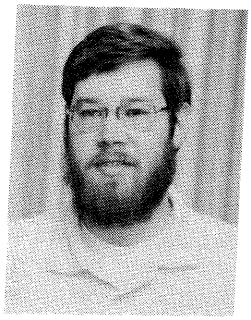
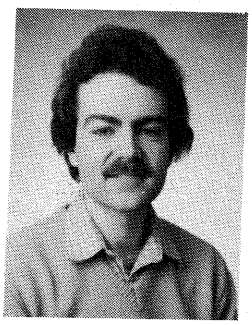
CIPS Congress 86, May 1986, pp. 311 - 317.

SOME EFFECTS OF GRAPHICAL USER INTERFACES ON PROGRAMMING ENVIRONMENTS

Duane Szafron
Brian Wilkerson
University of Alberta



Duane Szafron
Assistant Professor
Programming Languages Group
Department of Computing Science
University of Alberta
Edmonton, Alberta
T6G 2H1



Brian Wilkerson
Graduate Student
Programming Languages Group
Department of Computing Science
Universtiy of Alberta
Edmonton, Alberta
T6G 2H1

We investigate the effects of graphical user interfaces on programming environments. Graphical interfaces have an impact on design, syntax, incremental compilation and execution. Programming environments can now directly manipulate symbolic program modules. Traditional textual syntaxes of programming languages can be replaced by more general graphical syntaxes. Graphical user interfaces can simplify incremental compilation by holding program fragments in various states of correctness. The runtime environment can use a graphical interface to select arbitrary code fragments for execution and to view and modify the program state.

Nous discutons l'effet des interfaces graphiques sur les environnements de programmation. Ces interfaces influent sur la programmation dans les phases de conception, syntaxe, compilation incrémentielle, et correction. La conception peut être tenue en compte par la manipulation de symboles représentant les modules du programme. La syntaxe traditionelle devrait être remplacé par une syntaxe graphique. La compilation incrémentielle peut être appuyée par multiples fenêtres contenant chacune une partie du code en développement. L'environnement de correction peut utiliser l'interface graphique pour sélectionner morceaux arbitraires pour exécution et modifier l'état du programme.

Introduction

This paper investigates the effects of graphical user interfaces on programming environment design. The first programming environments, such as the Cornell Program Synthesizer [1] and Emily [2] were designed for non-graphical environments. Since then, graphical environments like PECAN [3], GUIDE [4] - [5] and MUPE-2 [6] have been developed. The graphical user interface can affect many aspects of the programming environment. These aspects include: the display of design information, the syntax of the programming language, the incremental compilation process, and the run-time environment.

One possible approach to the graphical representation of programs is to display and edit different aspects of the program in different views. This approach has been taken by both PECAN and GUIDE. Aspects may include: code, declarations, documentation, static program structure, dynamic calling sequences and runtime behavior. A view may consist of text or graphics displayed in a window or a portion of a window.

Detailed design information may take the form of Yourdon style structured design charts, [7] calling sequence trees and other graphical representations. Because of the graphical nature of these representations, they must be viewed and edited with special purpose editors. This paper will discuss the paradigms used by such editors.

Standard programming languages have been defined using a linear syntax. The introduction of graphics allows for a two dimensional definition of the syntax. Automatic formatting of the textual representation of the program clarifies the internal structure of the program and the legal transformations. It also enforces consistency between users. Multiple fonts can be used to emphasize the structure and provide visual clues to correctness and completeness. This paper will examine and evaluate some of the options available for specifying graphical syntaxes.

There has been considerable discussion in the literature concerning the method of handling incomplete and incorrect programs. This is manifested in the different approaches to incremental compilation. This paper investigates graphical interfaces that allow us to design and evaluate new solutions to these questions.

The run-time environment may be affected the most by the use of a graphical interface. The use of features such as error windows, graphical displays of data structures and the run-time stack, execution of arbitrary code segments, and the interactive assignment of values to unassigned variables are among the possibilities. This paper looks at some of the features commonly included in programming environments, and suggests additional features.

Design

The first programming environments were concerned only with the coding phase of the software development process. The only structures that the user could edit were parse tree nodes and declarations. Graphical interfaces provide a convenient method for representing and editing larger structures like procedures and modules. Many new programming environments, like MUPE-2 and GUIDE allow the user to manipulate these larger structures. Tree structures can be used to display and edit the static structure of a program, its dynamic calling sequences or both. For example, figures 1 and 2 respectively show the static and dynamic structure of a sample program, where each node is a procedure. Note that the environment must have a way of representing simple recursion in which a procedure calls itself and complex recursion in which two procedures call each other. In figure 2, recursive calls of both types are represented by dashed lines.

A programming environment must do more than display these structures if it is to support the design phase of the software development process. It must support the integrated editing of these structures. For example, it must be possible for the user to select procedure "C" from figure 1, cut it and paste it below procedure "D". This operation must be automatically reflected in the internal representation of the program. On the other hand, consider an attempt to cut procedure "A" and paste it below procedure "D" in figure 1. This action should be rejected since figure 2 indicates that program "P" calls procedure "A". The new arrangement in figure 1 would violate standard scoping rules.

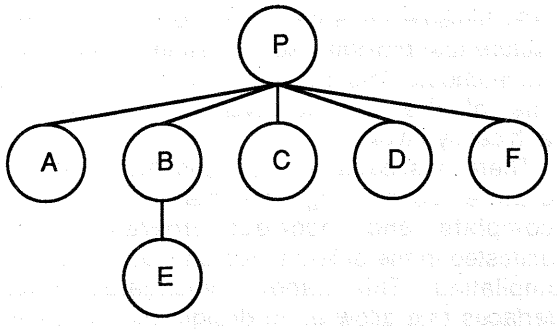


Figure 1 A static structure tree

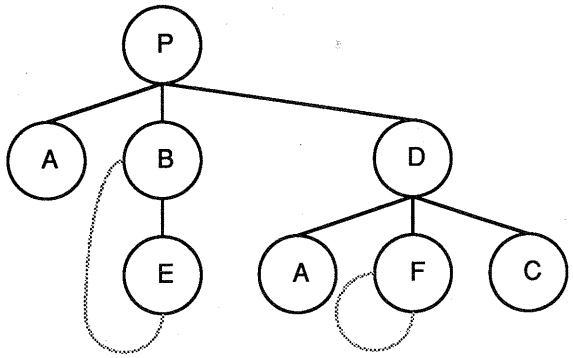


Figure 2 A dynamic structure tree

Notice that changes in the static structure of a program will be reflected in the declaration views of the procedures and that changes in the dynamic structure of a program will be reflected in the code views of the procedures. Ideally, such changes are really design changes and should be made in the structure views and reflected in the code and declaration views. In practice, the user often wants to make these changes in the code and declaration views. It is not clear which approach is better.

The key to integrating design into programming environments is to carefully map the causal relationships between the views. The next step in the evolution process is to introduce specification into the environment.

Graphical Syntax

The introduction of graphical user interfaces into programming environments has had little effect on the syntax of the programming languages which they support. The major reason for this is probably that environment designers usually work with existing languages and feel obliged to make the environment's view of the language look familiar to the users. However, even languages like Smalltalk, which were developed concurrently with a programming environment, have not taken significant advantage of the graphics in the language's syntax specification. We feel that graphical user interfaces can have a significant impact on programming language syntax both in new languages and existing languages.

In the case of a new language designed for integration within a programming environment, there is no à priori limit on innovative syntax. If the environment is designed to produce only target object code, then no complications will occur. However, if the environment must produce source code for external compilation, then it is necessary to have a textual form of the language source code. This implies that there must be a mapping between the "graphical" syntax of the programming language code and its textual representation. There may be

many graphical syntaxes that map to the same textual representation. Such mappings set limits on the generality of graphical syntaxes.

This situation is analogous to the ALGOL60 situation in which many representations with different lexics mapped to the same reference language syntax [8]. A similarity also exists between the development of the precise syntax for ALGOL and the development of graphical syntaxes. The precise definition of ALGOL syntax was due to the development of BNF. Graphical syntaxes probably will not receive widespread acceptance until we have a standard form for specifying them.

In the case of existing languages, environment designers are only limited by the restriction that the graphical syntax must map to the textual syntax of the language definition. We shall explore some of the freedom for representation of language syntax provided by graphical user interfaces.

Structures and Keywords

Programming environments which have graphical interfaces can replace some text by graphical objects. It is not clear how much of this text the environment should replace. There are control and data structures which exist solely to structure other information. These structures are candidates for replacement by graphical objects. However, it may be advantageous to retain some text for clarity even in those cases where it could be eliminated. For example, it might be clearer if an IF statement, which could be represented as a purely graphical object, was labeled with text. In general, programming environments can represent structures as purely textual entities, purely graphical entities, or some combination of these. Figure 3 displays three sample possibilities. The graphical representations are based on Nassi-Schneiderman charts [9] and the language GAL [10].

```
Modula-2:
    Type HashMap =
        ARRAY HashMapRange OF NodePtr

GUIDE:
    Type HashMap IS ARRAY OF NodePtr INDEXED
        BY HashmapRange
```

Figure 4 Choose keywords for clarity not brevity

Since software engineers think in English, the text is a useful aid to comprehension. This bias might be the result of training and might not exist for a programmer who is trained to use a purely graphical syntax. Perhaps programming environments should use keywords only in the transition period until purely graphical objects are familiar to programmers. This is an interesting area for study in human factors.

A programming environment may use an extended set of keywords which does not match the textual syntax of a programming language. Language designers often abbreviate keywords, like "ELSIF" in Modula-2, to minimize typing. Since keywords are automatically generated by a programming environment, there is no reason to abbreviate them. The emphasis should be on clarity, not brevity. For example, compare the clarity of the two array declarations in figure 4. The first is from Modula-2 while the second is from the GUIDE programming environment.

The number of keywords in a language has traditionally been kept small to minimize the number of reserved words. Keywords need not be reserved words in programming environments since they can be recognized from context as they are generated. However, using a keyword as an identifier will result in an error if the source code is compiled externally. The solution to this problem is a modification of the identifier during the translation from graphical to textual syntax.

```
WHILE not done DO
    read(aChar)
    IF valid(aChar) THEN
        done := TRUE
    ELSE
        showPrompt
    END
END
```

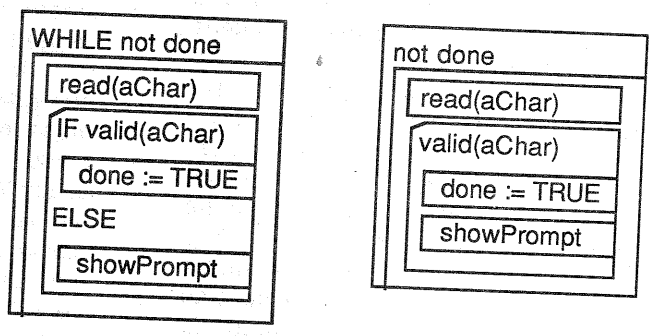


Figure 3 Structure representation using graphics and text

The use of keywords for identifiers can still lead to user confusion. The graphical interface can remove this confusion in a simple way. Keywords can be highlighted by the use of a different font, emphasis or color. This has the additional advantage that it emphasizes the structure of the program. The same technique can be used on those keywords which represent template place-holders. For example, in GUIDE, a hollow font is used for these place-holders as illustrated in figure 5.

```
FOR controlExpression DO
    IF condition THEN
        read(aChar)
    END IF
END FOR
```

Figure 5 Alternate fonts and styles

Environment designers should be aware that there is the possibility of "too much of a good thing". Care should be taken to avoid the use of fonts which are difficult to differentiate; strange fonts, which can be difficult to read; and the use of too many different font types.

Format

Although BNF can be used to specify the syntax of a programming language in a textual way, it is not used to specify the textual format of the language. Most modern programming languages are free format and tokens are separated by white space (one or more blanks, tab characters or end of line characters). This was a response to the situations encountered in FORTRAN and COBOL where indentation requirements and continuation marks were a hindrance to the programmer.

The current practice results in a variety of indentation formats which lack consistency between users and often are not self-consistent within a single program. Line-breaks also have no meaning and artificial structure separating or line ending tokens (like ";" in Pascal, Modula and Ada and "." in Smalltalk) are used. Language designers have abandoned two very valuable representation characteristics as a tools in syntactic specification. Indentation and line-breaks should be re-examined as vehicles for conveying syntactic information in the context of programming environments.

A fixed indentation scheme can be made automatic so that it is not a hindrance. A specific syntactic meaning can then be ascribed to indentation, so that fixed indentation can convey internal structural information. For example, the classic dangling ELSE problem [11] disappears.

In addition, fixed indentation also clarifies user options. For example, in GUIDE, information is changed by selecting structures. When the mouse button is pressed while the cursor is on a character, the smallest structure containing that character is selected. A fixed indentation scheme can make it obvious to the user which structure will be selected.

A line break can also be used to convey syntactic information. Since graphical user interfaces allow scrolling both vertically and horizontally there is no need to proceed to a new line in the middle of a programming structure. A line-break can be used as a natural end of structure (or structure separating) marker. This removes the need for structure ending or separating tokens in the case of a simple sequence of structures.

There are other situations where formatting can convey syntactic information in a very natural manner. The representation of expressions in GUIDE is a good example. Although the user may enter an expression without parentheses, according to the standard precedence rules between operators, the display of expressions is fully parenthesized to convey the internal structure and syntactic meaning of the expression. Figure 6 illustrates this feature.

```
answer := a * (b + c * d) + e * f          {as entered}

answer := (a * (b + (c * d))) + (e * f)    {as displayed}
```

Figure 6 Fully parenthesized expressions for clarity

Incremental Compilation

Incremental analysis or compilation is used in several programming environments: Cornell Program Synthesizer, POE, GUIDE, etc. For this discussion, it doesn't matter whether the environment is intended to produce machine code or some other intermediate representation. The major question that must be answered in such environments is what to do in the case of syntactic or semantic errors. Should the incorrect information be accepted and the error flagged or should the environment refuse to accept it? If the environment refuses to accept the information, then the user must take corrective measures and re-enter the information. This is a burden which the user should not have to bear. For example, if an assignment statement is entered and the variable on the left hand side is undeclared, a semantic error must be reported, but the user should not have to re-type the assignment statement after declaring the variable. It is for this reason that the Cornell Program Synthesizer and POE allow some errors in the program.

On the other hand, if the information is accepted into the program, then the environment must cope with an accepted program which is incorrect. Such a program is more difficult to represent and execute. Graphical user interfaces provide a good solution to this dilemma. The user is required to enter information in special dialog windows. As correct information is accepted, it is transferred to other windows which contain a representation of a correct program. Incorrect information simply waits in an entry window until other changes result in it becoming correct. For example, in GUIDE, a user enters an assignment statement in an assignment dialog box as displayed in figure 7. When the statement has been typed, the user "pushes" the accept button.

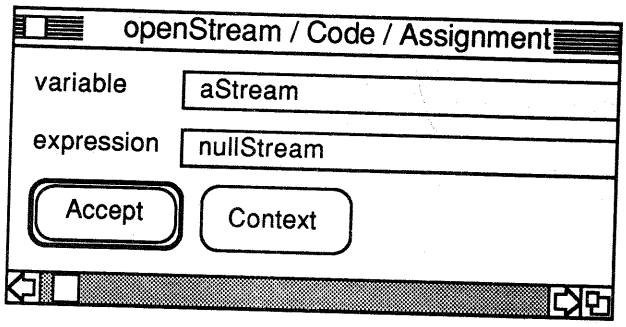


Figure 7 An assignment statement dialog in GUIDE

If a syntax error occurs, then the incorrect portion of the entry is highlighted so the error can be fixed. If a semantic error occurs then the user is informed as illustrated in figure 8. After the user presses the "OK" button to signal acknowledgement of the error, the assignment statement remains in the dialog box. The user can enter a declaration for the variable and then return to the dialog box and "push" the accept button. When the assignment statement is correct, it is accepted into the code and displayed in a code view in another window.

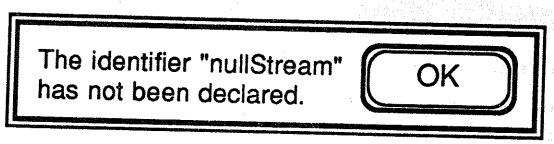


Figure 8 Notification of a semantic error in GUIDE

There is a more difficult hurdle to face in only accepting syntactically and semantically correct programs. There are several situations in which the user wants to allow a correct program to become incorrect for a short period of time. For example, consider the following situation. A user has declared a variables to be an INTEGER and has several references to this variable in the program. The user decides to change the variable to a REAL. If the declaration is changed so that the variable is a REAL, some of the structures which reference the variable will become semantically incorrect. If the program must remain correct, it appears that the structures containing these references will have to be deleted before the declaration can be changed.

A graphical user interface can supply a solution to this problem. When a declaration is changed, all structures containing this declaration are excised from the program. They are not thrown away however. Instead, they appear in dialog boxes just as they would if they were being entered. The structures can then be edited or other actions can be taken so that they become correct again and can then be accepted back into the code.

Since a large number of excisions may take place, the user may not know where the contents of a particular dialog box will be inserted if it is accepted. Some context mechanism is necessary to inform the user of the associated location. For example in GUIDE, each dialog box contains a context button as shown in figure 7. When the user presses the context button, a window for the appropriate view is displayed and an insertion bar shows where the structure will be inserted if the accept button is pressed.

Runtime

Graphical interfaces can have a significant impact on the runtime features that a programming environment provides. The major use of a run-time capability in a programming environment is debugging. Traditional debugging features include: setting breakpoints, dumps and single step execution. These features can be improved and extended with a graphical interface.

Debugging consists of three major activities: selecting code to be executed, viewing the program state and modifying the program state. Many of the features which can be introduced with a graphical interface can also be done without graphics, by using a text based debugger, but the graphics make them much more appealing. There are many text based language debuggers which are never used because of the difficulties in: learning how to use them, selecting execution start and end points, modifying the program state and displaying useful information.

Selection

In most compiled programming languages, the smallest unit which can be executed is a program. This is true in spite of the fact that many languages

allow separate compilation of smaller components such as functions, procedures or modules. On the other hand, in interpreted languages, the smallest unit of execution is much smaller. It is an expression in Lisp and Smalltalk, and a clause in Prolog. However, even in the case of interpreted languages it is difficult for the user to select a set of expressions and have them executed if a non-graphical interface is used. The problem is simply one of selection. With a mouse and bit-mapped display a programming environment can allow arbitrary structures to be selected for execution. For example, a block of three statements inside of an IF structure could be selected and executed or the condition of a WHILE statment could be executed. In addition, this random access to structures makes the setting of breakpoints and temporary suspension of execution a trivial exercise.

For example, figure 9 shows a typical graphical debugging window in which the text to be executed has been highlighted. The user has inserted a pause sign between the IF and WHILE statements. When a pause sign is encountered during execution, the execution is suspended so that the user may examine the program state. The user may then resume execution at any time.

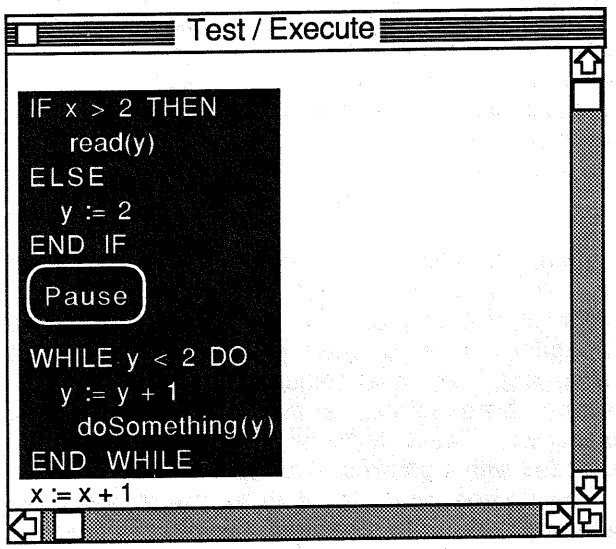


Figure 9 A graphical debugging window

Program State Display

Selective execution is useless without a means of detecting the results of the execution. A graphical interface can provide multiple windows, each displaying a different aspect of the executing code as in PECAN. In general, an environment could provide separate windows for: selecting the code to be executed, displaying the values of symbols, displaying the run-time stack, displaying the dynamic structure tree, and reporting run-time errors.

The values of symbols may be displayed in many ways. One approach is to use the declaration view, to select variables, parameters and constants so that their values may be inspected. Figure 10 contains such a declaration view.

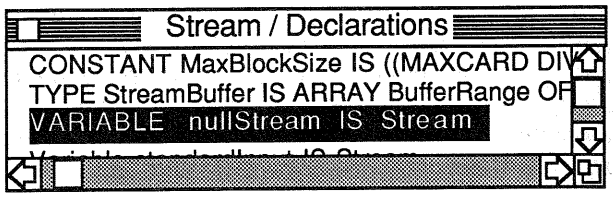


Figure 10 A declaration view with a variable selected

Program State Modification

It is very inconvenient for the user to have to supply complete state information prior to the execution of a code segment. Since it is not until runtime that the required set of state information can be determined, this is when it should be supplied. This eliminates the need for specifying extraneous information. Two examples of state information are the values of uninitialized variables and the values to be provided during input operations. A graphical interface provides a simple mechanism for this information to be supplied as it is needed. Each time an uninitialized variable or input operation is encountered, the user is asked for the value. This can be done through a dialog window which pops up, accepts a value and disappears.

For example, if the highlighted code of figure 9 is executed, the dialog box of figure 11 is generated. If the user enters a value of "2" for "x", then the ELSE clause of the IF statement will be executed and the user will not have to specify an input value for the read statement.

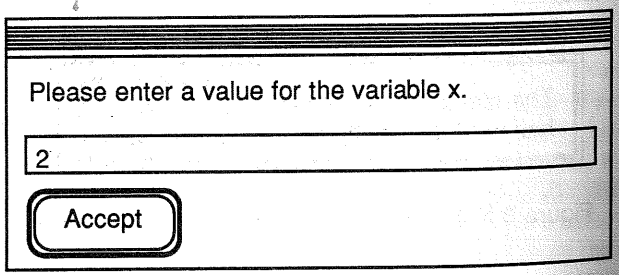


Figure 11 A dialog box for an uninitialized variable

A dialogue window can also be used to change the value of a variable during pauses in execution. For example, at the pause displayed in figure 9, the user may elect to change the value of "y" before continuing with the execution. The symbol "y" is chosen and the value is changed in a dialog box.

Conclusion

Graphical user interfaces can have a profound effect on both programming environments and the programming languages which they support. The traditional textual syntax of a programming language should be replaced by a more general graphical syntax which takes advantage of the graphical interfaces. Alternate syntaxes should be developed for existing languages and new languages should be designed with graphical interfaces in mind.

Detailed design can now be incorporated into programming environments through manipulation of symbols which represent program modules. Both dynamic and static program structure can be manipulated at a high level.

Graphical user interfaces simplify the problems of incremental compilation by allowing multiple windows which can hold program fragments in various states of correctness. It is now possible to construct a usable environment in which only code which is both syntactically and semantically correct is allowed in the program at any time. Any incorrect text which must be manipulated can be viewed in alternate windows.

The runtime debugging environment is the beneficiary of most of the graphical interface work to date. Graphical interfaces provide a good mechanism for selecting arbitrary code fragments for execution. They also provide the means to selectively view and modify many aspects of the program state.

The graphical features described in this paper are appealing and environment designers are in the process of incorporating them. In many ways, this is reminiscent of the situation in programming languages a few years ago. At this point we would like to insert a word of caution. Designers should remember two words as they charge ahead: "feature interaction". For example, the many programming language features introduced by Ada are very appealing to programmers. However, there is a cost associated with them. Features such as default parameter values and labelled parameters interact in subtle ways. [11]. We are currently studying feature interaction in programming environments as we are implementing the GUIDE programming environment.

References

[1] Tim Teitelbaum and Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", CACM (September 1981), pp 563-573.

[2] W. Hansen, "User engineering principles for interactive systems", Fall Joint Computer Conference (1971).

[3] Steven P. Reiss, "Graphical Program Development with PECAN Program Development Systems", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, (May 1984), pp 30-41.

[4] Duane Szafron, John Adria and Brian Wilkerson, "GUIDE: An Environment for Software Design," INFOR (January 1985), pp 31-52.

[5] Duane Szafron and Brian Wilkerson, GUIDE: Preliminary Users' Manual, Programming Languages Group, Department of Computing Science, University of Alberta, forthcomming.

[6] Nazim H. Madhavji, Luc Pisonneault, Surajit Choudhury and Nathan Friedman, "The MUPE-2 Programming Environment Project: An Overview", Proceedings of the CIPS ACI Congress '85, pp 372-382.

[7] Edward Yourdon and Larry Constantine, Structured Design, Prentice-Hall, (1979).

[8] Peter Naur, "Revised Report on the Algorithmic Language ALGOL 60", CACM, (January 1963), pp 1-17.

[9] I. Nassi and B. Schneiderman, "Flowchart Technique for Structured Programming, SIGPLAN Notices, (August 1973), pp 12-17.

[10] M. B. Albizuri-Remero, "A Graphical Abstract Programming Language", SIGPLAN Notices, (January 1984), pp 14-23.

[11] Bruce J. MacLennan, Principles of Programming Languages: Design, Evaluation and Implementation, Holt, Reinhart and Winston, (1983).

[12] C.N.Fisher, Anil Pal, Daniel L. Stock, Gregory F. Johnson and Jon Mauney, "The POE Language-Based Editor Project", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, (May 1984), pp 21-29.