A Highly Integrated Tool Set For Program Development Support

Gregor Engels
Informatik, Abt. Datenbanken
TU Braunschweig
Postfach 3329
D - 3300 Braunschweig

Thorsten Janning
Lehrstuhl für Informatik III
RWTH Aachen
Ahornstr. 55
D - 5100 Aachen

Wilhelm Schäfer
STZ
Software-Technologie-Zentrum
Helenenbergweg 19
D - 4600 Dortmund 50

Abstract

This paper describes the design of the integrated user interface of the software development environment IPSEN (Integrated Programming Support Environment). We explain the characteristic features of the IPSEN user interface, namely the structured layout of the screen, the command-driven tool activation, and especially the highly integrated use of the IPSEN tool set. We demonstrate those features by taking a sample set of tools of the IPSEN environment. That tool set supports all the programming-in-the-small activities within IPSEN. Finally, we sketch the realization of two prototypes running on an IBM-AT and a net of SUN workstations.

Key Words: software development environment, user interface, software tool,integration, programming-in-the-small

1 Introduction

This paper reports about results which have been achieved within the **IPSEN - project** (Integrated Programming Support Environment). The main objectives of the project which is mainly carried out at the University of Aachen are: (1) the development of new concepts for the design of an integrated Software Development Environment (SDE) and (2) the implementation of a prototype to prove the feasibility of the developed concepts. In fact, two prototypes of IPSEN have already been implemented on two different workstations (an IBM-AT and a net of SUN workstations). Those prototypes provide integrated tool sets to support all task areas of the software development process, namely Programming-in-the-small (PIS), Programming-in-the-large (PIL), Documentation (DOC), and Project-Organization (ORG).

The aims of IPSEN are **related to many other projects** in the SDE-field like e.g. /DK 84/, /Ha 82/, /MV 85/, /Re 84/, /RT 84/. However, IPSEN differs from most of them in the following points which are: (1) IPSEN provides tools to cover the whole software development process instead of just PIS as in e.g. /Re 84/, /RT 84/, (2) it is not directed towards a special program language like /MV 85/, and (3) a syntax-directed editor is not regarded to be the central tool of the whole environment with other tools being appendices to this initial starting point like in /DK 84/ or /Ha 82/. All tools provided by IPSEN were designed as a comprehensive tool set from the very beginning which results in a uniform internal realization of all tools as well as a very uniform user interface.

In particular, the design of a very sophisticated **user interface** was regarded to be one of the central points, because high user acceptance of a SDE can only be achieved by not only providing for appropriate tools but also by supporting a very easy and highly integrated use of them. Consequently, the IPSEN user does not have to handle all the single different tools, but he regards the whole environment as being just four very sophisticated powerful tools for the task areas PIS, PIL, DOC and ORG.

The **aim of this paper** is to describe mainly the design of the integrated user interface of IPSEN. As the example we use the area PIS, because it might probably be the most well-known one. Therefore, it is not necessary to introduce a lot of new concepts especially developed to support PIL, DOC or ORG. Those concepts are described in other papers more carefully (/LN 85/, /Le 88/, /Ja 87/).

In PIS, we chose Modula-2 as the language to be supported by IPSEN. Therefore, the tools described in this paper support the development of a program module or an implementation module (which will be called module later on) in the Modula-2 sense.

Although, the IPSEN user interface is the main topic of this paper, we also sketch the main ideas of the **realization** of this interface to show how the high integration of tools within IPSEN is achieved by the implementation as well.

The **organization of the paper** is as follows: In the next section, we describe the design of the "how-part" of the IPSEN user interface, i.e. the structured screen layout and the possibilities of command input by the user. This part is common to all tools and all task areas, because it does not assume a certain functionality of a given tool set. However, it is illustrated by using examples from the PIS area. To demonstrate how this design is applied to the tool set of a certain task area, the functionality of the IPSEN tool set for PIS is described more carefully in section 3, i.e. the "what-part" of PIS. In Section 4, the internal realization of IPSEN is sketched. Section 5 gives an overview of the current state of implementation of the two prototypes and describes future research directions within IPSEN.

2 The User Interface

Fig. 2.1 gives a **snapshot of program development with IPSEN**. The user has just developed a Modula-2 module to compute the minimum and maximum of a cardinal array by using the syntax-directed Modula-2 editor of IPSEN. The module is currently executed by using the Modula-2 interpreter of IPSEN. During execution the user had stopped the execution of module MiniMax within the body of the FOR-statement and he had requested to display the current values of the used variables by using the IPSEN runtime analyser. By continuing the execution the IPSEN interpreter executes the IF-statement again, recognizes a runtime error and gives an appropriate error message (cf. Fig. 2.1 (a)). In Fig. 2.1 (b) the user starts to correct the array definition and, therefore, types in the according command.

Each of the tools provided by IPSEN corresponds to one or more special **views** which are represented by corresponding windows on the screen. Such views are cutouts of external representations of one internal high-level representation of a software document (cf. Section 4). They are generated by the corresponding tools, e.g. an editor view, an execution view, and a runtime

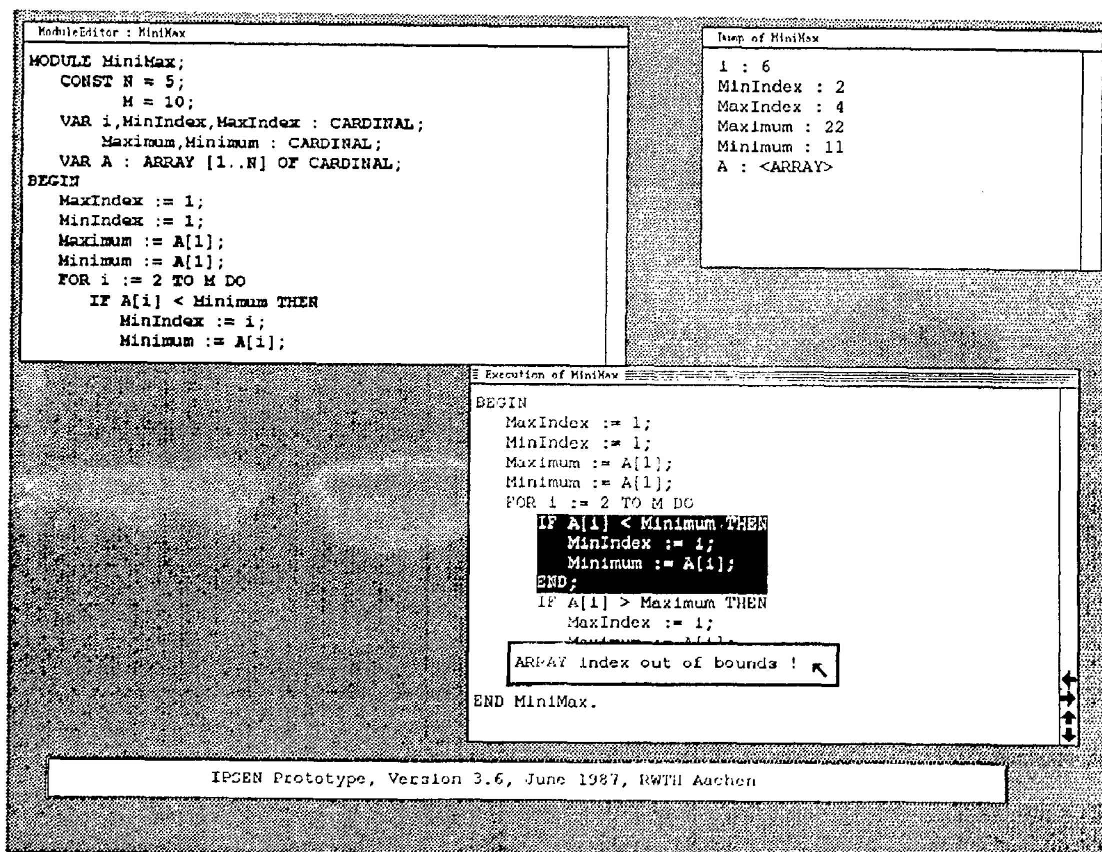

Fig. 2.1 (a)

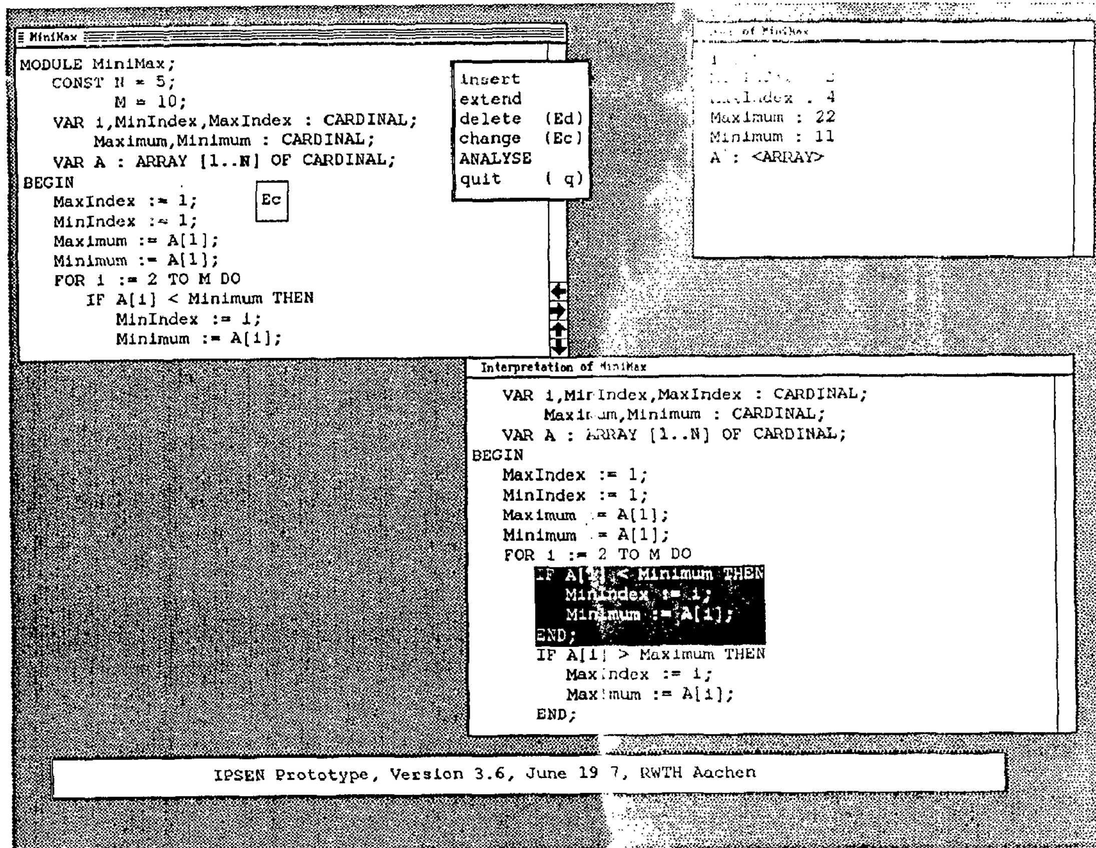

Fig. 2.1 (b)

analysis view of module MiniMax. A further editor view could be a graphical representation, e.g. a Nassi–Schneiderman diagram, a further runtime analysis view could be the more detailed display of a complex structured variable, e.g. variable A in Fig. 2.1 (a).

A view, according to its corresponding tool, either allows the IPSEN user to change the underlying internal data structure or just enables browsing through its external representation without any possibility of changing it. Therefore, we call the first kind of view an **active view** (e.g. an editor view), the second kind of view a **passive view** (e.g. the execution view). As usual in nowadays SDEs a structured cursor (instead of a traditional character–oriented one) is used to highlight the currently interesting syntactic structure in each view. That structure is called the **current increment**. To distinguish between an active and a passive view, the current increment within an active view is marked by boldface letters, whereas it is marked by an inverse representation within a passive view (cf. Fig. 2.1).

The kind of view also determines who selects a **new current increment**. In principal, it can either be selected by a tool or by a user. Within an active view, a user usually selects a new current increment. However, in some cases, the corresponding tool has to do that automatically. For example, when deleting the current increment within the editor view, the editor selects the following program structure to be the new current increment. (If that structure does not exist, it takes the surrounding structure.) Within a passive view, the tool always selects the new current increment. For example, the interpreter selects the new current increment to be the next statement which is to be executed. Nevertheless, the user can select an increment within a passive view. That selection, however, does not effect the passive view but determines that increment to be the new current increment within the corresponding active view, because, in IPSEN, a passive view is always dependent on an active view, its so–called **main view**. One main view is, for example, an editor view, one of its dependent views is the execution view. Let us explain the reason for that by the following example. Execution stops within the execution view because of a runtime error. Now, the user wants to change his module at that location where the error occurred. However, the editor shows only the first part of the module, where the execution was started. By selecting the new current increment of the editor view within the dependent execution view, the cutout shown in the editor view is automatically scrolled and shows the same cutout (or a similar one depending on the size of the windows) as in the execution view and still keeps the strict correspondence between a view and a tool.

To avoid confusion between different current increments, there is always only one current increment at the time on the screen. That means the finally selected increment is always regarded to be *the* current increment. The corresponding view is called the **current view**. That view is always represented in a window which is not overlapped by any other view.

Selection of a new current increment is carried out by using a mouse-like device. In contrast to other projects (e.g. /TR 81/, /Ma 87/), IPSEN only supports cursor movements controlled by the mouse. So, we avoid any difficulties a user usually faces when controlling syntax-oriented cursor movements by the cursor function keys of the keyboard.

After having described the representation of a module by different views, we, now, describe how the user can manipulate such a module, i.e. how commands are put in. IPSEN provides a command-driven user interface. In any situation it provides only those commands to the user the execution of which is possible and useful in that situation. The list of so-called **valid commands** is always determined by the current increment. For example, if the current increment is a statement within the editor view, the input of a declaration would be an invalid command (, given that the syntax of the supported programming language is defined accordingly.)

In some cases, the former man-machine dialogue additionaly influences the list of valid commands. For example, the command "continue" (which means "continue execution after having stopped") is only displayed when the interpreter has been invoked before by a corresponding "start-command". As this "dialogue history" is also taken into consideration by IPSEN when fixing the list of valid commands, we can say that our valid commands are highly context dependent (cf. also /MV 85/).

The list of valid commands is displayed within a **menu window** on the screen. Of course, the contents of that window is always updated, when a new current increment is selected. However, in some cases even if only considering context dependent commands, the resulting list of valid commands can become quite complex. In order not to confuse the user by that, i.e. to keep the menus still easy to grasp, our commands are structured within so-called **command groups**. These command groups again correspond to the defined tools. One command group includes all currently valid commands of one tool. A command group name is always displayed by using capital letters, whereas a single command name is written in small letters (cf. Fig. 2.1 (b)). A selection of a command group causes the expansion of that group and the formerly expanded group is shrinked. The initially expanded command group (when the menu is displayed for the first time after selection of a new current increment) is always that one which corresponds to the current view, i.e. to the according tool.

As command input by menu selection becomes quite awkward for the more experienced user, IPSEN also provides the possibility to type in commands via the keyboard. In addition to the name given in the menus, any command has a unique **short command name** which is used when typing in the command. When the user starts typing in a string instead of selecting a command within the menu window by the mouse, a so-called input window is automatically displayed within which the typed characters are echoed and correction of them is possible. That window is always displayed context dependent, i.e. close to the current increment. This makes it easy for the user to identify such small windows on the screen, and keeps a structured screen-layout (cf. Fig. 2.1 (b)).

Furthermore, that short command name is also displayed within the menus as a bracketed string behind the command name (cf. Fig. 2.1 (b)). This supports an unexperienced user in learning the short command name when using the menu selection for command input. After a certain time of experience with IPSEN he is able to remember the short names and can renounce a menu selection.

The idea of "learning the system by using it" is also applied when choosing **moment and duration of menu displays**. Menus are only displayed as long as the user does not use short command names. After having typed in the first correct short command name, the menu window automatically disappears. So, it does not uselessly spend space on the screen and worse than that, it possibly partially overlaps other windows which contain much more important information. The menu window, however, is immediately automatically displayed again, when the user does a typing error or types in an invalid command. Additionally, he can also request to see the menu again by typing in a "?".

As one could see from the so far given description, the IPSEN user interface provides for all the benefits coming from a command-driven user interface. However, in some cases it is more convenient for the user not being forced to type in or select a command explicitely. Two different types of **implicit command activation** are provided by IPSEN.

Firstly, the syntax-directed editor allows textual input of arbitrary large program fragments which is done again using an input window. The IPSEN editor is, therefore, called a hybrid editor. A more detailed explanation of that editor will follow in the next section.

The second type of implicit command activation is given when the list of valid commands only contains just one command. When the user selects a new current increment and IPSEN recognizes that the list contains only one command, that command is automatically executed. For example, selecting the placeholder for the right side of an assignment statement (an expression) as the new current increment within the editor view would result in the automatic activation of a change command, i.e. an input window is opened to enable the user to type in an expression. (Expressions are always typed in as strings (cf. section 3)).

Summarizing the **structured screen layout**, IPSEN divides the whole screen into windows which all have a certain logical type. Those window types are: (1) graphic which means that the window corresponds to a view and shows a graphical representation of an internal representation, (2) text which means that analogously a textual representation is displayed, (3) menu which means that a list of currently valid commands is displayed and the user is able to select one of them by a mouse-like device, (4) message which means that an error or system message is displayed and the user has to acknowledge that message by a mouse-click, and (5) input which means that input via the keyboard is echoed in such a window, and a simple text editor is provided which allows the user to correct his typed input. The size and location of all those windows on the screen can be changed by the user.

Our structured screen layout, especially the correspondence between tools and views is similar to that proposed in /Re 85/. In older approaches the division of the screen is just static (e.g. /TR 81/, /DL 84/), i.e. dynamic opening, closing and moving of windows on the screen does not exist. However, our approach also differs from newer projects in some details. Firstly, the strict correspondence between tools and views and the smooth transition between the use of different tools (which will be pointed out more carefully in the next section) is not always existing in other projects. This sometimes results in a very complex screen layout, i.e. the screen is completely overloaded and unstructured by a lot of windows, whereas IPSEN uses as few windows as possible. Secondly, the possibility of menu as well as keyboard command input and the smooth transition between them is unique to IPSEN.

3 The Tool Set

As mentioned before, the IPSEN tool set for programming-in-the-small consists of tools to develop and test a Modula-2 module. These tools are an editor, a static analysis tool, an execution tool, and a runtime analysis tool. Similar tools can be found in the tool set of other software development environments (cf. /He 84/, /He 87/). It is the topic of this section to explain the IPSEN specific features of these tools. In particular, we want to show that the tool set was designed as a set of equivalent tools from the very beginning. This is in contrast to a lot of other projects, where the whole SDE has been build around a syntax-aided editor (e.g. /DK 84/, /Ha 82/). The resulting highly integrated mode of working within the IPSEN tool set will be demonstrated by some examples.

In IPSEN the editing of the source code is supported by a syntax-oriented editor for Modula-2 (cf. /EN 87/). The user selects the current increment in the editor view and types in a command to trigger an editing activity. Possible commands are insert commands to extend the source code at the current increment, or delete commands to delete the current increment. Here, the IPSEN editor differs from most other editors, as, in addition to the contextfree correctness, the contextsensitive correctness (i.e. static semantics) of the source code is checked and preserved as well. For instance, a variable identifier can only be inserted within the statement part, if it has already been declared. Analogously, declarations can only be deleted, if there are no used occurrences in a statement part any more, or such an occurrence matches another declaration in a surrounding block.

For reasons of user friendliness, syntax-oriented editing can not be applied on the level of expressions or even on the level of lexical units like identifiers. In those cases, the usual text-oriented editing is provided by all syntax-oriented editors. In IPSEN, however, this usual text-oriented editing style can be chosen by a user on all syntactical levels, ranging from a single identifier up to the whole module. This switch to the text-oriented editing mode is done by invoking the change command (cf. Fig. 2.1 (b)). This feature of the IPSEN syntax-oriented editor first of all eases the modification of larger increments and avoids a huge set of different change commands. For example, it is very easy to change a WHILE-statement into a FOR-statement by using this text-oriented editing style. Furthermore, the user can choose and alter his personally favoured style of editing at any time, i.e. he is not forced to a pure syntax-oriented editing style. Of course, the modified text has to be checked by a **parser** (/Sl 86/), whether it contains contextfree or contextsensitive errors, after the modification has been

finished. Because of this mixture of syntax- and text-oriented editing styles, the syntax-oriented editor in IPSEN is called a **hybrid editor**.

Furthermore, the IPSEN editor has some enhanced features to provide support for testing a module or even a single procedure. For example, breakpoints, assertions or counters for statistical investigations (e.g. loop counters) can be inserted or deleted in the source code like usual Modula-2 increments. This means that all testing preparation activities are handled like usual text editing activities, which yields a uniform user interface for those two kinds of activities.

The use of the syntax-oriented editor guarantees the contextfree as well as contextsensitive correctness of the currently handled Modula-2 module. Besides the contextsensitive rules of Modula-2, additional contextsensitive rules should be observed by a certain Modula-2 module to ensure that it is a software product of high quality. Such contextsensitive rules concern,

- the consistency between the declaration and statement part, i.e. that each declared identifier is really used within the statement part
- minimality, i.e. that there are no statements which can never be reached during execution of the module
- runtime security, i.e. that each variable is initialized before its first use.

In contrast to the contextsensitive rules of the programming language, these rules are not checked automatically during editing. It is up to the user to decide when these rules should be checked and whether such a rule should be observed within the current module. All checks can be invoked by corresponding commands which enable the static analysis of a module.

The command set of the tool **static analysis** includes

- a test whether a declared variable identifier is used within the statement part
- an analysis which indicates all applied occurrences of a declared variable identifier
- an analysis which indicates all global variables which may be used when a procedure is invoked (side effects of a procedure call)
- an analysis which investigates the data flow within a module and indicates all used occurrences after the setting occurrence of a variable identifier (set / use chains).

The above mentioned tools syntax-oriented editor and static analysis support the incremental input of a Modula-2 source code, thereby guaranteeing that the source code is always syntactically correct and of high quality. Executing and testing activities within the software development process are supported by the tools execution and runtime analysis which will be explained now.

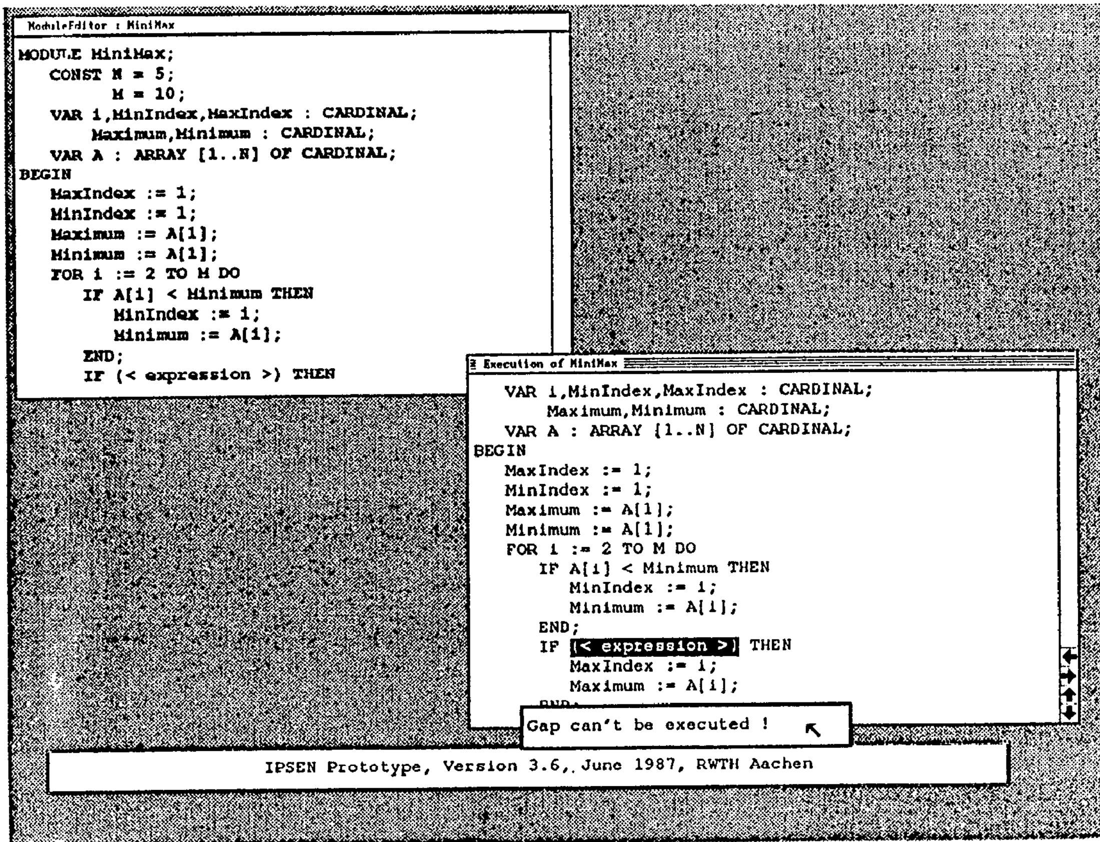

Fig. 3.1

Testing of the currently handled module may be started by the user at every time. In particular, the execution may be started even when there are still gaps in the source code (e.g. missing expression in an IF-statement (cf. Fig. 3.1)). Executable increments are a whole Modula-2 program module or a single procedure. At the beginning of an execution, IPSEN provides for commands to set the values of imported or global variables, or input parameters. After this initialization, the user can control the execution by setting different kinds of **interrupts**.

First of all, the execution is stopped automatically, if a **runtime error** is recognized (e.g. division by zero, array index out of bounds). In this case, the source code has to be modified by the user and the execution has to be started again. A second case of such an implicit interruption occurs, if a **gap** in the source code has to be executed (cf. Fig. 3.1). In this case, a message is displayed and the user can fill the gap by directly invoking a command of the editor. After that, the execution is continued automatically. The filling of a gap is the only case, where a continuation of the execution is possible and allowed, after the source code has been modified. In any other case, the execution is stopped and has to be started again from the beginning by the user. This restrictive procedure has been chosen in IPSEN in order not to support an experimental style of programming.

Similar to stepwise execution by a usual debugger, the IPSEN user can explicitely determine at which points the execution should be stopped. As all activities in IPSEN are increment-oriented, **determination of interruption points** is increment-oriented, too. According to the underlying contextfree syntax tree of a module, the user can determine by corresponding commands that increment where execution should stop the next time. Possibilities for that increment are the first statement in the body of the current increment (son), the next statement on the same level (brother), or the next statement on the next higher level (uncle). Those different interrupt points enable a user to slow down the execution speed in critical regions of the source code and to speed it up in already checked parts of the source code.

Finally, the execution is stopped when a **breakpoint** is reached, which has been inserted into the source code by a test preparation command before the execution has been started.

Those different possibilities to stop the execution ... able to provide for additional commands to obs... control and data flow of the executed source co... appropriate support for these observations is offered by the **runtime analysis** tool. The commands of that tool can be invoked by the user whenever the execution is inter-

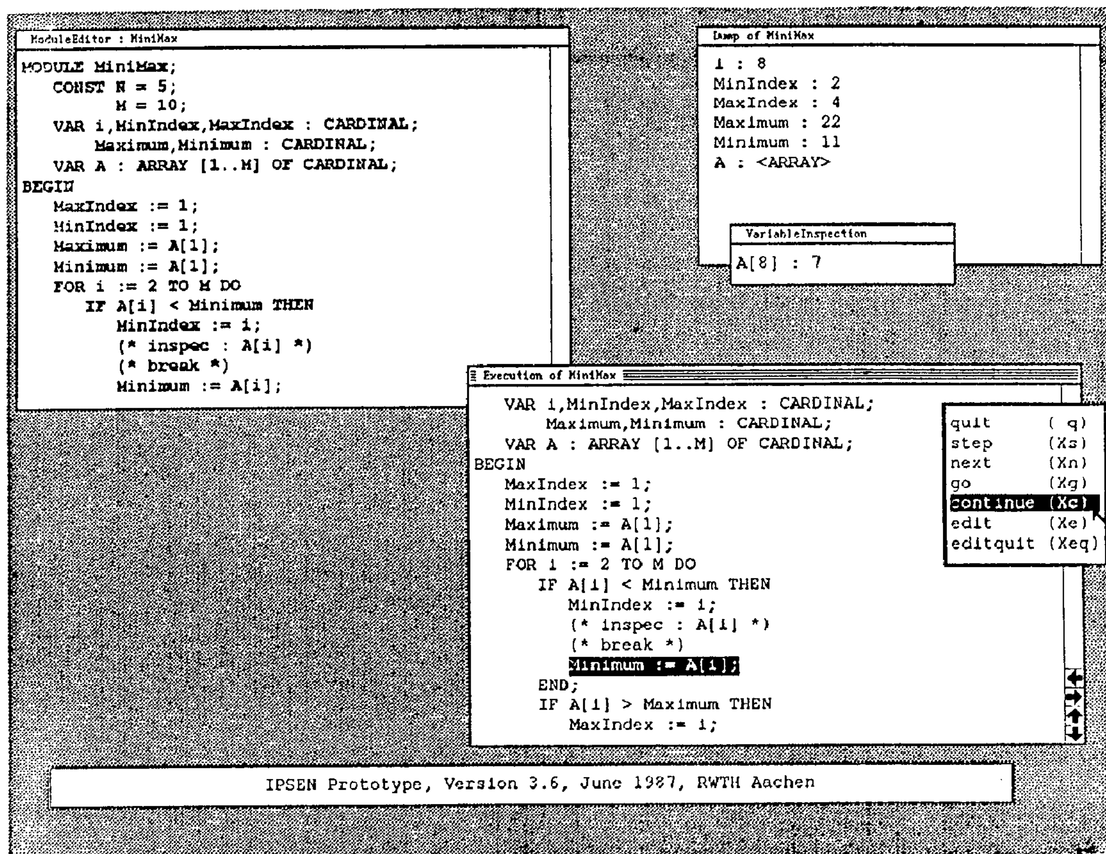

Fig. 3.2

rupted. Two important commands of this tool enable the output of a dump related to the source code, i.e. a list of *all* variable names together with their values, and the output of the value of one certain variable selected by the user. Figure 3.2 gives a snapshot of an IPSEN session illustrating the integrated manner of working with these different tools: The editor view shows a cutout of the currently handled module. The execution has been stopped at a breakpoint. The statement after the breakpoint is marked as the current increment in the execution view. The user has invoked the dump command at an earlier interrupt point. The corresponding runtime analysis view which contains variable names together with their current values is still on the screen. Additionally, the values of the variable A[i] are shown in a further runtime analysis view.

Besides those runtime analysis activities, test preparation commands or static analysis commands can be invoked at any interrupt point. It is a usual procedure during the test phase to invoke the same analysis commands at a certain interrupt point whenever the execution reaches this point. Therefore, these commands are saved on demand by the user and are implicitely invoked the next time, the execution reaches this point. Figure 3.2 gives an example, where the command to inspect the variable A[i] has been inserted into the source code. The user now has the possibility to build up interactively an appropriate **test environment**. All such implicitely invoked commands are represented as Modula-2 extensions in the source code and can be edited by the syntax-oriented editor. At any point in time, the whole test environment can be switched on or off.

Summarizing our example, we illustrated that the tools syntax-oriented editor, static analysis, execution and runtime analysis form a highly integrated tool set. This means that all tools have the same common user interface. This concerns the structured screen layout by using views as well as the kind of invocation of commands (cf. section 2). Furthermore, there are no tool-dependent modes. This means that the user can activate any command of any tool with the only restriction that it must be a valid command for the current increment. Furthermore, the execution of a command may be stopped and a command of another tool may be invoked and executed before the execution of the first command is finished, e.g. it is possible to invoke analysis commands whenever the execution of a module is interrupted. This strategy is not applied, when the dialogue becomes too confusing or even encourages an experimental style of programming, e.g. it is not possible to invoke arbitrary editing commands when the execution of a module is interrupted.

4 Sketch of the realization

The tool set described in this paper is part of the IPSEN-prototype which has been implemented successfully on an IBM-AT and is just being ported on a net of SUN-workstations. **Integration** as the main guideline for the user interface design has also **heavily influenced the realization** of the prototype.

In particular, we use **one internal high-level data structure** containing all the knowledge for all the different tools (cf. /EN 86/). That includes the contextfree structure of a module as well as the contextsensitive structure, the information for the test environment (e.g. breakpoints), and the information for the hybrid interpreter (e.g. P-code fragments) which is the realization of the execution tool (cf. /ES 87/). Hence, there are several advantages of IPSEN compared to some other SDE-projects where different data structures are used (cf. /Re 84/). Firstly, using one uniform model for different kinds of knowledge representation makes it much easier to model all operations carried out on that representation. Secondly, problems in preserving consistency between different representations do not exist.

Contrary to most other SDE-projects (cf. /Ha 82/ or /DK 84/) we do not use attributed trees as the model for our internal data structures but **attributed graphs**. Only such a universal model enables to model the different information needed by the different tools very efficiently. Implementing the necessary graph operations is directly supported by a special data base system, called graph storage. It allows to store and retrieve arbitrary large graphs. The basic strategies applied to obtain very fast retrievals and updates are dynamic hashing and semantic paging (cf. /LS 87/).

The IPSEN-prototype has been designed in a way that guarantees a **maximum of adaptability and portability** (cf. /ES 85/). Adaptability means that new requirements can be satisfied without enforcing a lot of changes. It is not difficult to add new tools for PIS or even tools for another task area. This has been shown by applying the same approach to build systems for PIL, DOC and ORG (cf. /Be 87/, /Ja 87/, /Le 88/). Portability means that the system can easily be adapted to different hardware facilities and operating systems by only adapting the machine dependent modules of the system components *graph storage* and *window system*. Portability has been proved by porting these two components from the IBM-AT to a net of SUN-workstations.

5 Concluding Remarks

We described the main characteristics of an integrated software development environment. On one hand, we developed an integrated user interface for the whole environment. On the other hand, we designed a complete set of tools working in a highly integrated way. The underlying architecture enables an easy adaption to new task areas and an easy portation to new hardware facilities.

The current status of the implementation is as follows: Tools for the task areas PIL, PIS, ORG and DOC have been implemented on the IBM-AT. The prototype consists of about 70.000 loc Modula-2 source code and is running under MS/DOS. Response time for editor operations does not exceed 2 seconds which fulfills all requirements for a first prototype. The portation of the tools described in this paper to a net of SUN-workstations running UNIX has been finished successfully.

After porting and integrating the tools for the task areas PIL, PIS, DOC and ORG our **future work** will be to extend the prototype to get a multi-user environment. Furthermore new working areas (e.g. requirements engineering) will be investigated.

Acknowledgements

The authors are indepted to all members of the IPSEN team working hard for the progress of the project.

References

/Be 87/ E. Berens : Support of Technical Documentation in Software Development Environments (in German), Master's Thesis, University of Osnabrück

/DK 84/ V. Donzeau-Gouge / G. Kahn / B. Lang / B. Melese: Document Structure and Modularity in MENTOR, in /He 84/, 141-148

/En 86/ G. Engels: Graphs as Central Data Structures in Software Development Environments (in German), Ph.D. Thesis, University of Osnabrück

/EN 87/ G.Engels / M.Nagl / W. Schäfer: On the Structure of Structured Editors for Different Applications, in /He 87/, 190-198

/ES 85/ G. Engels / W. Schäfer: The Design of an Adaptive and Portable Programming Support Environment, Proc. of the 8th International Computing Symposium 1985, Florence, Italy, 297-308

/ES 87/ G. Engels / A. Schürr : A Hybrid Interpreter in a Software Development Environment, in Proc. of the 1st European Software Engineering Conf., Strasbourg, Sept. 1987, to app. in LNCS, Springer

/GM 84/ D.B. Garlan / P.L. Miller : GNOME : An Introductory Programming Environment Based on a Family of Structure Editors, in /He 84/, 65-72

/Ha 82/ N. Habermann et al.: The Second Compendium of Gandalf Documentation, Tech. Report, Carnegie-Mellon University

/He 84/ P. Henderson(Ed.): Proc. of the ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 19,5

/He 87/ P. Henderson(Ed.): Proc. of the Second ACM SIGSOFT / SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices 22,1

/Ja 87/ Th. Janning: Access And Responsibility Control in a Software Development Environment (in German), Master's Thesis, University of Osnabrück

/Le 88/ C. Lewerentz: Incremental Programming-in-the-Large, Ph.D. Thesis (in German), RWTH Aachen, forthcoming

/LN 85/ C. Lewerentz / M. Nagl: Incremental Programming in the Large: Syntax-aided Specification Editing, Integration, and Maintenance, in Proc. of the 18th Intern. Conf. on System Sciences, Hawaii

/LS 87/ C. Lewerentz / A. Schürr: The Software Documents Database GRAS (in German), in: GI-Softwaretechnik-Trends 7-2

/MP 87/ N. Madhavji / L. Pinsonneault / K. Toubache: A New Approach to Cursor Movements in Programming Environments, Technical Report SE-87.1, SOCS McGill University, Montreal

/MV 85/ N. Madhavji / D. Vouliouris / N. Leontsarakos: The Importance of Context in an Integrated Programming Environment, in Proc. 18th Int. Conf. on System Sciences, Hawaii

/Re 84/ S.P. Reiss: Program Development with PECAN Program Development Systems, in /He 84/, 30-41

/Re 85/ S.P. Reiss: PECAN: Program Development That Supports Multiple Views, in IEEE Trans. on SE, Vol.11, No.3, 276-285

/RT 84/ T. Reps / T. Teitelbaum : The Synthesizer Generator, in /He 84/, 42-48

/Sc 86/ W. Schäfer: An Integrated Software Development Environment (in German), Ph.D. Thesis, University of Osnabrück

/Sl 86/ U. Schleef: An Incremental Parser as Part of a Syntax-Directed Editor (in German), Master's Thesis, University of Osnabrück