

Since we wish to be able to add new classes and modify existing classes without too much disturbance to the rest of the system, DEMOB makes no assumptions about the domain classes, other than that they will respond to a limited set of messages, which provide the interface between these classes and DEMOB. These messages allow DEMOB to ascertain the default method of interaction and the operations available for the domain object. Besides general purpose objects such as integers, reals, booleans, lists, etc., a variety of data objects specific to IP are also required. These include images, cameras, pointsets, value ranges, and image transform specifiers[4]. A class hierarchy of these

objects is shown in figure 3.
Associated with each object is a set of messages which are used to process the data e.g. a Camera object responds to the PHOTO message by producing an Image object. The processing steps in the application are built up using these messages, which are inserted into the graph as described below. Also associated with each object is a set of interactions e.g. the Real number object has a set of interactions allowing the number to be modified in a variety of ways. These interactions can be inserted into the graph as described below.

Graph Nodes

The structure of the dataflow graph allows control over the order in which operations are carried out. Further control is provided by supplying specialised nodes which implement further control structures. The class hierarchy of the objects used in constructing graphs is shown in figure 4. The Node class provides the functionality common to all nodes e.g. the ability to receive and transmit tokens. Further, more specialised functionality is provided by the subclasses of Node.

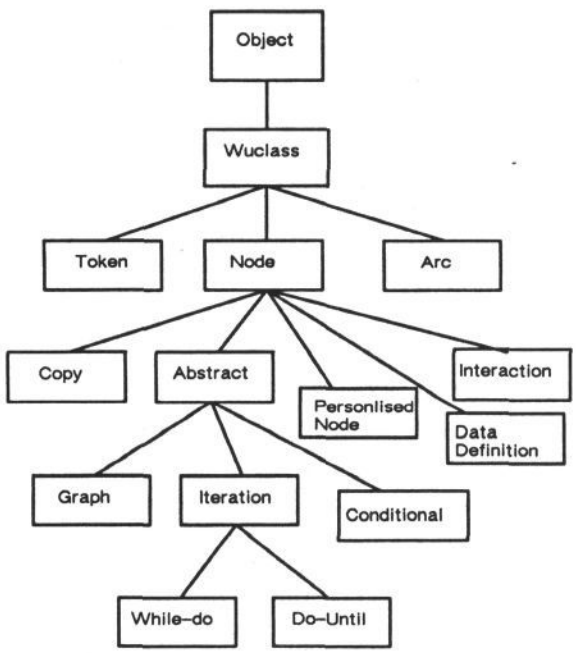

Figure 4 . Inheritance hierarchy of graph objects

One such class is the "Data Definition" class, whose instances act as sources of data for the dataflow graph. When the node is created the user is asked to define the data object, and on subsequent execution a token representing the data object is output by the source node. In the example graph shown in figure 1, four data objects are needed : a camera, a mask image, a pointset (defining the region over which the operation is done) and a value range (which defines the thresholds for the slice operation).

Multiple references to data objects are frequently required, so a "copy" node class is provided. These work by incrementing the reference counter of the input token, and passing out multiple references to it via the output arcs.

A "graph" node, like any other, can receive tokens from its input arcs, and passes tokens to its output arcs. Internally its execution is represented by a collection of nodes and arcs i.e. a sub-graph. A graph can thus have a layered structure, with a "root" graph node controlling the execution of its sub-graph, which itself can contain graph nodes.

Iterative execution is implemented by providing a variety of "iteration" node classes. For example, a specialised type of graph node, the "while-do" node controls the execution of a condition graph and an action graph. During execution the set of input tokens is passed to the condition graph, which executes and returns a boolean object. If it is "true" the set of input tokens is passed to the action graph, which returns a new set of tokens, and the cycle is repeated. If the boolean is "false", the tokens are passed out as output.

Conditional execution is implemented by providing an "if-then-else" node class. This node controls a condition graph, a "true" graph and a "false" graph. The execution cycle is similar to that of the "while-do" graph i.e. the input tokens are passed to the condition graph, and a boolean object is returned. Depending on its value, the input tokens are passed to the "true" or "false" graph.

These nodes introduced so far provide the control structures of conventional structured programming (a subroutining facility could easily be added). A further class of node, the "personalised" node, is included to allow user selected actions on data objects. Actions are carried out by sending a message to an object e.g. to add two numbers together the message is :-

```
C = ADD$(A,B);
```

where A, B,C are references to the number objects, ADD is the message, and A is the receiver of the message. The personalised node has instance variables allowing it to store this message. The convention adopted is that the message is sent to the data object input via the first input arc, with the data objects from the other input arcs as parameters. The returned object from the message is assumed to be a list of output data objects, which are transferred to the output arcs. In this case a single object, C, is returned. During the creation of a personalised node, the user is prompted to define a receiver, which then offers for selection all the available messages for that class of object.

Part of the user's application may involve run-time interaction with the data objects, and a special "interaction node" class is provided to allow this. When creating this node, the user will be prompted to define the class of input, and will be asked to select from all available interactions with this class. An instance of this interaction (see below) will be created and placed in the interaction node. When the data object arrives along the input arc, the interaction will take place, and the modified data object will be output along the output arc.

The Interaction Model

The programming of interaction required a large part of the effort in developing DEMOB. A graphics sub-system was developed, which implements a GKS[5] model of graphics i.e. overlapping rectangular regions of the display device, called "viewports", display "icons" in "worlds" through "windows" opening on the worlds. Viewports are grouped together in frames. These components were implemented by defining classes of objects, with all device-dependent code being confined to the Screen class.

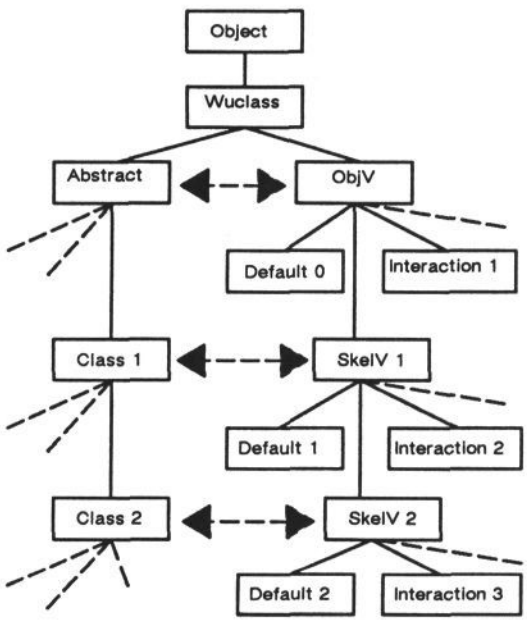

Figure 5. The View Class Hierarchy

Rather than have each domain object controlling its interactions, we decided that each interaction should be defined by a separate class, to increase modularity and to save memory space. When an interaction with an object is required, an instance of the relevant class, known as a "view", is created and is put in control of the object. The view object would inherit the instance variables and most of the methods needed to represent and manage the display from its superclasses, and need only provide locally the code needed to determine the course of this particular interaction.

We thus define a class hierarchy of views to accompany the hierarchy of domain objects, as shown in figure 5 . This view hierarchy contains "skeleton" classes which provide all the instance variables and methods common to interactions, and "view" classes which define particular interactions. Each object has a default view class, which implements the most general interaction possible i.e. to display the object and to offer all interactions available for it. It also allows the user to select sub-components of the object for further interaction. When a new domain class is added a corresponding skeleton class with default view and any further view classes are also added.

RESULTS

A basic version of DEMOB has been implemented on a CVAS 3000 (Visual Machines Ltd.) system. Images of the screen, showing a sample graph during and after execution, are shown in figures 6 and 7. These show a graph interaction window, with associated menu, and displays of a source image and a sliced image.

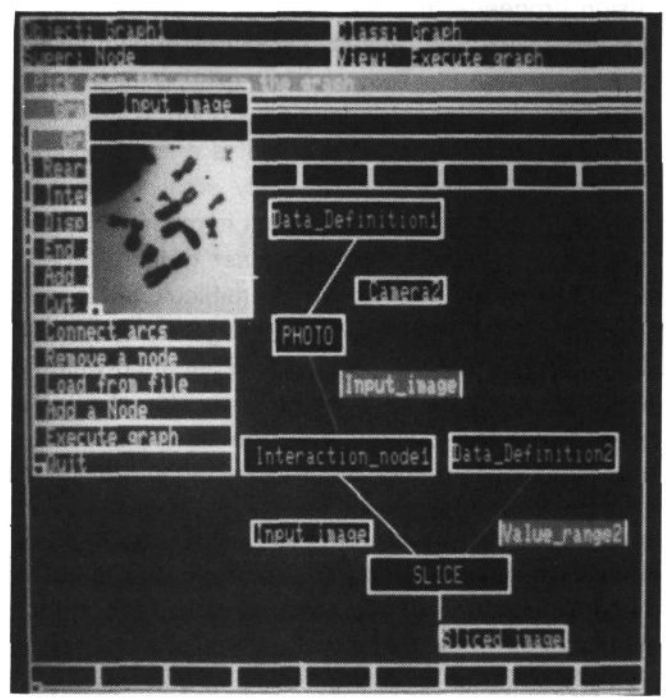

Figure 6. Sample graph during execution

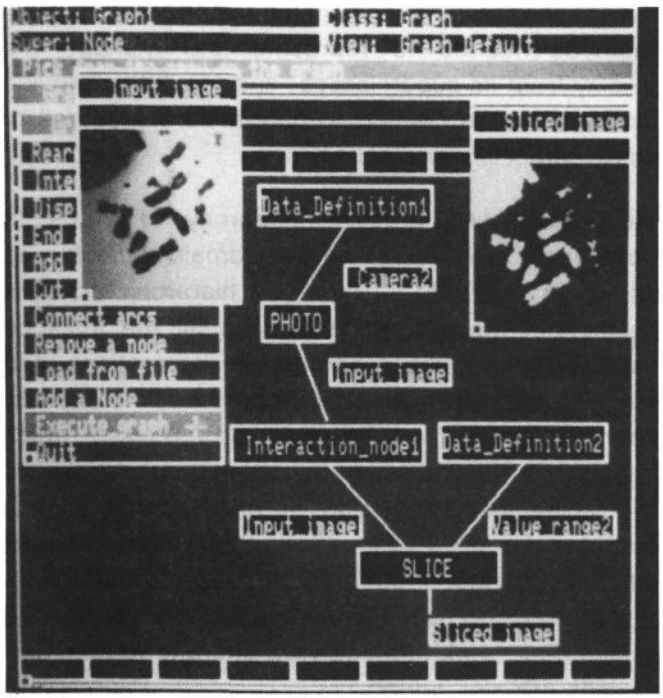

Figure 7 – Sample graph after execution

DISCUSSION

Limitations of the OOP Environment

Designing with objects proceeds by identifying the instance variables and methods of classes. As the design proceeds, the programmer identifies ways in which code can be reused, usually by spotting ways of adding new classes and by splitting or moving methods and instance variables around in the class hierarchy. The final system contains a large number of classes, each with a relatively small number of methods, each containing a small number of lines of code. It is easy for the programmer to lose track of such a system, and it can be difficult for a new programmer to add new functionality to the system, since in order to do so, he must understand what he is inheriting. We thus need tools to allow the programmer to browse around the source code in a structured manner such as are supplied in Smalltalk[6].

In our OOP environment, all class and method definitions are compiled. While this increases run-time efficiency, it also means that the definition of an object is fixed. Further, although the code is compiled, no static type-checking is done.

Design Issues Raised in OOP

Initially, design concentrated on using the "information hiding" aspect of OOP, but as we gained experience, inheritance gained in importance. The derivation of a class hierarchy can be difficult, in part because the only relationships provided are the "is-a" and "is-an-instance-of" relations. These do not, however, describe the relationship between say, views and their objects. For these objects the "information hiding" aspect of OOP is a disadvantage, since the views need to have access to the instance variables of their objects. This problem arises because it is difficult to decompose IP applications into independent static classes of objects i.e. where information can be encapsulated permanently into objects. In practice we wish to be able to merge the information from several classes of object e.g. in applying an operation over a pointset in an image. Where a static encapsulation of information suffices(e.g. the dataflow graph) the model works well. The frame paradigm [7] presents a richer environment for the representation of domain knowledge,allowing object decomposition as well as more complex relationships between objects.

Our experience has shown that the use of mixed typed and object variables reduces the advantages of using OOP, since much code had to be handcrafted to deal with the typed variables. Of course, these are needed to map onto the IP software and hardware. The correct way to deal with them is to implement a basic set as objects, with methods handcoded to implement the functionality provided automatically for other objects. It is instructive to compare our approach with the Eiffel OOP language [8] . This language has a compiler which recognises the use of a limited set of simple types, but any complex data structure must be represented as an object.

Design Issues in the User Interface

The user interface is crucial to the success of a tool such as DEMOB. Users are sensitive to features which appear relatively trivial, such as the particular style of interactions, or the wording of prompts. There are a variety of ways in which an interaction, such as modifying an integer, can be implemented. One solution is to recognise that these interactions have much in common, and thus could be represented by an appropriate view class hierarchy.

Instead of being presented with the dataflow graph, some users felt that it would be better for this to be hidden, and instead to present only the data objects. Design would proceed by creating and selecting objects for processing. The user would then selects an appropriate operation which could take these objects as inputs. The selected action would be invoked, causing new objects to be created.

CONCLUSIONS

The dataflow graph is adequate for representing an application. However the need to fully specify the dataflow is tedious for programmers, since the

normal constructs of conventional programming languages correspond to a lot of dataflow, which the programmer does not normally need to specify explicitly.

While OOP greatly increases the robustness and reuseability of code, and provides a useful paradigm for the development of complex software systems, the behaviour and relationships between the complex data structures used in IP are not adequately modelled by message passing objects arranged in a single inheritance class hierarchy. To use OOP the development environment should be fully integrated with the language to allow rapid modification of an evolving system. A clean interface between typeless data structures (objects) and typed data structures should be maintained. The dataflow graph provides a viable means of representing procedural knowledge, but needs careful design of the user interface in order to gain user acceptability.

REFERENCES

1. *ACM SIGPLAN Notices*, Vol. 21, No. 10, Oct. 1986
2. **Brad C. Cox,** "*Object Oriented Programming – An Evolutionary Approach*", Addison – Wesley, 1986
3. *IEEE Computer*, Special Issue on Dataflow Systems, Vol. 15, No. 2, Feb. 1982
4. **Graham J., Taylor C.J., Dixon R.N.,** "A compact set of image processing primitives and their role in a successful application program", *Patt. Recog. Lett.*, 4, 325 – 333, 1987
5. **Hopgood F.R.A., Duce D.A., Gallop J.R., Sutcliffe D.C.,** "*Introduction to the Graphics Kernel System (GKS)*" A.P.I.C. Studies in Data Processing No. 19, Academic Press, 1983
6. **Goldberg A., Robson D.,** "*Smalltalk – 80 The Language and its Implementation*", Addison – Wesley Series in Computer Science, 1983
7. **Wood P.W., Pycock D.P., Taylor C.J,** "A Frame–based System for Modelling and Executing Visual Tasks", *Alvey Conference 1988.*
8. **Meyer B.,** "Eiffel: Programming for Reusability and Extendability", *SIGPLAN Notices* Vol. 22., No. 2, Feb. 1987