Int. J. Man–Machine Studies (1992) **37**, 399–418

Interacting with structure-oriented editors

STEN MINÖR
Department of Computer Science, Lund University, Sweden

Why have structure-oriented editors failed to attract a wider audience? Despite their obviously good qualities, they have almost exclusively been used for education and for experimental purposes in universities and research labs. In this paper a number of common objections raised against structure-oriented editors are quoted and commented upon. Many objections concern the interaction of such editors. Therefore the aspect of interaction in structure-oriented editors is analysed in more detail. We pin down the differences between interacting with text and structure-oriented editors, thus obtaining a deeper understanding of how structure-oriented editors can be improved to suit both naive and expert users. An analysis based on Norman's model for user activities is presented both for text editing and structure-oriented editing of programming languages. The analysis illustrates the trade-offs between structure-oriented editing and text editing of programs. It is also used to suggest some improvements to structure-oriented editor interaction in order to minimize the mental and physical effort required. The interaction problems have earlier been dealt with in hybrid editors, which combine structure-oriented editing and text editing in one system. This approach is also commented upon and discussed. Conceptual models are presented and compared for text editors, structure-oriented editors and hybrid editors. An interaction model for structure-oriented editors based on direct manipulation is suggested. The model is examined in terms of semantic distance, articulatory distance, and engagement as suggested by Hutchins *et al.* It is also related to the analysis of user activities and the discussion of conceptual models. The direct manipulation model aims at obtaining a simple but powerful interaction model for "pure" structure-oriented editors that may be appreciated by different user categories. Finally, some objections against structure-oriented editors not concerning interaction issues are commented upon, and some directions for future research are outlined.

1. Introduction

Traditionally, text-oriented editing has been used as a generic tool for manipulating programs as well as documents and reports etc. Editing programs in terms of text has the advantage of making use of the user's previous knowledge of writing and typewriting, but does not give any language-specific support. Text editors, even today, are the predominant tools for program manipulation. One reason might be that other types of editors have failed to convince users of their advantages. The questions of what is most "natural" for the user and whether a program should be edited as texts or not are still controversial ones.

TEXT-ORIENTED EDITORS

Representing programs as text allows the use of several generic tools for processing textual representations. Furthermore, program text can easily be exchanged between different systems. The generality is, however, due to the low level of the representation. It also results in tools that in general are not supporting specific tasks such as program editing. One of the best known text-based editors is Emacs

0020-7373/92/100399 + 20$08.00/0

(Stallman, 1981). One important feature of Emacs is its extensibility allowing customization of the editor and new functionality to be added. This can be used to support program editing with automatic indentation and parenthesis-balancing capabilities. The program representation and manipulation in these editors are, however, still textual, and the programming support supplies single features rather than making the impression of a thoroughly designed, specialized tool for program editing.

HIERARCHICAL TEXT-ORIENTED EDITORS

A variant of the text editor is the hierarchical text editor, or outline editor. A hierarchical text editor is basically a text editor with a hierarchical structure superimposed on the text. The nodes of the hierarchical structure may be of arbitrary size. They typically contain chapters of a report or procedures of a program. The hierarchies may be used for presentation with a varying degree of detail, e.g. presentation of the procedure names in a program only. Except for the hierarchies, the same editing functionality as a conventional text editor is supported. Ed3 (Strömfors, 1986) is a hierarchical text editor with additional functions supporting program construction. Welsh and Toleman (1992) have described structure-oriented editors which treat larger units, such as programs, modules, procedures and functions as hierarchical objects while smaller units such as statements, declarations and expressions, are presented and edited as text.

HYPERTEXT SYSTEMS

Hypertext systems (Conklin, 1987) may be viewed as a generalization of hierarchical editors which superimpose a graph structure instead of a tree structure on the edited texts. Hypertexts have not been utilized widely for program editing to date, even though their abilities for sophisticated navigation could be of great value for software development. Hypertext concepts in programming environments have been described by Sandvad (1989) and Nørmark (1990).

STRUCTURE-ORIENTED EDITORS

Structure-oriented editors operate on the syntactic level of the language underlying the edited structure. Programs are created and modified in terms of language constructs and are represented as syntax trees. Textual representations are only used for presentation of programs and for input purposes in some cases. Structure-oriented editors have knowledge of the underlying language and guarantee that the resulting program is syntactically correct. This is accomplished by using a restrictive approach as in program synthesizers or "pure" structure editors, which allow only syntactically correct programs to be constructed; or it may be achieved by supporting the user through checking the program's syntax and reporting errors. Furthermore, structure-oriented editors offer the user guidance on the available syntactic constructs at every step of program construction.

Several structure-oriented editors and environments have been reported in the literature, e.g. Mentor (Donzeau-Gouge *et al.*, 1980, 1984), The Cornell Program Synthesizer (Teitelbaum & Reps, 1981), The Synthesizer Generator (Reps & Teitelbaum, 1984), Poe (Fischer *et al.*, 1984), Gandalf (Notkin, 1985), Muir

(Winograd, 1987), Centaur (Borras *et al.*, 1988), SbyS (Minör, 1990), Mjølner/Orm (Magnusson, *et al.*, 1990), PSG (Bahlke & Snelting, 1992) and Pan (Van De Vanter, Ballance & Graham, 1992). Most of these structure-oriented editors have been developed for editing formal languages, such as program languages, specification languages, grammars etc. Other application areas are document editing (Walker, 1981), command language editing (Minör, 1987), data structure editing (Fraser, 1981) and graphical editing (Szwillus, 1987; Backlund *et al.*, 1990; Göttler, 1992). Structure-oriented editing has also been suggested as a general paradigm for a variety of computing tasks (Notkin, 1984). We will, however, focus on editing formal languages, and particularly programming languages, in this paper.

HYBRID EDITORS

Some structure-oriented editors include text-editing capabilities, as well as structure-editing functionality, resulting in hybrid text/structure editors. This was done to overcome the interaction problems in pure structure editors, which we will expand upon in the next chapter. A hybrid editor is basically a structure-oriented editor with a parser added to it. Program fragments can be created as text pieces which are parsed into substructures and pasted on to the main program structure. Non-structural modifications can be performed by cutting a tree fragment from the main program structure, unparsing it to textual form, have the user edit the text, reparsing it, and finally pasting the resulting structure on to the main program structure. Examples of hybrid editors are the Cornell Synthesizer Generator, Poe, PSG and Muir.

THE STRUCTURE OF THIS PAPER

The remainder of this paper will discuss the interaction of text editors, pure structure editors and hybrid editors in more detail. It is organized as follows: in the next Section, a number of frequently mentioned problems of structure-oriented editors are listed and commented upon. Sections 3, 4 and 5 contain a deeper analysis of the interaction in structure-oriented editors. In Section 3 an analysis of user activities based on the seven-stage model suggested by Norman is presented. Section 4 discusses mental models for structure-oriented editors, and Section 5 discusses use models or metaphors.

In Section 6 a new approach for interacting with structure-oriented editors based on direct manipulation is presented. In Section 7 the proposed approach is discussed and related to the objections in Section 2 and to the analysis in Section 3. Finally, in Section 8 a few issues for further research are suggested, which are important to render structure-oriented editors more usable.

2. Objections against structure-oriented editors

Structure-oriented editors based on program synthesis are often criticized as being tools useful to novices only. That program synthesis is favorable for beginners in learning a language is less controversial. Program synthesizers relieve beginners from dealing with the details of concrete syntax and enforce syntactically correct programs, allowing more effort to be spent on problem solving and programming concepts. Furthermore, they guide the novice by presenting the valid syntactic categories at each step of program construction.

Among expert users, the opinions about the usefulness of program synthesizers are more diverse. In the academic community, environments built on structure-oriented editors have been an active research area. The editors have been used for different experimental purposes within these projects, but have seldom been used for other purposes. In "real" systems for industrial use, the impact of structure-oriented editors has been very moderate. Despite the claimed qualities of program synthesis its ideas have not been widely accepted. This also holds true for areas other than program construction, where structure synthesis is applicable as a technique. Some frequently mentioned problems with structure-oriented editing for program construction are listed below.

(i) A sequence of actions is required at each step of the incremental program construction. This is particularly awkward when editing "expressions" in programming languages.
(ii) Programs have to be entered in a top-down fashion, which constrains the user.
(iii) Some transformations may be done in a more efficient way using a text editor.
(iv) Structure-oriented editors enforce syntactic consistency and thus limit the user's freedom. Experienced users often intentionally introduce an intermediate inconsistent state when modifying a consistent program fragment to another.
(v) Expert users know the language and do not need the guidance provided by structure-oriented editors.
(vi) Structure-oriented editors use a program representation that is incompatible with conventional text-oriented environments.
(vii) The textual presentation of program structures supported by most systems is sometimes misleading.

We will try to comment on these objections one by one and suggest solutions that might make structure-oriented editors more tractable for expert users as well as novices.

OBJECTION (i), "EDITING EXPRESSIONS IS AWKWARD..."

This objection concerns efficiency in terms of actions required by the user to construct a structure. The interaction of traditional structure synthesis follows the following pattern: select a node—give an expand command—select an alternative from a menu with expansion alternatives. The last selection is often done in a nested menu with a number of levels and thus a number of selections. Syntactic categories that have a complicated structure or a comprehensive concrete syntax are often more efficient to construct using structure synthesis. The number of actions in this case is small compared with typing text. For simple structures such as arithmetic expressions in programming languages, the opposite holds true. Entering an addition in many languages requires only one action using a text editor—typing a "+". Using synthesis, several actions may be required, and many users feel uncomfortable constructing simple structures this way. A traditional text editor is more efficient for this, and despite the advantage of structure synthesis, many users would hesitate to use it simply because it is tedious to use for these cases.

OBJECTION (ii), "TOP-DOWN CONSTRUCTION CONSTRAINS THE USER..."

The fact that structures have to be constructed in a strict top-down fashion results from the restriction that only syntactically correct structures are allowed in the correct context. The context thus has to be created before a certain structure is created, and the user is trapped in a mode. One solution to this problem is to provide a clipboard facility that allows structures of arbitrary syntactic category to be constructed. The structures created will be synthesized in the usual (top-down) manner, but not until the structure is pasted on to another structure will the syntactic correctness of the paste operation be checked. The result is a combination of top-down and bottom-up techniques. Structure fragments are initially created top-down, but they can be put together in a bottom-up fashion. More elaborate support for bottom-up construction in structure-oriented editors is, however, an approach that may be interesting for further studies in order to obtain a less restrictive way to construct structures.

OBJECTION (iii), "TRANSFORMATIONS ARE MORE EFFICIENT IN TEXT EDITORS..."

Some modifications of structures that are structurally difficult can be done with simple textual modifications. An example is a procedure in Pascal that can be changed to a function just by modifying the word "procedure" to "function" (and adding the function type), leaving the rest of the structure untouched. Using a text editor, such modifications are probably more efficient in terms of number of keystrokes. Yet, there is a risk of introducing syntactic errors. Other modifications do, however, require substantial efforts in textual representations, e.g. modifying a "while" statement in Pascal to a "repeat" statement. In this case, structure-oriented editing would be preferable. In both these cases, support for syntactic transformations is desirable. An intuitive interaction model for transforming structures which make transformations more convenient, efficient and safe to use in both cases discussed above, is desirable.

OBJECTION (iv), "ENFORCED CONSISTENCY LIMITS THE USER'S FREEDOM..."

Many experienced programmers express a desire to "fiddle around" with the textual representation and tolerate inconsistent states while it is modified from one consistent form to another. They do not want to be scrutinized; they "know what they are doing". Experienced programmers form a user category with a long tradition of using character-oriented user interfaces and command languages. Some of the reluctance toward structure-oriented editing may be due to a conservative attitude toward non-traditional tools, but to some extent it may result from inferior interaction in existing structure-oriented editors. There is actually no reason for *wanting* a program to be in an inconsistent state at any time. What we really want is an editor that allows programs to be manipulated conveniently with maintained consistency. Whether this is a realistic goal, or if users actually cannot avoid allowing inconsistent states in their work constructing a consistent program, is, however, still an open question.

OBJECTION (v), "EXPERTS DO NOT NEED GUIDANCE..."

Basically, this problem is the same as the trade-off between command languages and menus, as examined by Norman (1983). Command languages offer a greater flexibility and are more efficient to use for experienced users who are familiar with the language, while menus offer assistance for novices and are often self-explanatory. Furthermore, menu interaction eliminates syntax errors. The traditional interaction model for structure synthesis uses menus to select appropriate syntactic categories when a node is selected for expansion. Experienced users find this awkward since they know the applicative alternatives and what syntactic construct they want at a particular place. The guidance from the menus is superfluous and gives rise to a number of unnecessary actions. The ideal editor would offer guidance to inexperienced users without forcing it upon experienced ones.

OBJECTION (vi), "STRUCTURE-ORIENTED EDITORS USE A DIFFERENT REPRESENTATION..."

Structure-oriented editors represent and store structures in a different way from traditional text-based tools. For the sake of compatibility translation between the formats are needed. Translation from textual form to structures requires a parser; translation from structures to text requires an unparser. These translation tools do not necessarily have to be integrated with the editor. They can be stand-alone tools in the same way as tools for translation between different file formats of operating systems. Furthermore, they need not necessarily be based on the same grammars or grammar formalisms as the editor.

OBJECTION (vii), "THE PRESENTATION MAY BE MISLEADING..."

The fact that most structure-oriented editors use textual presentation of structures resembling the conventional textual representation, as much as possible, sometimes leads to misleading or puzzling results. A few examples are:

- The same programming language can be described by several different grammars. The different structures of the grammars will affect the interaction in a structure-oriented editor, since the tree structure of the program is not conveyed by a textual presentation. If the user intends to select a structure by pointing at one of its concrete constituents, the result may be confusing if the grammar is not structured as the user expects.
- Nodes in the target tree not represented by any concrete syntax tokens cannot be directly referred to by pointing to their screen representation. Such nodes can be a result of the abstract grammar structure, e.g. a division of a construction into two sub-constructions. Another example is a list node with one descendant that does not have any concrete syntax before and after the desoendants. In a textual presentation, it is not possible to distinguish the list node from the descendant.
- Information present in an abstract syntax tree representation of a program may be lost in a textual presentation. A textual presentation may be ambiguous despite its being generated from an unambiguous abstract structure. One example is associativity and precedence in arithmetic expressions. The pre-

cedence and associativity are clearly defined by the tree structure. A textual presentation has to be extended, e.g. with parentheses, in order to obtain an unambiguous presentation. Furthermore, in order to obtain a presentation that uses parentheses only when needed, context-sensitive presentation may be required.

- The contents of a lexical token may, by accident, be confused with the textual presentation of surrounding syntactic categories. A lexical token with the contents "a*b" cannot be distinguished in a textual presentation from a syntactic structure with identical concrete syntax representing multiplication.

Our conclusion is that environments where programs are manipulated as structures should also visualize the structures explicitly. The only reason to hide the structure is to obtain a presentation that is identical to traditional text-oriented environments. This is, of course, worth aiming at in order to take advantage of the user's previous knowledge of programming languages. However, if the presentation format results in confusion, and available structural information is omitted, the advantages of compatibility with traditional presentation formats must be questioned. An alternative is to hide the structures completely, and both manipulate and present them as text, an approach utilized in Pan (Van De Vanter *et al.*, 1992).

Most of the above objections concern the interaction and presentation of structure-oriented editors. In order to make such editors accepted as usable tools, the interaction has to be improved. In the following chapters we analyse the interaction aspect more thoroughly.

3. An analysis of user activities

A model for user activities in human–computer interaction has been suggested by Norman (1986). The model involves seven approximate stages that characterize the user's mental and physical activities when interacting with a system. The stages comprise bridging the "execution gap" between the user's goals and physical actions as well as bridging the "evaluation gap" between the system's physical output and an evaluation with respect to the user's initial goals and intentions. Such a detailed analysis of user actions may be of use for minimizing the user's mental and physical effort when using a system, and in that way may bring the system closer to the user's needs and capabilities. The seven stages are:

(i) Establishing the goal
(ii) Forming the intention
(iii) Specifying the action sequence
(iv) Executing the action
(v) Perceiving the system state
(vi) Interpreting the state
(vii) Evaluating the system state with respect to the goals and intentions

When applying this model to program construction, the overall goal is to create a program or program component for a certain purpose. The goal gives rise to a number of intentions. Some intentions may be decomposed into a number of sub-intentions. When a sub-intention is at the level supported by the computer system, it may be executed. In program construction, this level consists of intentions in terms of the available language elements. The execution of an intention comprises specification of the action sequence and execution of the action. An intention of inserting, modifying or deleting a language element is translated into a number of edit operations realized by a number of physical actions.

After an action has been executed, the changed system state must be perceived, i.e. in this case the user has to perceive the appearance of the program after the modification. The result is then interpreted and evaluated with respect to the intention, to subgoals and to the overall goal. The evaluation may give rise to one or several new intentions that start a new cycle in the activity model.

Most stages in the model are mental activities. Only stage four is a physical activity. If is of profound importance that not only the physical effort but also the mental effort, is minimized in an interaction model. Making a reasonable trade-off may be difficult since the requirements of one stage may conflict with the requirements of other stages. In trying to minimize the physical activities, for instance, the required mental effort may increase.

Norman's model is an approximate description of user activities in human–computer interaction. It cannot be used for proving that an interaction model is optimal in some sense, or for making quantitative measurements of mental or physical effort. It can, however, be used to lead a qualitative discussion about interaction properties, and it can also be used for qualitative comparisons between different systems. We will thus make a qualitative comparison between editing programs as text and as structures based on Norman's model in order to try to pin down the differences. The comparison concerns editing programs with traditional screen-oriented text editors and structure-oriented editors based on traditional, menu-driven structure synthesis.

A COMPARISON BETWEEN TEXT-ORIENTED AND STRUCTURE-ORIENTED EDITORS

The task of defining the overall goal, e.g. to create a piece of functioning software, is mainly dependent on the functionality of the system as a whole. Hence, the interaction model will not influence this stage to any great extent. The forming of intentions is dependent both on the constructs available in the programming language and on the functionality of the editor. These intentions may occur at many different levels, e.g. "correct an error in this program", "add a procedure in the program", and "insert a character". The decomposition of intentions to lower-level intentions is a mental activity that should be minimized. The more levels we have to deal with, the more mental effort is needed. In a text editor the lowest level is single characters, whereas in structure-oriented editors the lowest level of intention directly corresponds to programming language concepts. The range of intention levels is thus larger in text editors, resulting in a potentially larger mental effort than in structure-oriented editors.

Having formed an intention, the user has to specify an action sequence. Action sequences for text editors include text-editing operations and insertion of characters. An action sequence corresponding to an intention at the language level may be of varying complexity, depending on the concrete syntax of a construction. Textually comprehensive constructions result in more complex action sequences, while textually simple constructions result in simple action sequences. The mental effort for specifying an action sequence varies depending on the user's experience. Skilled users may have "automated" the specification of action sequences and thus eliminated the lowest level of intentions, resulting in a considerably lower mental effort at this stage.

Structure-oriented editors typically have a more direct mapping from intentions at the language level to command sequences. The variation in complexity is lower and is not dependent on the concrete syntax of the language. It is, however, dependent on the abstract syntax which is reflected in the menus. The number of menu levels and the number of entries in each menu determine the complexity of the command sequence realizing an intention. Minimizing the mental effort caused by action sequence complexity in structure-oriented editors is an interaction issue to a large extent. The fact that command sequence complexity varies in text editors and is more constant in structure-oriented editors results in text editors that, in some cases, require a lower mental effort at this stage.

The physical effort for executing the command sequence depends both on the command sequence complexity and on the type of input device, e.g. keyboard and mouse, and the need to switch between different devices. The latter is hard to compare for text and structure-oriented editors in general. This ergonomic level, to a high degree, depends on the actual implementation of the primitives for executing command sequences, e.g. using pop-up menus, pull-down menus, control characters etc. Concerning the former, the physical effort is, like the mental effort, more constant in structure-oriented editors than in text editors.

After an action sequence has been executed by the system, the changes are perceived and interpreted by the user. The difference in perception between text editors and structure-oriented editors depends on the presentation form. We assume that both use textual presentation here. When the result is interpreted, the fine granularity of a textual program representation requires a significant amount of mental effort in text editors. However, a structure-oriented editor guarantees syntactic correctness. The effort for interpretation is hence considerably lower. Finally, the result is evaluated and compared with the intention which initiated the interaction cycle. As stated above, the interaction with text editors has more possible levels of intentions and will consequently also require potentially more mental effort for evaluation and comparison.

The discussion above, based on Norman's model, indicates that the interaction of structure-oriented editors, in many respects, is preferable to the interaction of text-oriented editors. In some cases, however, the specification and execution of command sequences require a lower mental and physical effort using text editors. A goal for structure-oriented editors is to reduce the complexity for all command sequences down to, or below the level of text editors. If this can be achieved, we can obtain an editor that, in most respects, is more suitable for program manipulation than text editors.

4. Mental models

Waters (1982) argues against pure structure editors. He claims that the textual and structural viewpoint complement each other, and that an editor should maintain both representations simultaneously. Different commands operate on each representation separately, and a modification to one representation will affect the other representation instantly. As a motivation for the design Waters states, "When you come up with a *better* way to do something, should you create a system which *mandates* its use or should you create a system which *offers* its use in addition to older methods? I think that the conservative approach of adding the new to the old is usually the best one." Water's view of structure-oriented editors seems to be rather widespread. We will, however, not address the arguments here, but merely point out that hybrid solutions, where functionality stems from different kinds of systems, may result in complicated conceptual models.

Norman (1986) argues that the conceptual model of a system is of primary importance in the design of man–machine interfaces. He distinguishes between two models: the *design model* and the *user's model*. The design model is the designer's conceptual model, which should be based on the user's task, requirements and capabilities. The design model is reflected in the *system image*. The system image is a physical concept concerning the presentation and the interaction with the system including display, commands, menus, error messages, documentation etc. Interacting with the system image, the user builds a conceptual model of the system, referred to as the user's model. The designer's goal should be a user's model compatible with the design model. This can only be achieved through a carefully designed system image.

Hybrid editors lack a clear, consistent design model compared with pure text and structure editors. The same program is represented in two different ways and the user has to keep track of both representations. The model is more complicated than it has to be in order to meet the requirements of the task. The deficiencies in the system model will be reflected in the system image. The two representations are manipulated by different command sets. The text-editing commands are not applicable to the structure representation, and structure-editing commands are not applicable to the text representation. Syntactic errors in the textual representation may either cause inconsistencies between the two representations, or cause the user to become trapped in a mode until the errors are corrected. One may argue that experienced users are capable of keeping track of two representations with disjoint command sets. They probably can, but it is certainly not desirable, and it would be particularly confusing for inexperienced users.

5. Use models

The evolution of *use models*, or metaphors (Carrol & Thomas, 1982), has to a large extent made computers accessible to inexperienced users. A use model takes advantage of a user's previous knowledge of the task domain. It can be of great value, particularly when previously manual tasks are computerized, but also for tasks that are specific for computers. A successful example is the desktop metaphor of the Macintosh finder. The knowledge in this case is not from the task domain of book-keeping in a file system, but from tasks in everyday life. By transferring knowledge from a familiar domain to similar computerized activities, the user can make use of well-known concepts and operations, and is relieved from learning a new "language".

Typewriters may serve as a use model for text editors. Although various editors make use of this use model more or less explicitly, the previous knowledge most users have from typewriting is of use when using a text editor. For structure-oriented editors, no such knowledge from other activities can be used. To the best of our knowledge, no use model has been suggested for structure-oriented editors.

6. Direct manipulation

The concept *direct manipulation* was coined by Shneiderman (1983). The essence of the concept is to supply the user with an interface providing a physical model of the real or some imaginary world, rather than using intermediary structures as command languages or menus. The concept was examined in detail by Hutchins, Hollan and Norman (1986), who described it in terms of directness in distance and engagement. It has been shown to be successful for concrete tasks, such as editing graphical objects or managing files in a file system. It has been shown to be less suitable for expressing abstractions and general behavior. However, structure editing is a concrete task in which a structure is manipulated. This is why we believe it is a suitable interaction model for structure editors.

A DIRECT MANIPULATION INTERFACE FOR STRUCTURE-ORIENTED EDITORS

Traditional interaction in structure synthesis is based on successive expansion of meta-nodes. The meta-nodes and the menu entries are often named in terms of production names in the grammar. A different approach for interaction is direct manipulation of syntactic categories represented in their concrete form. Figure 1 shows such an interface.†

The figure contains three parts: the target structure where a program is built, the palette where program pieces are fetched and the clipboard used for temporary storage of program fragments. The pieces, corresponding to different syntactic categories are displayed in the palette in their concrete form, i.e. in the same way as they are displayed in the target structure. They can be picked from the palette and inserted into the target structure using the mouse. Only pieces that "fit", i.e. have the appropriate syntactic category, can be inserted at a specific place. Three kinds of syntactic pieces exist in the palette: construction, list and lexical elements. A construction element has a specific number of descendants, represented as meta-nodes ("holes") in the palette, as well as in the target structure. The meta-nodes in the palette are just for presentation and may not be expanded. A list element has an arbitrary number of sub-components of the same syntactic category, represented as a syntactic piece with two "overlapping holes". Lexical elements contain text entered from the keyboard.

The representation in the palette may be generated from language grammars to have the same appearance as in the target structure. The target structure is presented in a semi-graphical manner. The syntax is textual, but sub-components are displayed as boxes in order to make clear what syntactic units constitute a structure. Finally, the clipboard is used as a workspace where arbitrary fragments can be built and later be pasted on to the target structure. The clipboard may also be used for storing more complex fragments that are used frequently.

† The figure is not a screendump from our system.

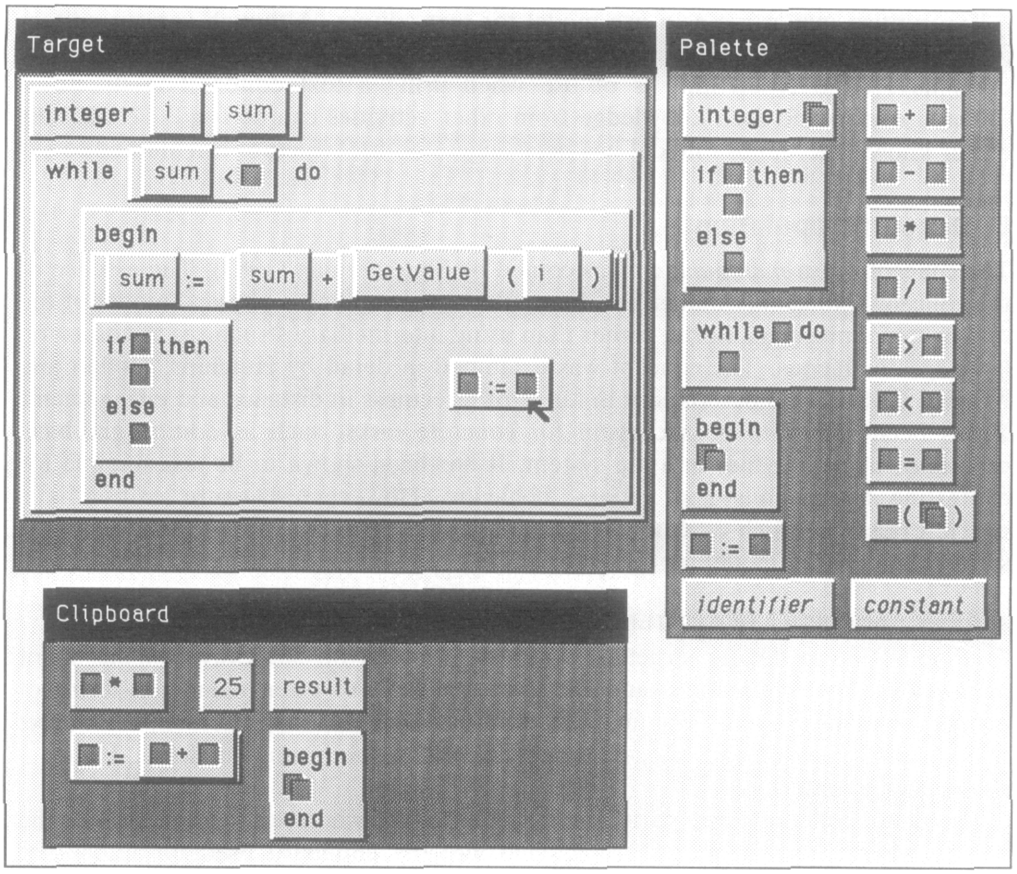

FIGURE 1. A direct manipulation interface for structure-oriented editors.

The editing functionality in such an interface may be realized in a more intuitive manner than in traditional structure synthesis. Instead of expanding a meta-node by giving a command and choosing the expansion alternatives from menus, a syntactic piece is dragged from the palette to the destination meta-node. This is similar to the technique used in many interactive graphical editors. A piece may also be dragged from the palette to a position between elements in a list, resulting in an extended list. Cut-and-paste operations are realized by dragging structures between the target structure and the clipboard. Structure transformations are expressed by dragging a structure from the palette and placing it on an existing structure instead of on a meta-node. The semantics of the transformation are, however, determined by how transformations are implemented.

One observation is that the interaction model suggested does not necessarily need a focus of attention (cursor) for expansion, since structures are dragged from the source to the destination. An interface with fewer states can thus be obtained. However, the model does not exclude the use of a cursor. A cursor can serve as the current point of insertion. The user then just needs to select an entry in the palette in order to insert it as the cursor position. The use of a cursor can be seen as a shortcut to repeated dragging of pieces to succeeding positions. It may co-exist with the stateless approach of dragging pieces.

The interaction model described so far does not give the user any guidance about what syntactic pieces are applicable to insertion at a certain point in the target structure. When a syntactic piece is dragged to the target structure, it is only accepted at a syntactically correct place. However, guidance can be supplied without violating the principle of direct manipulation. By pointing at a meta-node in the target structure, the applicable syntactic pieces may be highlighted in the palette. In this way, the benefits of user guidance may be obtained without forcing it upon the user as discussed in Section 2.

HOW DIRECT IS THE INTERFACE?

The characteristics of direct manipulation interfaces have been investigated by Hutchins *et al.* (1986). They consider two aspects of directness in an interface important, *distance* and *engagement*. Distance is a "measure" of the distance between the user's intentions and the physical requirements of the system. It can be divided into two forms of distance, semantic distance and articulatory distance. The former describes the ability to express the user's intensions in a system. It is not a property of the interface alone, but of the system's functionality as a whole. In a system that is semantically direct, the user is able to express his/her intentions in a straightforward fashion without the need to translate it into a complicated expression. The latter concerns the *form* of an expression that is used to turn the functionality, to account both for input and output. The form of an expression should be related to its meaning, whether it consists of command names, graphical presentation, mouse movements or auditory signals.

According to Hutchins *et al.* (1986), direct engagement is a matter of the user's feeling of being involved directly with a world of objects rather than communicating with an intermediary. It is a matter of creating an illusion, making the user interface and the computer "invisible". A necessary property for obtaining direct engagement is that input and output languages are inter-referential, i.e. an input expression is allowed to incorporate or make use of previous output expressions. It is also necessary that the system is responsive and unobtrusive, not interfering or intruding. It must also provide the user with a world of objects to interact with, and the objects must be felt to be the objects of interest.

A COMPARISON WITH THE TRADITIONAL APPROACH

On what basis do we claim that the presented interface for structure synthesis is direct manipulation, and that the traditional approach for structure synthesis is not?

Direct manipulation is not a concept that can be clearly and exhaustively described. It is thus, not possible to define what is, and what is not, direct manipulation. We can only evaluate the characteristics of the interfaces in terms of distance and engagement as described above, and conclude that one system is more direct than another.

The basic functionality of a structure-oriented editor need not be modified in order to use the interface. The semantic distance has consequently not changed. The articulatory distance has changed since the form of output and particularly input expressions is different from those in traditional structure synthesis. The presentation in traditional structure synthesis is imitating the presentation in textual editors. The presentation of a syntactic piece in Figure 1 is still textual, but the syntactic pieces are displayed as objects that can be manipulated. The form of the expressions for manipulating structures is more direct than in traditional structure synthesis. The insertion of a syntactic piece in a structure is realized by picking it from a "box of pieces" and physically dragging it to the destination rather than using commands or menus. Cutting, pasting and transformations are likewise realized by physical actions rather than using menus.

There are several reasons for claiming that the engagement is more direct as well. The syntactic pieces and structures are the objects of interest in structure synthesis. The objects form a physical model, and interaction with the objects is not done via any intermediary such as command languages and menus. Input and output expressions are inter-referential. The display of changes in a structure of syntactic pieces is used as an object for further input. There is no reason to believe that the responsiveness should be lower than in traditional structure synthesis (there is, actually, no reason to believe that it should be higher either). The only occasion when the system intrudes is when a user tries to perform an action that would result in a syntactically incorrect structure. This speaks in favor of traditional structure synthesis where not even the trial to create incorrect structures is possible. However, by an unobtrusive error response we believe that the engagement will not suffer. The most obtrusive error response is to ignore an action that will cause a syntactically incorrect structure. Since direct manipulation interfaces rely on the visibility of the effects of operations, the "do nothing" error response may be appropriate (Lewis & Norman, 1986). Finally, the interface described here is closer to creating the illusion that the user is manipulating pieces of the program itself, rather than communicating with a user interface of a computer.

CONCEPTUAL MODELS

The design model, as discussed in Section 4, is fairly uncomplicated. It is based on the syntactic structure and a small number of operations. The system image in the model for interaction presented above does not cause any additional complexity, as is the case with hybrid editing. It would thus make the formation of a user's model compatible with the design model considerably easier.

USE MODELS—THE LEGO METAPHOR

The situation concerning use models, or metaphors, is similar to that in the desktop metaphor of the Macintosh finder. The previous knowledge in the task domain is that of computerized tools and methods we consider inappropriate. It would not be a good idea to use an obsolete metaphor from such tools. What we can do is find a well-known metaphor from everyday life that resembles the task carried out in structure synthesis to a large extent. We would like to suggest the *LEGO*† *metaphor*: a program is considered to be a construction consisting of pieces that are physically picked from a box and may be inserted only in places where they fit. Most operations resemble the actions of building with LEGO toy bricks (even if structure editing is an advanced form of a LEGO construction since pieces may recursively

† LEGO is the trademark of small plastic toy bricks that may be put together in various configurations.

contain other pieces). Furthermore, some operations, such as transformations where a piece is forced on to another, are more powerful than what is possible with ordinary LEGO bricks. Despite the fact that structure synthesis is considerably more flexible and powerful than building with LEGO bricks, we believe that the metaphor is useful for users lacking experience of a particular editor, or in structure-oriented editing in general.

SHORTCUTS

Finally, a few words about the physical effort needed for command sequence specification, or the ergonomic level of the interaction. Will users accept assembling program structures by picking them from a palette and dragging them to the target structure? Is the relatively frequent alternation between mouse and keyboard input acceptable? The latter problem is not unique for the interaction model presented here, but is rather an ergonomic problem in direct manipulation interfaces in general. Even if the ergonomic level may be considered as an issue of secondary interest compared with higher levels, its practical importance cannot be denied.

One solution is to provide *shortcuts,* allowing the keyboard to serve as an alternative to mouse input. Shortcuts may be more efficient, in terms of keystrokes and physical movements, and may be preferred by experienced users. The solution is, however, somewhat discouraging, since it violates the principles of direct manipulation. Despite that, the use of shortcuts can be motivated in order to minimize the physical effort. The shortcuts serve as an alternative to input and affect the interaction at the surface only. A cursor is used for pointing out the source or destination for the operations as discussed earlier. Cursor movements, as well as cut-and-paste operations may be carried out using standard techniques with arrow and command keys. The selection of syntactic pieces from the palette can be defined in terms of the concrete syntax. After entering a text, it is matched with the concrete syntax in the palette entries, and the syntactic piece of the entry will be inserted at the cursor position. The match of the input text and the palette entries may allow input to be abbreviated as long as one palette entry can be determined unambiguously. Entering a "be", for instance, results in the begin–end piece to be selected from the palette and inserted at the current cursor position. This ability to enter program text in terms of the concrete syntax should not be confused with parsing. It is only an alternative way of selecting items from the palette.

7. Discussion

In this paper we have identified a number of interaction problems with traditional structure synthesis which make many users reluctant to use structure-oriented editors. The interaction in structure synthesis for program construction has been compared with text editing. The comparison indicates that structure synthesis has the potential to be more suitable for program construction than text editing, despite the fact that structure synthesis has not been accepted among users in general. It also pointed out the cases where text editing is more efficient than structure synthesis. Hybrid editing has been discussed as a candidate for solving the interaction problems by combining the two approaches. A drawback of this approach is a complicated design model. Pure structure synthesis with a direct manipulation interface has been proposed in order to overcome the interaction problems. In this chapter the model will be related to the model for user activities, the objections to structure synthesis will be further commented upon, and a few remarks on the applicability of the approach will be given.

NORMAN'S MODEL FOR USER ACTIVITIES REVISITED

Recall from Section 3 that the stages in Norman's model of user activities where text editing in some cases is advantageous to structure synthesis are the specification and execution of action sequences. The mental and physical effort required for specification and execution of action sequences is constant in the direct manipulation interface, since it is not, or only to a very small extent, dependent on the target language syntax and the kind of operation. Because of the relatively small articulatory distance, the mental effort required would be low and comparable to the cases where a text editor is most favorable. An example is editing expressions in Pascal. Adding a "plus" using a text editor is done by positioning the cursor (if it is not already in position) and finding and pressing the " + " key. The corresponding action in the proposed interface is finding the plus entry, selecting it with the mouse, and dragging it to the destination. Alternatively, if a cursor is used as the point of insertion and it has already been located at the intended position, the same can be done just by finding and selecting the "plus" entry. The difference in mental effort required seems to be negligible between the general case of the direct manipulation interface and the best cases of text editing. The physical effort may be somewhat higher. In order to minimize the physical effort, shortcuts may be used as discussed in the previous chapter.

THE SEVEN OBJECTIONS REVISITED

Some of the objections to the interaction in program synthesis presented in Section 2 deserve a few further comments. The first objection (i), "structure synthesis is awkward, particularly for expressions", does not hold true for the presented interaction model as discussed above. The second objection (ii), "programs have to be entered top-down", is, likewise, false. The clipboard facility allows program fragments to be constructed bottom-up as well as top-down. Objection (iii) claims that "transformations may be done more efficiently with a text editor". This depends on the semantics of the transformation operation. The interaction for using transformations is, however, very straightforward, which is why transformations would be natural and efficient to use.

Objection (iv) claims that "the enforced syntactic consistency is limiting the user's freedom". We believe that the direct manipulation of syntactic pieces will not give the user the feeling of being constrained. It will, hopefully, be regarded as a natural and intuitive manner in which to construct programs. Objection (v) argues that "expert users do not need the guidance provided". Guidance is not forced upon the user as in traditional structure synthesis. It is provided on the user's request. Objection (vi) is that "structure-oriented editors use a different representation than traditional text-based tools". The direct manipulation interface has not changed that: they still do. Finally, the problems concerning the presentation, objection (vii), have been overcome by the semi-graphical presentation. The program is both edited and presented as a structure. Thus, there will be no confusion resulting from a textual presentation where some structural information is omitted.

LIMITATIONS

In this paper the emphasis has been on interaction issues of structure synthesis for *program* construction. As mentioned in Section 1, structure-oriented editors have been used in various other areas, e.g. document processing and graphical editing. The proposed interaction model implies both practical restrictions and possibilities compared with traditional structure synthesis and hybrid editors when using it for different purposes. Hybrid editors use parsing as alternative input, which restricts them to handling structures that have a textual representation. The proposed model does not imply such restrictions. The model allows syntactic pieces to be graphical objects or icons. The size of the palette does, however, put some physical restrictions on what languages can be used in practice. Using a structure-oriented editor as a command language interface (Minör, 1987), to an operating system, would probably not be practical using a palette, since such languages are often very large. For form fill-in applications (Shneiderman, 1986) it would not be realistic to keep several forms in a palette. In other words, the language should be relatively small and not too verbose, if the interaction model is to be applicable in practice.

The proposed direct manipulation model for interaction does not exclude the use of traditional interfaces. It may be combined with conventional menu-based interfaces or with text editing and parsing in a hybrid editor. However, this may result in a more complicated design model, as discussed previously. The possible advantages of a hybrid solution must be carefully considered with this in mind.

This is surely not the last word on the interaction of structure-oriented editing. The implementation of the direct manipulation interface in the SbyS editor (Minör, 1990) is still at the prototype level and no quantitative evaluation to verify the qualitative comparison has been performed. We do, however, believe that the interaction model is one step towards structure-oriented editors based on structure synthesis with a simple and clear design model, which may be appreciated by users at different levels of expertise.

8. Issues for further research

Finally, we will discuss a few other issues besides the interaction, which are important to make structure-oriented editors more usable. One is objection (vi) in Section 2, about the representation of structures. Text-based systems have the advantage of a standardized representation. Programs and other documents may thus, easily be ported between different systems. A standardized representation for abstract syntax trees is crucial for structure-oriented environments and is one area where further work is required.

The problem that programs have to be entered top-down, objection (ii), is solved in the proposed interface by a clipboard facility. A bottom-up construction facility in the editor, where the user may freely enter structure fragments and later combine them into a syntactically correct structure, would support bottom-up construction more explicitly. In that case, syntactic consistency would not be enforced in every step of program construction and inconsistent states would be possible, as mentioned in objection (iv). Further research in this area may result in structure-oriented editors that constrain the user less.

Another area where more research is needed is semantic support. Structure-oriented editors only give the user syntactic support. In several structure-oriented environments, static semantics are *checked* incrementally. The semantic information could be utilized to *support* the user to construct semantically correct programs, e.g. by presenting semantically correct structures in a menu. In the Mjølner/Orm environment (Magnusson *et al.*, 1990), this technique based on attribute grammars has been utilized (Hedin, 1992). One step in this direction is also described by Bahlke and Snelting (1992). We believe that the semantic information available in structure-oriented environments can be further exploited to support the user, both for presentation, navigation and editing. In this way the advantages of using structure-oriented editors and environments may be apparent to the user, and he/she may be less reluctant to use a new tool.

Finally, new application areas where structure-oriented editors are suitable must be explored. Until now, structure-oriented editors have mostly been used for editing programming languages. One application area is application-oriented languages, or little languages (Bentley, 1986). Several applications have an embedded language, e.g. command languages in operating systems, database query languages, Hypertalk in Hypercard etc. Typical for many such systems is that users use them occasionally and never become experts in their embedded language. The support offered by structure-oriented editors is then important to the users. We believe that this is an area where structure-oriented editors have a great potential.

The work reported in this paper has been a part of the Mjølner project. I want to thank all the other members of the Mjølner project at Lund University: Mats Bengtsson, Lars-Ove Dahlin, Göran Fries, Anders Gustavsson, Görel Hedin, Boris Magnusson, Dan Oscarsson and Magnus Taube, for stimulating discussions that have contributed to this paper. I am also grateful to Sten Henriksson for many valuable comments on the style and contents of earlier drafts of this paper. Finally, I want to thank the editors of this special issue for carefully proofreading the paper and suggesting many improvements. The Mjølner project is supported in part by STU—*Styrelsen för Teknisk Utveckling* (The Swedish National Board for Technical Development) and *Nordisk Industrifond* (The Nordic Fund for Industrial Development).

References

Backlund, B., Hagsand, O. & Pehrson, B. (1990). Generation of visual language-oriented design environments. *Journal of Visual Languages and Computing,* **1,** 333–354.

Bahlke, R. & Snelting, G. (1992). Design and structure of a semantic-based programming environment. *International Journal of Man–Machine Studies,* **37,** 467–479.

Bentley, J. (1986). Little languages. *Communications of the ACM,* **29,** 711–721.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B. & Pascual, V. (1988). CENTAUR: the system. *Proceedings of the ACM SIGSOFT '88: Third Symposium on Software Development Environments.* pp. 14–24, Boston, MA.

Carrol, J. M. & Thomas, J. C. (1982). Metaphor and the cognitive representation of computing systems. *IEEE Transactions on Systems, Man, and Cybernetics,* **SMC-12**(2), 107–116.

Conklin, J. (1987). Hyptertext: an introduction and survey. *IEEE COMPUTER,* September, 17–41.

DONZEAU-GOUGE, V., HUET, G., KAHN, G. & LANG, B. (1980). Programming environments based on structure editors: the MENTOR experience. INRIA Research Report, No. 26, 1980. Also in P. R. BARSTOW, *et al.*, Eds. *Interactive Programming Environments.* pp. 128–140. New York: McGraw-Hill.

DONZEAU-GOUGE, V., KAHN, G., LANG, B. & MELESE, B. (1984). Document structure and modularity in Mentor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* pp. 141–148. Pittsburgh, PA.

FISCHER, C. N., JOHNSON, G. F., MAUNEY, J., PAL, A. & STOCK, D. L. (1984). The Poe language-based editor project. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* pp. 21–29. Pittsburgh, PA.

FRASER, C. W. (1981). Syntax-directed editing of general data structures. *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices,* **16,** 17–21.

GÖTTLER, H. (1992). Diagram editors = graphs + attributes + graph grammars. *International Journal of Man–Machine Studies,* **37,** 481–592.

HEDIN, G. (1992). *Incremental Semantic Analysis.* Ph.D. thesis, Department of Computer Science, Lund University, Sweden.

HUTCHINS, E. L., HOLLAN, J. D. & NORMAN, D. A. (1986). Direct manipulation interfaces. In D. A. NORMAN & S. W. DRAPER, Eds. *User Centered System Design,* pp. 87–124. Hillsdale, NJ: Lawrence Erlbaum Associates Inc.

LEWIS, C. & NORMAN, D. A. (1986). Designing for error. In D. A. NORMAN & S. W. DRAPER, Eds. *User Centered System Design,* pp. 411–432. Hillsdale, NJ: Lawrence Erlbaum Associates Inc.

MAGNUSSON, B., BENGTSSON, M., DAHLIN, L. O., FRIES, G., GUSTAVSSON, A., HEDIN, G., MINÖR, S., OSCARSSON, D. & TAUBE, M. (1990). An overview of the Mjølner/Orm environment: incremental language and software development. *Proceedings of TOOLS '90 Conference on Technology of Object-oriented Languages and Systems.* pp. 635–646, Paris.

MINÖR, S. (1987). Structured command interaction based on a grammar interpreting synthesizer. *Proceedings of the Second IFIP Conference on Human–Computer Interaction.* pp. 731–737, Amsterdam: North-Holland.

MINÖR, S. (1990). *On structure-oriented editing.* Ph.D. thesis, Department of Computer Science, Lund University, Lund, Sweden.

NORMAN, D. (1983). Design principles for human–computer interfaces. *CHI'83 Conference Proceedings.* pp. 1–10, *SIGCHI Bulletin.*

NORMAN, D. A. (1986). Cognitive engineerign. In D. A. NORMAN & S. W. DRAPER, Eds. *User Centered System Design,* pp. 31–61. Hillsdale, NJ: Lawrence Erlbaum Associates Inc.

NOTKIN, D. (1984). *Interactive structure-oriented computing.* Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA.

NOTKIN, D. (1985). The GANDALF project. *Journal of Systems and Software,* **5,** 91–105.

NØRMARK, K. (1990). A Programming Environment for CLOS Based on Hyper Structure, Research report, Institute of Electronic Systems, Aalborg University, Aalborg, Denmark.

REPS, T. & TEITELBAUM, T. (1984). The synthesizer generator. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Environments.* pp. 42–48, Pittsburgh, PA.

SANDVAD, E. (1989). *Hypertext in an object-oriented programming environment.* Report No. DAIMI PB-280, Computer Science Department, Aarhus University, Aarhus, Denmark.

SHNEIDERMAN, B. (1983). Direct manipulation: a step beyond programming languages. *IEEE COMPUTER,* August, 57–68.

SHNEIDERMAN, B. (1986). *Designing the user interface.* Reading, MA: Addison-Wesley.

STALLMAN, R. M. (1981). EMACS the extensible, customizable, self-documenting display editor. *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation,* SIGPLAN Notices. **16,** 147–156. Also in P. R. BARSTOW, *et al.*, Eds. *Interactive Programming Environments*, pp. 300–325, New York: McGraw-Hill.

STRÖMFORS, O. (1986). Editing large programs using a structure oriented editor. In R. CONRADI *et al.*, Eds. *Advanced Programming Environments, Lecture Notes in Computer Science,* **244,** 39–46, Berlin: Springer-Verlag.

SZWILLUS, G. (1987). GEGS—a system for generating graphical editors. *Proceedings of the Second IFIP Conference on Human–Computer Interaction,* pp. 135–141, Amsterdam: North-Holland.

TEITELBAUM, T. & REPS, T. (1981). The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM,* **24,** 563–573.

VAN DE VANTER, J. L., BALLANCE, R. A. & GRAHAM, S. L. (1992). Coherent user interfaces for language-based editing systems. *International Journal of Man–Machine Studies,* **37,** 431–466.

WALKER, J. H. (1981). The document editor: a support environment for preparing technical documents. *Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, SIGPLAN Notices,* **16,** 44–50.

WATERS, R. C. (1982). Program editors should not abandon text oriented commands. *SIGPLAN Notices,* **17,** 39–46.

WELSH, J. & TOLEMAN, M. (1992). Conceptual issues in language-based editor design. *International Journal of Man–Machine Studies,* **37,** 419–430.

WINOGRAD, T. A. (1987). *Muir: a tool for language design.* Report No. STAN-CS-87-1159, Department of Computer Science, Stanford University, Palo Alto, CA, USA.