A Generic, Customisable, Hybrid Structure-Oriented Editor

Koen De Hondt

Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
email: kdehondt@vnet3.vub.ac.be

Abstract

After a brief evaluation of some design choices of existing structure-oriented editors, several design aspects of the Agora Structure Editor (ASE) are discussed. In the design of ASE ergonomics has played a very important role. This contrasts strongly with existing systems, which seem to neglect it altogether.

1 Introduction

Since the work of Teitelbaum and Reps on the Cornell Program Synthesizer [Teitelbaum & Reps 81], research on structure-oriented editors and structure-oriented environments has boomed. In the last decade many software development environments featuring a structure-oriented editor were developed, and today structure-oriented editing still is an active research topic. Several approaches have been taken. Ad-hoc implementations for existing programming languages have been made, structure-oriented editor generators have been developed, structure-oriented editors have been integrated in generic environments. Several formalisms suitable for structure-oriented editing were developed in order to represent programs internally. Besides for programs, structure-oriented editing has also been used for manipulation of structured documents.

Up until now, very few prototypes have left the research laboratories. Due to deficiencies in their design, current structure-oriented editors are hard to use. They have to struggle against the well designed text editors of today, with which programmers have a long experience. As long as ergonomics of structure-oriented editors does not improve, the average programmer will be reluctant to use them.

We have evaluated several structure-oriented editors to pinpoint their deficiencies. Such an evaluation is not an easy task, since not all structure-oriented editor implementations are made public. In the literature several systems are reported on, but issues concerning ergonomics, in which we are particularly interested, are seldom discussed. Many documents describe how a structure-oriented editor is designed on the inside, but fail to mention what the *look and feel* is on the outside.
The results of our evaluation have served as basis for the design of a structure-oriented editor for Agora [Steyaert & al. 93], an object-oriented language developed at our lab.

2 Evaluation of Existing Structure-Oriented Editors

We have evaluated several structure-oriented editors based on one report for each editor (see the references), while two recent systems have been evaluated in detail.
SbyS, or Structure by Structure, is evaluated based on [Minör 90], [Minör 91], [Mjølner 91] and the implementation in the Mjølner/Orm environment, version 1.3. Working in Orm, an abstract grammar

of a target language was specified and the resulting structure editor for that language was used to write programs with. The abstract grammar was also specified with SbyS.

GSE, or Generic Text and Structure Editor, is evaluated based on [Koorn 92], [Centaur 92], [Hendriks 91] and the implementation in the Centaur environment, version 1.2. Centaur uses GSE for language specifications in the SDF formalism. An SDF specification of a target language was given using GSE. Since more specifications in other formalisms had to be given before Centaur could generate an environment for the target language, GSE was not used to write target language programs with.

2.1 Structure Synthesis

Structure editors basically provide three operations for program construction:

• expand	to replace a placeholder by a template
• insert	to add a template to a list of templates
• remove	to delete a template or a placeholder

Structure synthesis is done by selecting a program fragment and giving an expand command. Older editors have as many commands as templates, recent editors display a (hierarchical) menu containing all expansion alternatives.

Traditional structure editors only support expanding placeholders, that is, replacing a placeholder by an appropriate grammatical construct as defined by the language's grammar. As a consequence changing the component of a grammatical construct requires two editing steps : removal of the component and expansion of the placeholder appearing after the removal. It is clear that such editors lack a general replacement operation.

Structure synthesis ensures syntactical correctness of the synthesised program (modulo the non expanded placeholders). Due to the limited editing operations some changes to the program are hard to make. It takes a sequence of actions to carry out "complex" changes such as, for example, substituting a program fragment by another one containing that fragment. Even "simple" changes take several actions before completion: Changing a Modula-2 procedure to a function is a simple task when using a text editor (adding a colon and a function type), but becomes a complicated one when using a pure structure editor (cutting the formal parameter list, cutting the procedure body, removing the procedure template, inserting a function template, pasting the body, pasting the formal parameter list, expanding the function type).

2.2 Structural Transformation

Structural transformation is the replacement of a subtree of the program by another one that is a transformation of the original. The two subtrees belong to the same syntactic category. For example, replacing a statement by a loop statement containing the original statement as the body is a structural transformation. Structural transformation solves the problems discussed at the end of the previous section.

In [Shani 83] it is stated: "The reason for preferring textual commands to perform major structural changes in a program is the lack of appropriate structural commands in current editors.". To date, still little editors support structural transformations, and if they do, the transformers are usually hard coded.

Clearly, a structure editor should be open-ended concerning transformers. A fixed set of transformers is unsatisfactory. It limits user freedom. A user should be able to add, remove and change transformers. Hard coding them precludes modification and should therefore be avoided.

It is striking that relatively recent structure editor environment generators, like Centaur [Hendriks 91][Centaur 92], DOSE [Kaiser & al. 88], Pregmatic [van den Brand 92] and generic structure editor environments, like Orm [Mjølner 91], do not provide any formalism to describe structural transformations, as if structural transformation is felt to be redundant. We can only regret this fact. Structural transformation is an indispensable structure-oriented editing tool.

2.3 Hybrid Editing

In the last decade there has been some discussion on the use of text-oriented commands in structure-oriented editors [Waters 82][Shani 83][Notkin & al. 83]. Now, it is widely accepted[1] that allowing text editing in structure-oriented editors solves many interaction problems of pure structure-oriented editing, although many existing editors restrict the way text editing should be performed by the user.

2.4 Other Aspects

Besides the aspects handled above, the following comments can be made about current systems: Direct manipulation interfaces do not exist, sometimes navigation commands are awkward, search facilities are not provided, ergonomics is almost always neglected. Discussing these topics is beyond the scope of this paper, but a full discussion can be found in [De Hondt 93].

3 A Generic Structure-Oriented Editor

The Agora Structure Editor (ASE) is generic, that is, it is designed to be independent of the target language. It uses the structured grammar formalism (see next section) to describe language grammars. The production rules of the grammar are represented internally by instance of themselves (using placeholders for non terminals). ASE interprets the production rules during editing operations. As a consequence one can plug-in an arbitrary structured grammar in order to utilise ASE for writing programs in the corresponding language.

3.1 Structured Context-Free Grammars

Standard grammars are felt to be too general for use in a structure editor environment. Therefore several formalisms, restricting and structuring parsing grammars and abstract grammars, were developed in the eighties. [Minör 90] gives an overview of these formalisms, of which the GRAMPS formalism [Cameron & Ito 84] influenced the structured grammar formalisms used in Mjølner [Madsen & Nørgaard 87][Minör 90] and the Agora Structure Editor.

A structured context-free grammar is a tuple $G = (N, T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$, with N the set of non terminals, T_a the alphabet (identifiers and literals), T_s the surface syntax (reserved words and punctuation marks), S the start non terminal and P_A, P_C, P_R, P_L, P_X the sets of alternation rules, construction rules, repetition rules, list rules and lexeme rules respectively. The production rules have one of five possible forms.

$A \rightarrow Y_1 \mid Y_2 \mid \dots \mid Y_k$	alternation rule
$C \rightarrow c_1\ Y_1\ c_2 \dots Y_k\ c_{k+1}$	construction rule
$R \rightarrow b\ Y\ s^{*}\ e$	repetition rule
$L \rightarrow b\ Y\ s^{+}\ e$	list rule
$X \rightarrow \text{LEXEME}$	lexeme rule

Two production rules have different left-hand sides. An alternation rule $A \rightarrow Y_1 \mid Y_2 \mid \dots \mid Y_k$ specifies that $Y_1, \dots, Y_k$ are derivations of A. The non terminals in the right-hand side are mutually different. A construction rule $C \rightarrow c_1\ Y_1\ c_2 \dots Y_k\ c_{k+1}$ specifies a *heterogeneous* aggregation of non terminal symbols with the corresponding surface syntax. This form is equal to the one found in standard context-free grammars. A repetition rule $R \rightarrow b\ Y\ s^{*}\ e$ specifies a *homogeneous* aggregation of non terminal symbols with the corresponding surface syntax. The rule must be read as follows : R is a repetition of *zero or more* Ys separated by separator *s* and bracketed by begin and end delimiters *b* and *e*. A list rule $L \rightarrow b\ Y\ s^{+}\ e$ is a repetition of *one or more* Ys. A lexeme rule specifies that a

[1]SbyS is a notable exception. It is a pure structure editor. In [Minör 90], Minör argues against mixing text-oriented and structure-oriented editing.

syntactic construct has an associated string. The format of that string is considered a lexical issue, not a grammatical issue. Therefore it is not specified in the grammar.
One could argue that repetition rules are redundant, because repetitions could as well be described by recursive construction productions (cf. Chomsky Normal Form). However, repetition productions express repetitions in a natural fashion and they also result in a more convenient interaction in structure editors.
One can prove [De Hondt 93] that for any context-free grammar $G = (N, T, P, S)$ with $T = T_a \cup T_s$ and $T_a \cap T_s = \emptyset$ there exists a structured context-free grammar $SG = (N', T_a, T_s, P_A, P_C, P_R, P_L, P_X, S)$ that generates the same language. Proof follows from the fact that in standard context-free grammars repetition rules can be described with recursive construction rules.

3.2 Internal Representation

From structured context-free grammars, structured abstract grammars can be derived. The production rules of a structured abstract grammar have the same format as the production rules of a structured context-free grammar, but surface syntax is stripped. ASE employs abstract syntax trees based on structured abstract grammars to represent programs internally. It extends these syntax trees with objects representing placeholders.
The production rules of the target language's grammar are represented by abstract syntax trees containing placeholders. The placeholders represent the production rule's right-hand side non terminals. ASE interprets the production rules during editing to check whether structure-oriented commands are applicable at a given time.

4 Look and Feel of the Agora Structure Editor

From the evaluation in section 2 it is clear that the design of present day structure-oriented editors leaves room for improvement. The major goal of the Agora Structure Editor (ASE) project was to tackle the problems existing structure-oriented editors suffer of. ASE is intended to be a generic, customisable, hybrid structure-oriented editor in which ergonomics plays a central role. ASE is generic because it is designed to be target language independent. It is highly customisable because structure-oriented commands can be adapted to the user's needs. It is intended to be hybrid in order to freely mix text-oriented editing and structure-oriented editing.

4.1 Structure Synthesis

ASE comes with the following structure synthesis operations: replace/expand, unexpand, insert before/after, remove. The replace operation replaces an arbitrary structural selection by another picked from a set of alternatives. Expansion is replacement *of* a placeholder, unexpansion (shrinking) is a replacement *by* a placeholder. The insert before/after operations insert arbitrary program fragments picked from a set of alternatives. When replace and insert commands are given, the user is presented a pop-up menu containing all program fragments that can be substituted and inserted.

4.2 Customising Structure Synthesis Commands

ASE allows the user to customise the set of expansion alternatives used with replace and insert operations. With every production rule of the target language's grammar a pop-up menu is associated. These pop-up menus are nested according to the possible derivations, as laid down by the target language's grammar. The nesting can be determined by the user. Although more than two levels are discouraged from an ergonomic point of view, the user is free to nest as he likes, as long as the language's grammar is not violated.

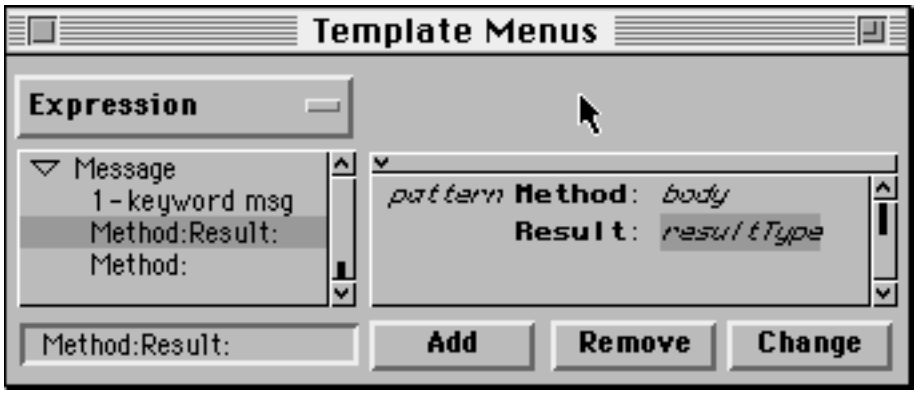


Figure 1

Figure 1 shows the interface with which the user is able to modify the expansion alternatives. On the left, (part of) the contents of the 'Expression' menu is displayed. The triangular marker indicates that 'Message' is a submenu of the 'Expression' menu. The 'message' submenu contains three items, of which one is selected. On the right, the program fragment (the expansion) corresponding with the selection on the left is displayed. Expansions are represented internally as abstract syntax trees. The text view on the right is an instance of ASE, which manipulates the expansion.

4.3 Structural Transformation

A structural transformation is an association between two structural patterns, the source and destination patterns. The former specifies what program fragments the transformation is applicable on. The latter specifies what the source looks like after transformation.
Transformation is performed by:

- pattern matching a target tree to the source pattern, associating with each placeholder in the source the matching subtree of the target
- replacing the target tree by a copy of the destination pattern with each placeholder substituted by its associated subtree originating from the pattern match

The transform command in the Tools menu (see figure 2) puts up a menu containing all structural transformations applicable on the current selection. The editor determines applicability of the transformations by pattern matching the current selection with each transformer source pattern.

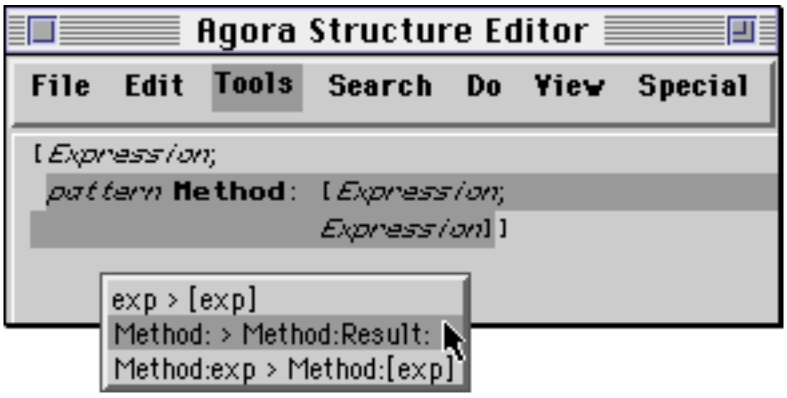
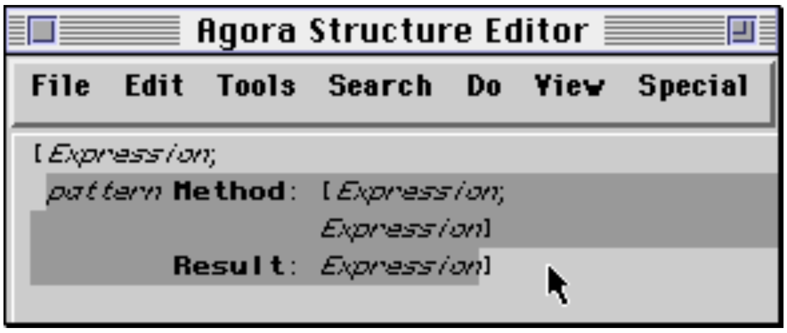


Figure 2

When the menu item under the mouse pointer is selected, the current selection is transformed according to the transformation having the name 'Method: > Method:Result:'. This transformation changes an Agora imperative method declaration into a functional method declaration, as shown in figure 3.



```
[Expression;
 pattern Method: [Expression;
                  Expression]
         Result: Expression]
```

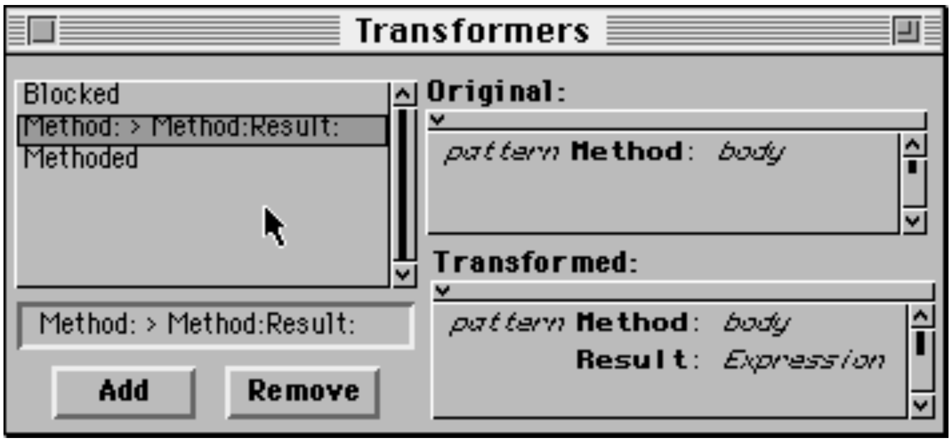
Figure 3

This approach to structural transformation seems very promising. Nevertheless, investigation is still to be done to make the pattern matching technique more powerful. The current matching technique only supports matching a tree to a placeholder, but, in practice, matching a list of trees to a placeholder is necessary to construct more powerful transformations. For example, one wants to be able to transform a list containing some component into one without that component.

This technique is simple, yet powerful. Therefore it is surprising that other environments do not provide a similar transformation operation.

4.4 Customising Structural Transformations

Structural transformations of the Agora Structure Editor are not hard coded, but user-defined. Providing a language to manipulate the internal representation of a program, probably seems appealing to many (cf. Mentol in [Donzeau-Gouge & al. 84]), but adding such a language to the environment puts a burden on the user who has to learn an additional language *and* the internal representation used by the structure editor. Therefore another approach was taken. ASE's set of structural transformations can be interactively modified through the interface depicted in figure 4. The selected structural transformation is the one used in figures 2 and 3. The 'original' and 'transformed' fields define the source and destination structural patterns. The source and destination patterns specify that the selected structural transformation can be applied to any subtree matching a method declaration and that it transforms that method declaration into a functional method declaration with the same pattern and body.



Blocked
Method: > Method:Result:
Methoded

Method: > Method:Result:

Original:

```
pattern Method: body
```

Transformed:

```
pattern Method: body
        Result: Expression
```

Figure 4

Structural patterns are represented internally as abstract syntax trees and can thus be manipulated by ASE. The 'original' and 'transformed' fields in figure 4 are in fact instances of ASE.

4.5 Direct Manipulation

ASE uses the pointing device as the major navigation and editing tool. Apart from textual and structural selection, the mouse is used to implement very fast shortcuts for common editing operations. Selecting program fragments and dragging them to another place is known as *drag & drop*. It can be used to replace and insert program fragments (the two most important structure synthesis operations). In particular identifiers can be typed once and dragged and dropped later on. Normally drag & drop has *copy behaviour*, that is, the replacement fragment or the inserted fragment is a copy of the dragged fragment. Sometimes *move behaviour* is more convenient: The dragged fragment is removed from its original position in the program and put elsewhere.

Drag & drop is also an implementation of cut/copy and paste (cut for move behaviour, copy for copy behaviour), using the whole program (and all structure editor windows on the desktop) as a random access multiitem clipboard.

From experience with the existing prototypes it has become clear that drag & drop is a very powerful tool. To the best of our knowledge[2], the Agora Structure Editor is currently the only structure-oriented editor featuring drag & drop.

4.6 Hybrid Editing

In our view a hybrid structure-oriented editor should support text editing of arbitrary program pieces. Text editing should be possible at all times and it should not put the editor in a mode. The user should be able to switch from text editing to structure editing and back at any time.

These design options pose some technical difficulties. When the attention is moved away from a text edited program piece, a parser must be invoked to generate the corresponding partial abstract syntax tree. To support hybrid editing without modes, the editor should not complain about erroneous program text. From a technical standpoint this means that more than one erroneous program piece can exist in the abstract syntax tree representing a program. Furthermore, these erroneous program pieces should be subject to structure-oriented editing operations as all other program fragments. To prevent that the abstract syntax tree turns into a tree only containing erroneous program pieces, the editor should merge text fragments into program text that can be parsed. How erroneous pieces should be merged is still under investigation at the time of writing.

5 Implementation Status

Several prototypes of the Agora Structure Editor exist. The first prototype (1992), implemented in Pascal, supports construction of complete Agora programs. It provides all editing operations presented in previous chapters, except structural transformation and general structural search & replace. It features drag & drop and textual editing of arbitrary program pieces. Presentation schemes can be modified and replacement and insertion menus can be adapted to the user's needs.

The current prototype is implemented in VisualWorks\Smalltalk. It is integrated in the programming environment for Agora. Now, the editor is completely independent of Agora. An editor is available for a target language if (1) a grammar class hierarchy for the target language is constructed, (2) those classes are connected with the structured grammar class hierarchy used by the editor classes, (3) a parser for the target language is provided.

The editor currently provides all structure-oriented editing operations, including structural transformation. Transformers can be added, removed or changed through the interface presented earlier. Due to minor technical problems involving dragging in VisualWorks\Smalltalk, drag & drop is not yet supported, but will be in the near future. The implementation of the hybrid aspect and the search facilities is currently ongoing.

[2]In [Minör 91] a direct manipulation interface for structure-oriented editors is suggested, but not implemented. Program fragments can only be dragged from a palette onto the program, not from within the program itself.

6 Conclusion and Future Work

Some design issues of the Agora Structure Editor have been presented. What really stands out against other work is the customisability of the editor. Replacement and insertion menus can be tailored to the user's needs and structural transformations can be defined interactively.
Since we embrace ergonomics as a principle, and other systems seem to neglect it, ASE differs strongly from other systems in that respect. The direct manipulation technique drag & drop is but one of the features that raise ergonomics to a higher level.
The editor has a hybrid nature. Text editing can be initiated at any moment, at any place in the program. No modes are introduced by the editor. The user can switch from text to structure editing and back at any time.
Due to the fact that the design is independent of the target language, the editor can also be considered generic. The independence relies on the application of structured grammars for internal representation of programs. The production rules of the target language's grammar are represented internally by instances of themselves, which are interpreted by the editor during editing.

In its current state, ASE can be used to implement other interface components. Currently ASE is being used for inspection purposes and will be used for debugging and browsing in the near future. Nevertheless work still needs to be carried out concerning multiple structural selection, full blown structural transformation, search, table driven parsing, merging erroneous text fragments and conditional program presentation.

Acknowledgements

I would like to thank Patrick Steyaert and Karel Driesen for reading drafts of this document.

References

[Allison 83]	Allison, L., *Syntax Directed Program Editing*, Software-Practice and Experience, Vol. 13, 453-465, 1983
[Cameron & Ito 84]	Cameron, R.D., Ito, M.R., *Grammar-Based Definition Of Metaprogramming Systems*, ACM Transactions on programming Languages and Systems, Vol. 6, No. 1, January 1984
[Centaur 92]	*Centaur 1.2 Manual*, September 1992
[De Hondt 93]	De Hondt, K., *A Customizable, Ergonomic, Hybrid Structure-Oriented Editor*, Master thesis Vrije Universiteit Brussel, August 1993
[Hendriks 91]	Hendriks, P.R.H., *Implementation of Modular Algebraic Specifications*, PhD thesis, University of Amsterdam, 1991
[Kaiser & al. 88]	Kaiser, G.E., Feiler, P.H., Jalili, F., Schlichter, J.H., *A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation*, Software—Practice and Experience, Vol. 18(8), 733-748, August 1988
[Koorn 92]	Koorn, J.W.C., *GSE : A Generic Text and Structure Editor*, Report P9202, University of Amsterdam, Programming Research Group, January 1992
[Minör 90]	Minör S., *On Structure-Oriented Editing*, PhD. Thesis, Department of Computer Science, Lund University, Sweden, 1990
[Minör 91]	Minör, S., *Interacting with Structure-Oriented Editors*, Technical report LU-CS-TR:91-74, Department of Computer Science, Lund University, Sweden, 1991
[Mjølner 90]	Mjølner Informatics ApS, *Sif - A Hyper Structure Editor, Users Guide*, Mjølner Informatics Report MIA-90-11(0.1), November 1990

[Mjølner 91]	The Mjølner Group, *Mjølner/Orm User's Guide (Version 1.3)*, Department of Computer Science, Lund University, Sweden, 1991
[Notkin & al. 83]	Notkin, D., Habermann, N., Ellison, R., Kaiser, G., Garlan, D., *Respons to Waters' article on structure-oriented editors*, SIGPLAN Notices, Vol. 18, No. 4, April 1983
[Shani 83]	Shani, U., *Should Program Editors not Abandon Text Oriented Commands?*, SIGPLAN Notices, Vol. 18, No. 1, January 1983
[Shneiderman 83]	Shneiderman, B., *Direct Manipulation: A Step Beyond Programming Languages*, IEEE Computer, August 1983
[Steyaert & al. 93]	Steyaert, P., Codenie, W., D'Hondt, T., De Hondt, K., Lucas, C., Van Limberghen, M., *Nested Mixin-Methods in Agora*, Proceedings of the ECOOP 93 Conference, July 1993
[Teitelbaum 80]	Teitelbaum, T., *The Cornell Program Syntheziser: A Tutorial Introduction*, Technical Report TR 79-381, Revised June 1980, Department of Computer Science, Cornell University, Ithaca, New York, 1980
[Teitelbaum & Reps 81]	Teitelbaum, T., Reps, T., *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*, Communications of the ACM, Vol. 24, No. 9, September 1981
[Toleman & al. 92]	Toleman, M.A., Welsh, J., Chapman, A.J., *An Empirical Investigation of Menu Design in Language-Based Editors*, Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Software Engineering Notes, Vol. 17, No. 5, December 1992
[van den Brand 92]	van den Brand, M.G.J., *Pregmatic, A Generator for Incremental Programming Environments*, PhD. Thesis, Catholic University of Nijmegen, the Netherlands, 1992
[Waters 82]	Waters, R.C., *Program Editors Should Not Abandon Text Oriented Commands*, SIGPLAN Notices, Vol. 17, No. 7, July 1982
[Zelkowitz 84]	Zelkowitz, M.V., *A Small Contribution to Editing With a Syntax Directed Editor*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 9, No. 3, May 1984