2.1 Tight Integration on One Document: The Programming Environment

G. Engels, M. Nolli, W. Schäfer, B. Westfahl

We describe in this chapter the outside modularity and user interface of IPS!N tools, putting a strong emphasis on various aspects of integration. As starting point, we have selected the PS environment, to be explained in this section as (1) the language supported by the environment needs no explanation, (2) the functionality is easy to understand, and (3) it nicely shows the IPS!N philosophy of high-level and immediate support of technical work (here: programming) without restricting the corresponding process. The PS environment was already implemented in 1985 as the first prototype within the IPS!N project IPSEl 1.0 and was reimplemented in IPS!N 2.5.

2.1.1 Introduction

The PS environment demonstrates integration of different tools operating on the same logical document. Each such document corresponds to a body of a module whose interface is specified in a software architecture. Specification of module interfaces belongs to PSI and is therefore not discussed in this section.

The PS environment supports Modula-2 as implementation language. We have to stress that it is an incomplete prototype rather than a full-fledged environment of practical use.

Only a subset of Modula-2 is supported, but this subset includes most controlled structures, but is rather limited with respect to data structures. To implement a complete prototype, we would have to apply well-known techniques of compiler construction for all languages constructs which was not regarded to be of great scientific value. Rather, we focused on a subset of these constructs to demonstrate the feasibility and main characteristics of our incremental approach to tight tool integration.

In the following, we present a short tour through the PS environment by showing and explaining snapshots from a demo session. The demo illustrates incremental integration of editing, analysis, instrumentation, and execution. A lot of lessons have been learned from IPS!N 2.5. The study of this prototype provides the reader with valuable insights into functionality and user interface of IPS!N environments. All remaining sections of this chapter refer to more recent prototypes, namely IPS!N 3.0 and IPS!N 4.0. Many editing, analysis, instrumentation, and execution commands may be interleaved arbitrarily. The user is not forced through lengthy edit-compile-debug cycles as in compiler-oriented programming systems. Tools provide for immediate response on syntactical and semantical level in order to inform the user about impacts of modifications as soon as possible. Efficient operation

is achieved through incremental realization (e.g. only the affected portion of a module is analyzed after an edit command).

Fig. 2.1. User interface of IPS!N 2.5

In the screen dump of Fig. 2.1 illustrates the user interface of IPS!N 2.5. When this prototype was implemented or reimplemented in 1985/87, no user interface tool kit was available which satisfied our requirements. Therefore, we developed a suitable window system on our own. Of course, the structure is completely different today. Therefore, all prototypes described in the remainder of this chapter provide for an OOP/MOTIF-like user interface whose implementation is based on X Windows. However, many characteristics of IPS!N 2.5 apply to more recent prototypes, as well.

Fig. 2.1 shows different kinds of windows. The text window shows a cutout of a small Modula-2 program which will be used throughout the sample session. The program calculates the least common multiple (1cm) of two positive numbers with the help of their greatest common divisor (gcd). The procedure gcd is an inner procedure of the procedure lcm.

Only a part of both is displayed in Fig. 2.1. The text window is decorated with a title (top left corner) and a scrollbar for scrolling the text to the left/right/top/bottom (side bar). The small window above the text window contains a menu of commands. Each command refers to the current increment selected in the text window. The current increment is emphasized in bold face (the whole program in Fig. 2.1), and it is selected by a mouse click. The contents of the menu window depends on the current increment, i.e. after each selection the set of legal commands is calculated according to the type of the current increment. A command may be activated either by mouse click in the menu window, or by typing in a shortcut which is enclosed in parentheses behind the command name (e.g. for execute).

Our sample session first demonstrates the editor which supports both syntax-directed editing and free text input. The advantage of such a hybrid approach consists in its flexibility. Syntax-directed editing may be applied by novice users who are not familiar with the underlying language, or it may be used to save typing in case of a verbose concrete syntax. On the other hand, free text input may be preferred by experienced users, or for performing bigger structural changes for which there are no syntax-directed commands.

Our first program modification consists of creating an assignment statement by syntax-directed editing (Fig. 2.2). We select the assignment operator (=), which is the first statement of the main program, as our increment and invoke the command context (screen).

Note that the syntax-directed editor only offers commands which do not violate the context-free syntax, e.g. it would not offer a command to create a variable declaration at this point of the program. In response to the command activate, placeholders for the left-hand and right-hand side as well as concrete syntax symbols ( and ) are generated, respectively.
Left-hand and right-hand side are treated as atomic increments with respect to syntax-directed editing. This means that they are manipulated by free text input in order to avoid syntax-directed editing of variables and expressions. Note, that in IPSEN 2.5 syntax-directed editing ends at this level. In more recent prototypes, any portion of the concrete syntax may be edited either in syntax-directed or in free input mode, i.e. syntax-directed editing ends at the level of lexical units of the underlying language. To use free input on the level of expressions is, however, more friendly to users.

In fig. 2.2, the user has just issued the setextend 1 gmn command and is now editing the left-hand side in the free input mode. Free input is performed in an input window which initially contains ASCII text of the current increment. During free input, no checks are made with respect to context-free or context-sensitive syntax. The user terminates free input by hitting a function key. Subsequently, context-free and context-sensitive analyses take place. In our example, the user has typed an identifier, which has not been declared before. The user is informed immediately about this error, and free input mode is entered once again. In IPSEN 2.5, free input is accepted only when no context-free or context-sensitive errors are detected; otherwise, all modifications are discarded. In more recent prototypes, we have removed the second restriction, i.e. context-sensitive errors are tolerated temporarily and may be removed later on. Our experience with IPSEN 2.5 has given evidence that exclusion of context-sensitive errors severely restricts the user in his natural way of program construction.

Fig. 2.3 illustrates free input on a complex increment, namely the statement list of the procedure 1 cm. To speed up computation of 1 cm, the original statement list is embedded into an if-statement whose then-part delivers the result in the special case a gmn = 1. In principle, such a program transformation could be supported directly by a syntax-directed command (embed current statement list into the else-part of an if-statement). However, this approach would result in a large number of special-purpose commands. The effort of implementing these commands does not seem to be justified as their effects can be achieved conveniently by means of free input and immediate syntax analysis. Note, that the user does not need to care about layout issues during free input. An indented layout is generated by an unparser which transforms the internal document representation into a human-readable text representation (fig. 2.4). In IPSEN 2.5, the user cannot modify the layout of this text representation; more recent prototypes include a layout editor for polishing the output produced by the unparser.
2.1.3 Execution and Analysis Integrated with Editing

IPSEN 2.5 provides also for seamless integration of editing, instrumentation, and execution. The execution tool is a two-level interpreter which evaluates control information to determine the next statement and interprets P-code for simple statements such as assignments and procedure calls. P-code is generated incrementally as required; therefore, execution of a loop is slow for the first iteration and much faster for subsequent iterations. The user may select among different modes of execution: step steps into a compound statement, next executes the next statement, and continue executes until the end of the program. At any time, execution may be interrupted manually. Furthermore, execution is stopped when a breakpoint is reached, an assertion is violated, or a runtime error is detected.

Fig. 2.6 shows an example for the last case. The lower left window is used to view the current execution increment (which is emphasized by an inverse representation), and to issue execution commands. The message window, displayed relative to the current execution increment, says that a variable has been read before it has been written. The window on the right-hand side displays a dump of the runtime stack. According to the dump view, the local variable %x of the procedure got has an undefined value. Inadvertently, this variable is accessed in the condition of the while statement.

Fig. 2.7 illustrates interleaving of editing and execution. In the upper left window, the program error is removed by editing the condition of the while statement (%x = replaced by %y + %z). Afterwards, execution can be resumed where it has stopped because of the runtime error. Of course, this does not work in all situations. For example, execution cannot resume after the current increment has been deleted. Furthermore, execution is aborted if the formal parameters of a currently executed procedure are modified. On the other hand, interleaving of execution and editing goes far beyond the functionality of conventional edit-compile-debug tools.

To conclude this subsection, let us demonstrate the functionality of the static analyses tool. This tool handles the analysis of the context-sensitive syntax. For example, it may be used to determine all global variables which are used in a certain procedure. In fig. 2.8, last analysis has been performed for the procedure get (current increment in the upper left window). The outcome is displayed in the lower right window. result is a global variable used in procedure 2 cm.

In fig. 2.9, another analysis has been applied to the same variable result of the whole program (see fig. 2.8), which delivers all applied occurrences. Some of them are contained in the procedure get. The programmer has used result inadvertently in 2 cm, where he should have used the local variable getvar. Therefore, all occurrences of result in 2 cm have to be replaced by occurrences of that local variable getvar. For that, editing and
2.1.4 Summary and Comparison

IPSEn 2.5 offers a window-based user interface which distinguishes between windows of different types for editing/instrumentation, analysis, and execution, in the latter case with an additional window to display the run-time stack. Documents in PsE are represented in text windows. Since the layout is generated by an unparsable, the user is relieved from formatting. Each command issued by the user refers to the current increment which is selected by mouse click. Commands are displayed in a menu window where they may be selected by mouse click; alternatively, the user may type in a shortcut. The contents of the menu depend on the type of current increment. Messages, e.g. error messages, are displayed in message windows. Finally, input windows are used for free text input.

The PsE part of IPSEn 2.5 offers tightly integrated tools which cooperate on one document. IPSEn 2.5 supports programming in Modula-2. Tools are seamlessly integrated, i.e. commands in different windows may be interleaved freely. All tools operate on a shared internal representation which represents context-free syntax, context-sensitive syntax, and dynamic semantics. The editor supports syntax-directed program construction as well as free text input. It checks and enforces both context-free and context-sensitive correctness. Programs may be prepared for execution by an instrumentation tool which operates in a similar way to the editor. Program execution is supported by an interpreter which provides different modes of execution (e.g. next, continue). Finally, a static analysis tool determines, e.g. global variables used in a certain procedure or all applied occurrences of a certain variable.

The IPSEn programming environment may be compared to other structure-oriented environments which were developed in the 80's. Gandalf/2, HIN 86, Mentor/2, D4K 84, PsG 2, BS 86; and the Cornell Program Synthesizer. RT 84 are well-known examples of such environments.

A syntax-aided editor plays a central role in a structure-oriented programming environment. The spectrum ranges from purely syntax-directed editors (e.g. Gandalf), which enforce syntax-directed editing down to the level of lexical units, to text editors which are integrated with an incremental parser running in the background (e.g. Magpie/4, DMS 84). IPSEn provides a hybrid editor which combines syntax-directed editing and free input. In this way, both novel and experienced users are adequately supported. Furthermore, the IPSEn editor performs incremental context-sensitive analysis, which is also done in, e.g. in PsG, the Cornell Program Synthesizer, and the Pan system. AEG 90.

In addition, IPSEn supports static analysis, instrumentation, and execution in a seamlessly integrated way. Not all structure-oriented environments cover these additional areas. For example, Pan and the Cornell Program Synthesizer are restricted to syntax-aided editing. Those environments which do provide support for the tasks mentioned above achieve seamless integration only rarely. For example, PsG allows for execution of incomplete programs. However, only program extensions (expansions of placeholders) are allowed during program execution. In contrast, IPSEn also allows program modifications at any time as long as they do not cause inconsistencies with respect to execution.