PB - 187 K. Nørmark: Program Development on Graphical Workstations

ISSN 0105-8517

Program Development on Graphical Workstations

Kurt Nørmark

DAIMI PB - 187
December 1984

Computer Science Department
AARHUS UNIVERSITY
Ny Munkegade – DK 8000 Aarhus C – DENMARK
Telephone: 06 – 12 83 55

TRYK: DAIMI/RECAU

Program Development on Graphical Workstations

Kurt Nørmark

Computer Science Department, Aarhus University, Aarhus, Denmark

Abstract.

The aim of this paper is to emphasize the merits of progr.m development on graphical workstations. A workstation with a graphical display and a pointing device is an interesting base for new programming environments. Sufficient machine resources are provided to support advanced software development tools, and a considerable improvement of the programmer-machine interaction is possible. Program editing on graphical workstations is discussed. Using windows, a program can simultaneously be presented at several abstraction levels and hereby good program survey and rapid program navigation is provided. We discuss how to cope with the large set of commands in syntax-directed editors, and how to create programs using the pointing device as the primary input medium. Testing incomplete program sketches by simulating unimplemented facilities via so-called state editing is explained. Simple graphical means seem to be useful when observing the program state. It is argued why continuous program observation combined with programmer controlled execution speed (including reverse execution) is valuable. Examples are drawn from the Ekko design—a proposed integrated program development system for graphical workstations—and from an implemented syntax-directed editor on a Perq workstation.

Note. This paper is to appear in the proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences, Honolulu, Hawaii, January 2-4 1985.

1. Introduction.

The aim of this paper is to point out the advantages in applying a graphical workstation as a host for program development systems. Most of the paper is based on research carried out in the Ekko project at Aarhus University in Denmark during 1982 and 1983. In the Ekko project a concrete system has been designed, but instead of a detailed presentation of Ekko in this paper we will focus on some areas of program development, where a graphical workstation seems to be an interesting alternative to more traditional hardware. As will appear, most of the ideas presented in the paper can be found in some already implemented systems, but we hope that this paper will be a small contribution to a more coherent exploration of the subject. Several examples will be drawn from the Ekko design as well as from other similar systems. Notice that only a very small part of the Ekko system is implemented. The implemented part of the systems is briefly illustrated and described in the appendix.

Today, software is typically developed and used on the same machine. The program development system consists of some primitive tools linked together in a general purpose operating system. Some important problems of such program development systems can be summarized in the following points:

(1) The user interface of the program development system is purely character-oriented preventing an adequate interaction style.

(2) Too little knowledge about the programming language is present in the program development tools.

(3) The integration among the program development tools is not sufficient.

(4) The incrementality of the tools is too small.

An *integrated* set of program development tools must be co-operating, it must be easy to switch between using the different tools without losing accumulated information and finally the tools must have a common user interface. And a tool is said to be *incremental*, if it is able to perform a task in several smaller units, and thereby obtain a result similar to the result obtained, if the task was performed in its entirety.

In our opinion, software should be produced using advanced program development systems in the same way that other advanced industrial products are produced in highly specialized environments. When the software is complete, it should be possible to move it to a target machine of another kind than the program development machine. The same approach is encouraged in Ada Program Support Environments[17].

Before we proceed, we will give a characterization of a graphical workstation. A *graphical workstation* is a single-user machine having a graphical raster display, a pointing device (hereafter called a mouse) and a keyboard. The processor should nearly be as powerful as a mainframe processor and a large primary as well as secondary memory must be available.

Several existing program development systems and isolated tools are hosted on various graphical workstations. Smalltalk-80 can be considered an object oriented program development system[7]. The tools in the system are well integrated and the user interface is largely based on windows and menus. Smalltalk is currently running on for example the Xerox Dorado and Dolphin workstations. Interlisp[9,19] has been, for more than a decade, one of the most advanced programming systems used in the scientific community. The Programmer's assistant, Masterscope and "Do What I Mean" are some of the well-known Interlisp tools. A version of Interlisp called Interlisp-D[9] is running on the Xerox 1100 series graphical workstations. Here all the ordinary Interlisp tools are present, together with some display oriented facilities for editing and data inspection. At Brown University a program development system called Pecan[16] is running on Apollo graphical workstations. Pecan provides many different views of programs and data, for example, a textual view of a program manipulated via a syntax-directed editor, a graphical flow chart view and a symbol table view.

As already mentioned, Ekko[2] is an integrated program development system designed specifically for a graphical workstation. Ekko supports development of Pascal programs during syntax-directed editing, dynamic examination tools and tools for program administration. The syntax-directed editor is implemented on a Perq workstation (see the appendix), and the entire system is described in detail in reference 2.

In the rest of this paper a more detailed discussion of program development on graphical workstations will be carried out. In section two some general advantages of applying a graphical workstation during program development will be presented. In section three program editing on a graphical workstation will be considered, and in section four and five program testing and program state observation is discussed.

2. General Advantages of Applying a Graphical Workstation.

In the introduction we postulated some problems of traditional program development systems, and we described some properties of a graphical workstation. In this section we will explain why the properties of a graphical workstation form one way to overcome some of these problems.

As has been demonstrated in, for example, Smalltalk-80 and Interlisp-D the graphical display and the mouse together form the basis of a qualitative improved user interface. The now widely used window concept, pioneered in the development of these systems, turns out to be an important way to achieve integration among the program development tools. A window visualizes an activity by constituting a frame around it, and the activity is still visible and present while other tasks are performed in other windows. Furthermore, simple graphical means can replace the exclusive usage of text in today's program editing, and a much more direct and natural interaction style becomes possible by allowing selection and manipulation of program elements on the display using the mouse.

A powerful processor is desirable for incremental execution of some tasks during interactive use of the system. If, for example, static semantic analysis or code generation is to be carried out during interactive editing in an incremental way, the editor's response time should not increase above some upper limit.

And finally, much memory is needed to support the graphical capabilities and to represent programming language information. Furthermore, in an integrated system much memory is necessary to store the state of temporarily suspended activities.

In the following sections, usage of a graphical workstation in more specific functions of the program development process, will be discussed.

3. Program Editing.

In this section we will discuss advantages of doing program editing on a graphical workstation. A *program editor* we consider as an editor knowing some of the formal rules of a programming language. One particular class of program editors is the *syntax-directed editors*, where syntactic program constructs rather than pure text are manipulated by the editing commands. The majority of existing program editors are implemented on computers having traditional interaction hardware: a character oriented display and a keyboard as the only input and output devices. The editors in Mentor[4], the Cornell Program Synthesizer[18] and Aloe[11] are all examples of such editors. But recently, several program editors have also been designed for various graphical workstations, e.g., the Pecan syntax-directed editor[16], the editor in Magpie[3], the GLSE editor[10] and the Ekko syntax-directed editor[2]. It is furthermore interesting to notice that Emily[8], the very first syntax-directed editor reported in literature, was based on equipment having a graphical screen (a vector technology display) and a pointing device (a lightpen).

In this section we will first thoroughly discuss various program presentation techniques. Then a note is given on the large command repertoire in especially syntax-directed editors. After that, we will describe techniques whereby the mouse can be used even more extensively than in most existing syntax-directed editors. Finally, some screen updating problems, and programming language comments in a syntax-directed environments, will be treated.

3.1. Program Presentation.

Basically, the bitmapped raster display of a graphical workstation is well suited for representing a large amount of textual as well as graphical information. But even the largest existing display can only contain programs of modest size in full detail on the display. Therefore, some kind of technique is necessary to select that subset of the program which is to be presented on the display. Two techniques are widely used:

(1) Windowing: The program is seen through a window which can be moved across the program (or equivalently the program can be moved relative to a fixed and non-movable window).

(2) Holophrasting: Certain program constructs are elided and short "cover names" are shown instead. The eliding can be automatic or controlled by the user.

In the first approach an unbroken and relatively small section of the program is presented in full detail on the display. Using the second approach a larger part of the program can be shown, but not in full detail. This approach requires that the editor knows the syntactic structure of the program.

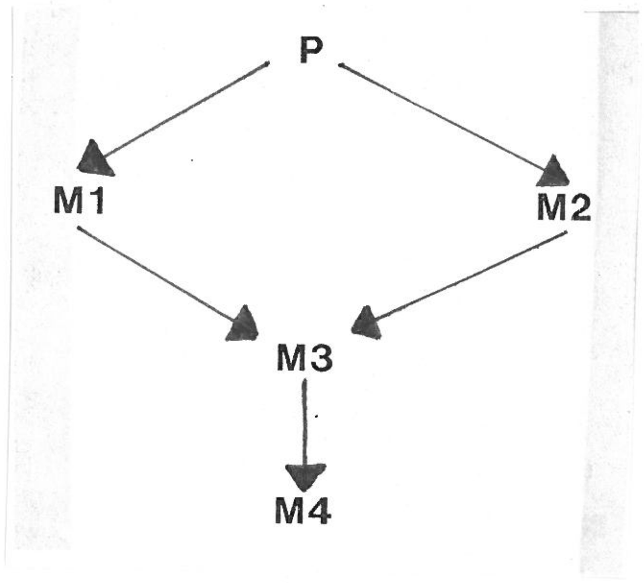


Fig. 1. Presentation of a program P and four modules, M1–M4. An arrow indicates "imports from".

We will now describe a more general program presentation scheme, which seems to be attractive, when the host of the editor is a graphical workstation. Let us introduce the technique by considering an analogy not bounded to the computer area.

We want to get some geographical information about a small country in Europe. Instead of traveling around in the world to find this information, most people would probably use an atlas to observe the information at an appropriate level of abstraction. In the "physical world" too many irrelevant details are present, and it is impossible to get a survey of the desired structure. In the atlas a map can be found, where nearly all details except the "country structure" are suppressed. If we start by looking at a world map then Europe can be located, but perhaps not the small country we are looking for. We then select a detailed map of Europe, and as a third step a detailed map of the desired country might be found. That is, by turning some leaves in the atlas the desired geographical information can be observed at various abstraction levels.

Having this analogy in mind, we will now describe some general properties of a program presentation scheme. The key problem is selection of abstraction levels for program presentation fitting the actual needs of the programmer. If the programmer wants to see the entirety of a program having many modules and hundreds of procedures, then nearly all details, except the module structure of the program, have to be suppressed in the presentation. A graph showing the modules and their import-export interrelations would probably be appropriate, see fig. 1.

Having provided the possibility of program presentation at several levels of abstractions, some means should be available for the programmer to select a construct at one abstraction level, and request presentation of this construct at another (or the same) level of abstraction. The two presentations should be shown simultaneously on the display, which is easy on a graphical workstation using two different windows. According to the map analogy, one useful application is selection of a construct at a high abstraction level in order to observe a more detailed presentation of the construct at a lower level. If, for example, the programmer wants to see some details of a particular module in fig. 1, then it should be possible to present this selection at lower level of abstraction in a new window, using the traditional textual program presentation or a presentation showing only the procedure structure of the module. But also the opposite direction may be useful, i.e., to select a construct presented in detail, and have this construct presented in another window, suppressing some of the details to provide a better survey. In our opinion, simultaneous program presentation at different abstraction levels is important, because it allows presentation of both a survey of a large piece of software and detailed views of some selected constructs at the same time. Furthermore, it provides a fast and flexible way of program browsing, similar to browsing known, for example, from Smalltalk-80[7].

Finally, it should be possible to modify a program at each abstraction level of program presentation. Forcing the programmer to make all modifications at one particular level of abstraction, say the textual level, is not satisfactory. It means, for example, that it should be possible to add and delete import-export relations in fig. 1 solely by manipulating arrow symbols, rather than stating textually in another window that "M1 imports M2". It is important to remember that each window is just a view of (some piece of) a program. That is, modifications in one window can affect the contents of several other program presentation windows.

The program presentation and modification scheme described above is fairly general. We will now consider some special cases of the scheme found in some existing program editors on graphical workstations.

A particular simple case emerges, if a program can be shown in different windows using the same abstraction level of program presentation. One useful application of this technique is to show some declarations of names in one window, while the context of some applied occurrences of the declared names are shown in other windows. Hereby it becomes possible to look at the definitions of names, while they are used in parts of the program far away from their definition. Support[21] is an example of a program editor allowing data definitions and control to be shown in different windows. Support is, among other machines, implemented on a Sun workstation.

In Pecan[16] a program can be seen through several so-called *views*: An ordinary textual view, a flow chart view, a textual view of declarations and an expression view showing the tree-structure of an expression. The program can be modified through the textual views, whereas the flowchart view and the expression views—at least in the implementation reported in reference 16—are read only. All the views in Pecan show the program at rather detailed abstraction levels. The expression view is even more detailed than the textual presentation of an expression. Views at higher abstraction levels such as "procedure-level connection diagrams" are only shortly discussed in reference 16.

In Ekko[2] a program (and the modules on which the program depends) can be presented at three abstraction levels: At the module composition level, at the procedure composition level and at the body level. The program can be modified at all three levels, and this is in our opinion very important. Exactly three windows are used, one for each abstraction level. At the module composition level a graph—like the one in fig. 1—shows the modules and their import-export relations. At the procedure composition level a tree shows the procedure structure of the module (or main program) selected in the module composition window. And at the body level a textual presentation of the procedure selected in the procedure composition window is presented. Textual presentation of local procedures via textual nesting is omitted in the body window, because this information is presented better in the procedure composition window. A similar approach, using textual program presentation at all abstraction levels, is present in Loipe[5]. The Ekko approach of program presentation on the screen resembles somehow the holophrasting approach. Some syntactic details are elided in the program presentation at the higher level, but the selected "holophrast" is automatically expanded and presented at the lower abstraction levels. The choice of the elements in the various abstraction levels determines a fixed syntactic elision scheme.

Until now we have only been concerned with presentation and editing of one program at a time. In the general case it should be possible to edit several programs simultaneously in a co-ordinate manner. Hereby it becomes easy and attractive to transfer some program constructs from one program to another. This is very important for reusability of already existing programs and program fragments. Still, we want to retain the possibility to present each program in several windows at different abstraction levels as described above. So, one means or another should make it possible to distinguish between different programs. If the workstation has a color display, each set of windows presenting a particular program can be shown using a particular background color.

In the Ekko system, a program fragment facility constitutes a rather limited elaboration of a multi-program editor. In a fragment window several program fragments can be defined. A fragment is not named, because it is identified as a prefix of itself, using a very small font (see the figure in the appendix). The selected program fragment can be modified as it is possible to modify the working object. The program fragment facility is a flexible equivalent to the traditional register facility known from many text editors. However, the general approach having several co-ordinate programs in the editor should be preferred, rather than one special program of interest together with a collection of program fragments. In Ekko it is, for example, not possible directly to insert a fragment into another. This seems to be an unnatural limitation.

3.2. Command Repertoire.

A syntax-directed editor is characterized by a very large set of commands. Some of the commands are directly derived from the programming language while others are independent of the programming language.

Each construct in the programming language gives rise to one or more commands, and it turns out that more and more commands are added to new syntax-directed editors in order to achieve more flexible modification facilities (e.g., the nest and transform commands in Aloe[11]). In Ekko, approximately 170 different programming language dependent commands are possible for the Pascal dialect on the Perq. It is very difficult or nearly impossible actively to remember the names of so many commands, even if some kind of systematic naming of commands is used. Using command menus, the user should not actively be able to enter a command name. It is sufficient if the user is able to deduce the semantic value of the command from the syntactic command name in the menu. It is well-known that it is difficult to grasp a menu with many fields. Therefore it is important to organize the system in such a way, that only a few menu fields are visible at any time. Ekko uses postfix syntax (i.e., the command follows the argument selection) and therefore the system is able to present only the legal commands at any time. The longest command menu in Ekko for the Perq Pascal dialect contains 34 commands (concerning nesting in different expressions), but the average length of command menus is much smaller.

Cursor movement commands are examples of programming language independent commands. If the keyboard is the only input device, text oriented commands or function keys have to be used for cursor movement. In the Cornell Program Synthesizer[18] 13 different cursor movement commands are available. When a pointing device is present, it is easier to select a new construct by pointing at some syntactic entity logically associated with the construct. An array type, for example, can be selected by pointing at the keywords "array" or "of" in Pascal. This technique is *absolute navigation* whereas the former is *relative navigation*. Absolute navigation is only possible if the new construct is visible on the display. Therefore a scrolling facility is also necessary if absolute navigation is used. In relative navigation scrolling can be activated implicitly when the cursor is located near the border of a window.

3.3. Programming by the Mouse.

Because the graphical workstation has two input devices (a keyboard and a pointing device), the user typically shifts between using the two devices. Some users consider such a shift as a discontinuity in the interaction. Therefore it seems wise to use one of the two input devices as the primary one, in order to minimize the shifting. It is relatively unproblematic to design a system where all actions can be issued via the keyboard, for example using various control sequences. We will in this section discuss the other extreme: maximal use of the mouse.

We have already mentioned several advantages in moving the cursor and initiating editing commands using the pointing device. But still a considerable part of the total amount of input is specification of user defined names and literals. The keyboard is well suited to enter the first occurrence of a name, but then the following occurrence can be specified via the mouse by referring to an already existing occurrence of the name. Typing only the first occurrence of an identifier reduces the number of primitive input actions considerably, and misspelling is nearly eliminated.

Several syntax-directed editors, for example Emily[8] and Pecan[16], collect already used names in a menu, which is presented when an identifier is expected. Some context information could be used to reduce the length of such a menu. In Ekko, an already existing occurrence of a name or a literal can be copied directly from one place in the program to another. (Select the existing occurrence of the name and while the button on the mouse is pressed, move the mouse slightly and the name will be copied). By referring to a scoped occurrence of a name instead of a name in a menu, the name binding process of the static semantic analysis (i.e., the process of binding applied names to the defining occurrences of the name) could furthermore be controlled directly by the programmer. The programmer is supposed to enter the defining occurrences of a name via the keyboard, and all the applied occurrences of the name by referring to the definition of that name. However, several interesting problems arise. The proposal requires definition before use, not textual but on the time scale. And it would be possible to break the normal scope rules of the programming language if the same name is used in different scopes, and if an inner name occurrence is bound to the outer meaning of the name. Finally, it is not clear how to visualize the actual binding between an applied name occurrence and a binding occurrence of the name. Anyway it can be seen, that by adapting an adequate editing discipline (perhaps somehow restricting the freedom of the programmer) more secure program editing is possible.

3.4. Updating the Screen.

Experiences from the Ekko project and from an earlier implementation of a syntax-directed editor on a timeshared computer in Aarhus[14], show that it requires many computing resources to obtain a "real screen oriented interface" of a syntax-directed editor. The reason is mainly the substantial structural difference between the internal program representation (a tree) and the usual textual screen presentation. The implementor of a syntax-directed editor has to be careful that (1) the screen updating process following a program modification and (2) the selection of a particular construct by pointing at it using the mouse, both can be carried out within reasonable response times. A powerful processor in a graphical workstation will help by making these two kinds of operations fast enough.

The screen updating can either be incremental or total. In both cases "screen flicker" should be minimized. In Ekko we use a total rewrite of the working object window for every editing action, and screen flicker is avoided by writing the program into an intermediate buffer, which is copied to the screen in one RasterOp operation[13]. Ekko traverses the whole syntax tree for every screen rewrite, but this is too slow when the program becomes considerably larger than the working object window. Either a smaller tree should be selected as the base of rewriting, or an incremental screen updating technique should be adopted.

Fast response times are essential for selection of a program construct with the mouse, in the same way that fast response times are essential when activating a key on a keyboard. In Ekko, the position of every program construct is stored in association with the nodes in the abstract syntax tree, and every program modification leads to a recomputation of this information. Selection of a program construct activates a simple searching procedure of complexity $O(h)$, where h is the depth of the abstract syntax tree.

3.5. Comments.

As the final subject in this section we will discuss comments in a syntax-directed programming environment. As has been discussed in several papers[6,11,18], comments as lexical elements cause problems in editors where the syntactic constructs are the elements of manipulation. Only those comments, which easily can be associated with syntactic structures in a program, can be handled satisfactory of existing syntax-directed editors. But still there are problems, for example concerning placement of the comments during pretty-printing. A classification of comments might help solving some of the problems, but in our opinion, the syntactic oriented comment concept is not satisfactory. It would be of great value to have a similar free comment facility in a syntax-directed programming environment, as we know from pure text-oriented programming environments.

In Dice[6], two classes of comments are identified: left comments and right comments. A left comment of a program construct is printed before the program construct. A right comment is regarded as an annotation of the construct, and it is printed to the right of it.

In the Cornell Program Synthesizer[18], the comment concept and the holophrasting mechanism are coupled together. Certain program constructs can associate a comment, and the program construct itself can be elided leaving only the comment on the screen.

Also in Ekko, a comment is associated with a syntactic structure in a program, and therefore we call it a *structure comment*. The facility is designed to provide a clean user interface easing program development by stepwise refinement[20]. A structure comment can be associated with every program construct corresponding to a node in the abstract syntax tree (except identifier and literal nodes). Just select the node, enter the text and the structure comment will substitute the program construct on the screen. A characteristic font is used on the screen of the graphical workstation, to make clear that the text is a structure comment (see the picture of the screen in the appendix). Two commands allow alternating views of the structure comment and the underlying program construct, but both cannot be presented at the same time as in the Cornell Program Synthesizer. The structure comment associated with a construct does not impose any restrictions on the editing commands on the construct. Therefore the structure comment concept makes it attractive to develop programs in a top down manner by stepwise refinement. An unexpanded structure comment can be refined in exactly the same way as the underlying unexpanded placeholder. The most important drawback of the structure comment concept in Ekko is, that it cannot be associated with a sublist, which does not correspond to a node in the abstract syntax tree. It is not clear what to do when the commented list is extended or reduced, but it would probably be an advantage to impose a hierarchical structure too on "flat lists" in a program. In this way a comment can be associated with an arbitrary node in the list hierarchy.

4. Program Testing.

Program testing is the process of examining whether the program realizes the vision of the programmer, i.e., to determine if logical errors exist in the program. Testing any non-trivial program is an incremental process, because several sub tests of different program aspects are to be performed to ensure the correctness of the whole program. In this section, tools supporting two different incremental program testing techniques will be discussed. The testing techniques to be treated are a *top down testing technique* and the more traditional *bottom up testing technique*.

Consider program construction by stepwise refinement as creation of a sequence of *program sketches* ranging from a very rough draft (only containing the main structures of the program) to the final and detailed program. In order to examine if the already implemented aspects of a program sketch are in harmony with the vision of the programmer, it is desirable to be able to execute each of these program sketches. The problems to cope with are the incomplete nature of the program sketches, for example, what to do if meeting an undeveloped placeholder, an unrefined structure comment, an undeclared variable or an obvious incorrectness during the execution. In our opinion, it is important not to force the programmer to refine the program in a way, dictated by the order in which the incompletenesses are met during testing. Instead, tools in the program development system should support simulation of unimplemented facilities in program sketches. The success of the described top-down incrementality of the program testing process is critically dependent on, whether a facility tested in one program sketch remains to be correct in the succeeding sketches.

In the bottom up testing technique, a piece of program P is created using already tested and hopefully correct facilities. Then P is tested by activation of the facilities in P, in such a way that the correctness of P is ensured too. A typical practical problem in this kind of testing is creation of an appropriate test context, e.g., creation of a collection of data objects on which a new set of procedures and functions can operate. But also the activation itself of the new facilities may cause problems. Very often a new program is constructed with the only purpose of testing the facilities in P.

Bottom up testing can in a natural way be done in systems supporting dynamic languages like Lisp and Smalltalk, and several systems for static languages like Pascal have adopted techniques allowing more flexible bottom up program testing. In Magpie[3], for example, a so-called *workspace* can be created. A workspace is an anonymous procedure in which testing actions easily can be initiated via a "Do It" command. A workspace is presented on the display in a particular window. Conceptually, a workspace can be nested into any procedure and hereby gain access to the local declarations in the procedure.

Top down testing is most obvious in systems supporting top down program creation using, for example, a syntax-directed editor. The Cornell Program Synthesizer[18] allows execution of incomplete programs, but the programmer is expected to implement the missing facilities as they are met. In the Ekko design it is proposed, that unimplemented facilities in a program P may be simulated by executing one or more program fragments. For example, invocation of an unimplemented function may be simulated by specifying a reasonable value of the function call, and a procedure call may be simulated by changing the value of the global variables, on which the procedure has effect. The execution of a program fragment modifies the program state of P, but P itself is not modified during the modification of the state. The fragment may invocate procedures defined in P as well as procedures defined in other program fragments. In the latter case the procedure is considered local to the most recent procedure activation. Changing the point of control could be done by execution of a goto fragment and by adding a label into the program itself, but it is not satisfactory to modify the program in order to edit the program state. Therefore, direct modification of the "program counter", by selecting the next statement to be executed using the mouse, is much more attractive. This technique, furthermore, allows the programmer to simulate boolean expressions in selective and iterative control structures. For example, the value "true" of the expression in an if-then-else statement may be simulated by pointing at the statement in the then-part before execution is resumed (assuming that the expression has no side effects).

In general, what is needed to carry out testing of a program is some kind of a *program state editor*. Using such a tool it should be possible to modify and augment the program state in a unified manner. Both simulation of missing facilities and creation of new test contexts should be carried out using a program state editor. Of course, editing of the program state can be carried out through the programming language, but it is interesting to imagine a real screen oriented "What you see is what you get" state editor. After all, the programming language is an indirect way to access the state of a program, just like the commands of non-screen oriented text editors are an indirect way of manipulating a textual document. It should be noted, that not only creation and modification of the program state is possible via such a program state editor, but inspection and observation can be carried out as well using this tool. In our opinion, designing a program state editor for a graphical workstation is an interesting subject for further research.

5. Program State Observation.

The output of a running program reflects that part of the program state which is essential to carry out the program testing process. But often it is necessary to get knowledge about some non-visible parts of the program state too, e.g., in order to locate the cause of an error. We distinguish between the *external program behavior*, which is visible through the output of the executing program, and *internal program state* reflected by the state of the data objects, the current point of control and the procedure call stack. In many traditional program development systems it is possible to observe the internal program state via a debugger.

In this section we will first discuss observation of the internal program state using a graphical workstation. Secondly, we will treat the problems occurring if both the internal program state and the external program behavior have to be shown simultaneously on the screen.

5.1. Observation of the Internal Program State.

Two parameters, among others, characterize observation of the internal program state: The abstraction level of the observation and the chronological correspondence between the current program state and the state shown on the screen. At least three abstraction levels of program observation may be distinguished:

(1) The abstraction level of the problem being solved. The concepts to be presented during program state observation at this level are closely related to elements in the problem.

(2) The abstraction level of the programming language used to solve the problem. Programming language concepts like arrays, records and procedures can be observed.

(3) The abstraction level of a low level programming language used to realize the high level program. Concepts associated with the target machine like "program counter" and "machine cell" are important at this level.

In our opinion, graphical means are well suited for presentation of the internal program state at level (1) and (2). Several windows can be used to show different aspects of the program state. The Incence system[12] has interesting facilities for graphical presentation of data objects on a graphical workstation.

The two extremes of the second parameter, the chronological correspondence between the current state and the state shown on the screen, are

(1) Continuous program observation: Every change in the internal program state, relevant at the selected level of abstraction, is automatically reflected on the screen.

(2) Discrete program observation: Predetermined or programmer defined snapshots of the program state can be observed on the screen.

The possibility of continuous observation of the internal program state is powerful in order to carry out effective debugging of a program. It is likely that the programmer during execution of the program catches an anomaly in the internal program state immediately, simply by observing the evolution of the program state. As the state of the program changes during program execution, the contents of the program observation windows should be changed too, according to the continuous observation approach. Therefore a high bandwidth between the processor and the screen of the program development machine is necessary, in order to make continuous observation of the internal program state realistic. A graphical workstation with a memory mapped display is in that respect superior to an ordinary graphical terminal running on a host via a low speed communication link.

Continuous program observation of any non-trivial program results in an overwhelming amount of information on the screen. The success of a program observation tool, therefore, critically depends on the facilities to control the observation in various ways.

Continuous program observation can be carried out in both the Cornell Program Synthesizer[18] and Loipe[5]. In the Ekko design[2], continuous observation of the internal program state at the abstraction level of the programming language is proposed. The values of some data objects are presented in a particular window at the screen. Composite data objects are presented in nested boxes, instead of using purely textual denotations like Ada aggregates[15]. In another window the procedure call stack is shown. The point of control is indicated at the module composition level, at the procedure composition level and at the body level in the three program description windows (the windows also being used in program editing, as described in section 3.1).

Several mechanisms can be used to control the internal program state observation in Ekko. The user can select which data objects are to be observed. The systems shows per default a (non-fixed) number of the most recent updated data objects. But the user of the system should be able to request observation of data objects of special interest. Some parts of the program may be uninteresting with respect to observation of the internal program state.

Thus, certain units of a program (procedures or modules) may be closed in such a way, that the flow of control and the local data objects in these units do not disturb the observer during program observation. Another means to control the grain of monitoring is to close one or more of the monitoring windows. If, for example, the detailed flow of control is uninteresting, the window giving this information may be closed. In this case, the flow of control still can be observed at the module and at the procedure composition levels.

During continuous program observation, it is important to be able to control the speed of the executing program and hereby the rate on which information is presented in the observation windows. Things typically happens too fast, even in systems where only the external program behavior can be observed. A "program execution speeder", allowing changes in the execution speed at any time during execution, would be useful. The speed control can be generalized to allow "single-step speed", zero speed and even negative speed. The latter possibility is a simulated reverse execution facility known from, for example, the Cornell Program Synthesizer[18] and from Cope[1]. Reverse execution is especially useful if continuous program observation of the internal program state is provided, namely in order to compensate for the human reaction time, in this case the period of time from an anomaly is identified, to the user have managed to stop the execution. The possibility of reverse program execution is based on information stored during "forward" execution. In Ekko, we have proposed to organize this information on a stack. The reverse program interpreter uses the information from this stack, together with the program itself, to restore previous program states, (see reference 2 for more details).

5.2. Observation of the External Program Behavior.

The presentation of the external program behavior, i.e., the program output, is normally not a problem during program development. But if the same screen simultaneously is to be used for both presentation of the internal program state and presentation of the program output, it may be difficult to test a program which uses the whole screen for program output. A very large screen, or a program development system having two independent screens, would naturally solve the problem, but it is interesting to consider solutions, where the program output during program development is presented in a window smaller than the screen of the target machine.

Several techniques can be used to remedy the lack of physical screen space. In the case of purely textual output, the well-known line folding and vertical scrolling technique can be used. But this solution is not appropriate for general graphical output. As a second solution, the entire output screen can be shown on a reduced scale, so that it fits into the output window of the program development system. This solution requires a very high screen resolution, probably together with special hardware to make the transformations fast. Finally, a virtual display of the same size as the screen of the target machine may be allocated in the program development system. In this case, program output is written into the virtual display, and part of the virtual display is continuously mapped into a smaller output window on the screen. A scrolling facility may be used to determine the actual mapping.

The latter solution is preferred in the Ekko design, although some kinds of interactions become awkward, e.g., dragging a screen element from one location to another, outside the visible area of the virtual display. The virtual display is technically attractive, because many workstations have a very fast so-called RasterOp operation[13] well suited to copy a rectangular area from the memory to the screen.

6. Concluding Remarks.

We have in this paper discussed the usage of a graphical workstation in the program editing, the program testing and the program debugging processes. The graphical workstation is a flexible and powerful workbench for the professional programmer. The improved integration among the program development tools, and performance of routine work by the program development system rather than by the programmer, is important for such users. The improved user interface of the program development system on a graphical workstation is probably most important for novices, but it is also pleasant for more skilled users.

Using a high quality program development system, the productivity of a programmer will probably increase because some time consuming tasks will vanish. But also from the programmers point of view, the quality of the working-day will be improved, because more time now can be used for meaningful problem solving instead of trivial routine work.

Besides the topics treated in this paper, the program administration problems are very important. It would be interesting to design program administration tools, taking care of the numerous relations among the information in the program development process. The capabilities of a graphical workstation will undoubtedly be of great benefit in such tools too.

Acknowledgments.

Most of this paper is based on ideas from the Ekko project, carried out by Karen Borup, Elmer Sandvad and the author of this paper. I am grateful to Karen and Elmer for their co-operation during the Ekko project. I would also like to thank Ole Lehrmann Madsen, Bent Bruun Kristensen, Nazim H. Madhavji, Brad Hartfield and the referees for useful comments during the preparation of this paper.

References.

1. James E. Archer Jr, Richard Conway, Fred B. Schneider, "User Recovery and Reversal in Interactive Systems", *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, January 1984.

2. Karen Borup, Kurt Nørmark, Elmer Sandvad, "EKKO—An Integrated Program Development System", DAIMI IR-51, November 1983, Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark.

3. Normann M. Delisle, David E. Menicosy, Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices*, vol. 19, no. 5, May 1984.

4. Véronique Donzeau-Gouge, Gérard Huet, Gilles Kahn, Bernard Lang, "Programming Environments based on Structured Editors: The Mentor Experience", Inria, Rapport de Recherches, no. 26, July 1980.

5. Peter H. Feiler, "A Language-Oriented Interactive Programming Environment Based on Compilation Technology", Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, May 1982.

6. Peter Fritzson, "Towards a Distributed Programming Environment based on Incremental Compilation", Dissertation no. 109, Department of Computer and Information Science, Linköping University, Sweden.

7. Adele Goldberg, "Smalltalk-80: The Interactive Programming Environment", Addison-Wesley, 1983.

8. Wilfred J. Hansen, "User Engineering Principles for Interactive Systems", *Fall Joint Computer Conference*, 1971.

9. "Interlisp Reference Manual", Xerox, October, 1983.

10. David B. Leblang, "Abstract Syntax Based Programming Environments", *Proceedings of the AdaTec Conference on Ada*, Arlington, Virginia, October 6-8, 1982.

11. Raul Medina-Mora, "Syntax-Directed Editing: Towards Integrated Programming Environments", Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, March 1982.

12. Brad A. Myers, "Displaying Data Structures for Interactive Debugging", Xerox, Palo Alto Research Center, June 1980.

13. William M. Newman, Robert F. Sproul, "Principles of Interactive Computer Graphics", Second Edition, McGraw-Hill, 1983.

14. Jakob Nielsen, Henrik Wendelbo Nielsen, Kurt Nørmark, Jan Sørensen, "EAGLE - a Syntax-directed editor, User's guide", (In Danish) DAIMI MD-45, Computer Science Department, Aarhus University, January 1982.

15. "Reference Manual for the Ada Programming Language", United States Department of Defense, Springer-Verlag, 1983.

16. 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*, vol. 19, no. 5, May 1984.

17. "Requirements for Ada Program Support Environments", "Stoneman", Department of Defense, February 1980.

18. Tim Teitelbaum, Thomas Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, vol. 24, no. 9, September 1981.

19. Warren Teitelman, Larry Masinter, "The Interlisp Programming Environment", *Computer*, vol. 14, no. 4, April 1981.

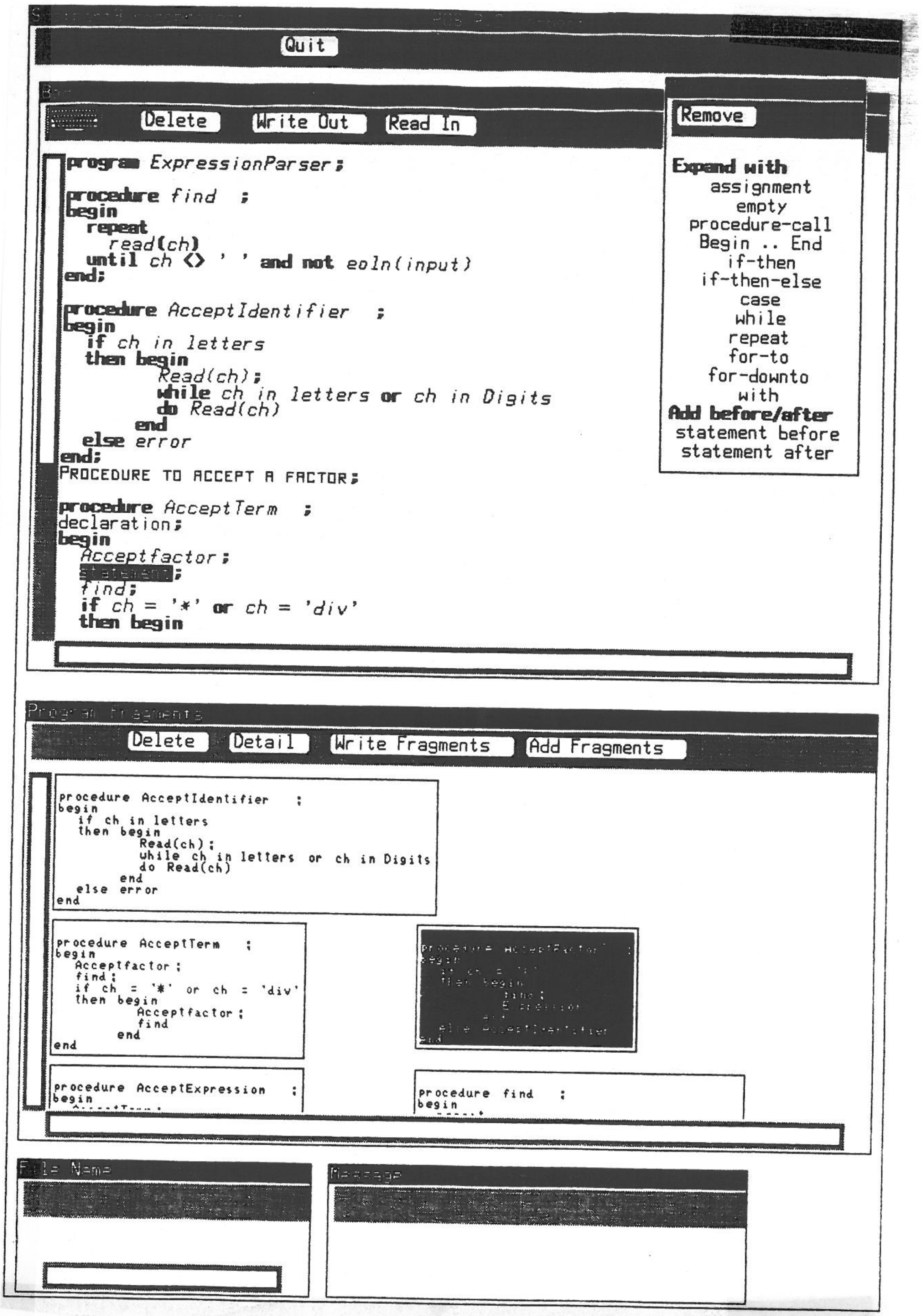
20. Niklaus Wirth, "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, April 1971.

21. Marvin V. Zelkowitz, "A Small Contribution to Editing with a Syntax Directed Editor" Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *SIGPLAN Notices*, vol. 19, no. 5, May 1984.

Appendix

This appendix contains a brief description of the implemented part of Ekko, which is a syntax-directed editor. In the upper window—called the Body window—a traditional textual presentation of the editor's working object is presented. The commands in the Command Menu operate on the selected statement placeholder. Three different kinds of programming language dependent commands are possible: expanding commands, list manipulation commands and nesting commands. The structure comment facility described in section 3.5 is implemented. The text "Procedure to accept a factor" is an example of a structure comment. Furthermore, it is possible to copy identifiers, literals and structure comments as explained in section 3.3. The keyboard icon indicates that keyboard input will be directed to the Body window.

The Program Fragment window can contain a collection of program fragments (see section 3.1). In the following illustration, the program fragment window is shown in survey mode. By activating the "Detail" command only the selected fragment will be shown in the window, and this program fragment can be modified exactly as in the Body window. Information is copied from one window to another by "dragging" it over the screen.



```
program ExpressionParser;

procedure find  ;
begin
  repeat
    read(ch)
  until ch <> ' ' and not eoln(input)
end;

procedure AcceptIdentifier  ;
begin
  if ch in letters
  then begin
          Read(ch);
          while ch in letters or ch in Digits
          do Read(ch)
       end
  else error
end;
PROCEDURE TO ACCEPT A FACTOR;

procedure AcceptTerm  ;
declaration;
begin
  Acceptfactor;
  statement;
  find;
  if ch = '*' or ch = 'div'
  then begin
```

```
procedure AcceptIdentifier  ;
begin
  if ch in letters
  then begin
          Read(ch);
          while ch in letters or ch in Digits
          do Read(ch)
       end
  else error
end
```

```
procedure AcceptTerm  ;
begin
  Acceptfactor;
  find;
  if ch = '*' or ch = 'div'
  then begin
          Acceptfactor;
          find
       end
end
```

```
procedure AcceptFactor  ;
begin
  if ch = '('
  then begin
          find;
          Expression
       end
  else AcceptIdentifier
end
```

```
procedure AcceptExpression  ;
begin
```

```
procedure find  ;
begin
```