A Generic Synthesizer and its Implementation

Sten Minör

LU-CS-TR:88-32

Also published in *Proceedings of the 15th Simula User's Conference*, 1987

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

A Generic Synthesizer and its Implementation

Sten Minör

Department of Computer Science,
Lund Institute of Technology / Lund University,
Sweden

Abstract

This paper presents an architecture for a generic synthesizer. The synthesizer is generic in the sense that it interprets an arbitrary context free grammar. The interpretive approach facilitates incremental modifications to the grammar interactively. This can be used by the user of the synthesizer, and not only by the implementor which is the case in most systems, for modifications to the language that is edited and for presentation and interaction personalization.

The overall architecture is presented and compared with other synthesizer architectures. Some applications where such a synthesizer is applicable are discussed. The grammars and the implementation are described and an example of the user's view of the synthesizer is given. Finally some topics that may be interesting for further investigation are proposed.

Categories and subject descriptors (Computing Reviews 1986): D.2.2 Software Engineering: Tools and Techniques, D.2.4 Software Engineering: Programming Environments.

1. Introduction

Several synthesizers, or structure editors, for program development have been developed during the last years. In such an editor a program (or an arbitrary hierarchical structure) is *synthesized* interactively according to a language grammar in contrast to traditional programming environments where a program is edited as a sequence of characters which is *analyzed* by a parser building the corresponding abstract syntax tree. Synthesizers have the advantage of only allowing syntactically correct programs to be constructed by handling a program as blocks of language elements that are directly manipulated by the user.

Earlier synthesizers, such as the Cornell Synthesizer, were "hand-crafted" for one specific language. More recent systems allows generation of synthesizers automatically from a language description. ALOE[3], Pecan[4], and the Synthesizer

Generator[5] are examples of such systems. The evolution is similar to that of parsers. Earlier parsers were tailored for one language while most parsers today are generated by parser-generators from a language grammar.

The synthesizer described in this paper, SbyS - Structure by Structure synthesizer, facilitates editing of hierarchical structures according to a language grammar which is *interpreted* during editing. The grammar can hence be modified incrementally by the *user* during editing. In synthesizers generated by a synthesizer generator only the *implementor* has possibilities to modify the grammar in a batch-oriented manner. The only existing editor known to the author that allows incremental modifications to the grammar is DOSE[6], a prototyping environment for language design. DOSE provides two different environments, the users environment and the implementors environment. The users environment is used to manipulate a program and the implementors environment is used to modify the language description.

In SbyS no such distinction is made. By representing a grammar in the same way as a program, i.e. as an abstract syntax tree, the grammar may be manipulated in the same way as the program by interpreting a metagrammar. This possibility for the user to modify the language grammar incrementally in a uniform structured manner gives a high degree of interactivity and flexibility.

Figure 1 illustrates the differences between (a) a hand-crafted synthesizer tailored for one language, (b) a synthesizer generated from a language grammar, and (c) a grammar interpreting synthesizer which allows incremental modifications of the grammar.

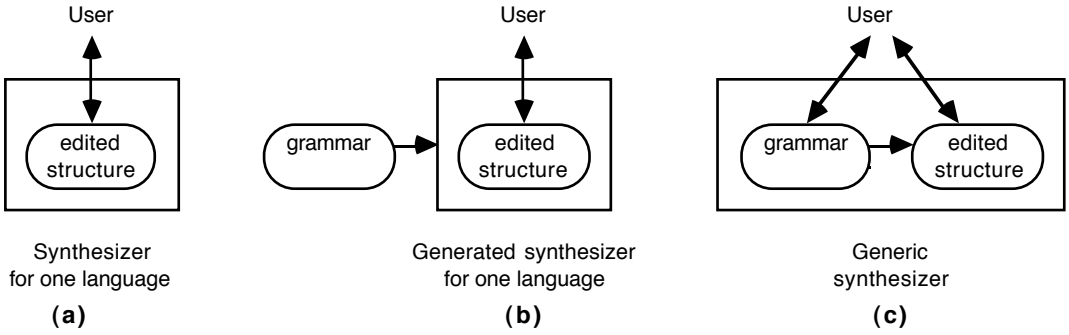

Figure 1
Different kinds of synthesizers

SbyS is here mainly discussed as a part of the Mjølner programming environment[11]. The Mjølner project is a joint Nordic project between three computer science departments and three industrial companies in Sweden, Denmark, and Norway. The environment will consist of a number of components sharing a common program representation. III[7], a predecessor to Mjølner developed at Lund Institute of Technology, is another example of such an environment where tasks like editing, static semantic analysis, code generation, debugging, etc. are handled by separate exchangeable components. SbyS may, however, be used for manipulation of arbitrary hierarchical structures as text documents or command languages.

The rest of this paper is organized as follows. Section 2 gives a broad outline of the architecture. Some examples of the applicability of such an editor is given in section 3. Section 4 describes the grammars used by SbyS. In section 5 a description of the prototype implementation is given. The architecture from the users point of view is shown in section 6, followed by future directions in section 7, and a summary in

section 8.

2. SbyS - a brief description

The basic functional unit in SbyS is the *synthesizer instance*. From the user's point of view it works similar to most synthesizers. The user is able to move in the edited structure, select sub-structures, expand incomplete nodes, and to cut , copy, and paste.A synthesizer instance is guided by language grammars and is thus totally language independent. The grammars are, in contrast to most synthesizers, interpreted. The interpretation involves searching in the grammars for appropriate productions and applying them during editing.

Grammars are separated into two kinds, *abstract grammars* which describes the logical structure of the language and *concrete grammars* which describes the physical presentation. In section 4 the different grammars are presented in more detail. Figure 2 shows a synthesizer instance with its *target structure* - the subject for editing - and its abstract and concrete grammars which are interpreted.

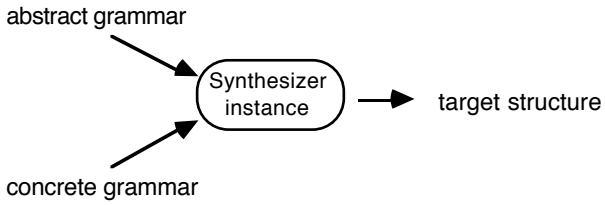

Figure 2
A synthesizer instance

Target structures and grammars are in SbyS represented in the same way, i.e. as abstract syntax trees. A consequence of this is that a grammar may be edited in the same way as the target structure by a synthesizer instance. It will the have a grammar as target structure and interprets grammars describing grammars , i.e. meta-grammars. By creating several synthesizer instances a target structure, its abstract, and its concrete grammars may be edited simultaneously. The grammars are shared between the synthesizer instances in a way that the target structure of one synthesizer instance serves as grammar to another. The different synthesizer instances are executing in parallel (as coroutines) forming a *configuration*. Figure 3 shows a configuration that allow simultaneous editing of a target structure and its abstract and concrete grammars.

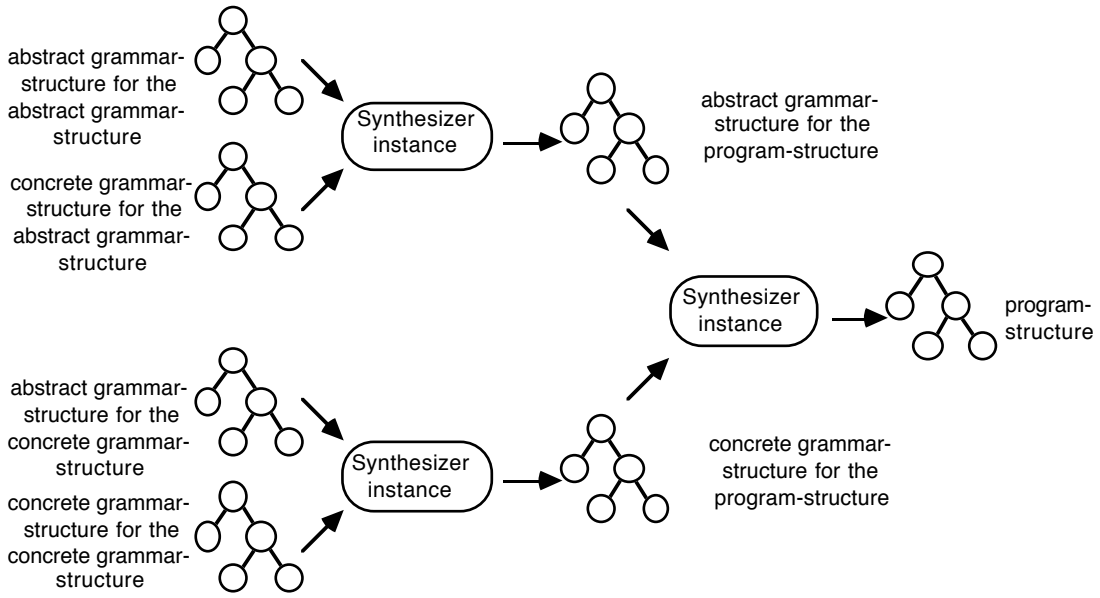

Figure 2
A configuration of synthesizer instances and grammar structures

The SbyS architecture implies that a target structure, a program for instance, can be edited simultaneously as its grammar is modified interactively. Editing of both programs, or other hierarchical structures, and grammars is done in a consistent way by different synthesizer instances.with equal behavior.

3. Application areas

The flexibility and uniformity of the presented architecture has some immediate advantages for the user of the editor. In contrast to traditional synthesizers the user, and not only the implementor, may to a large extent reconfigure the synthesizer interactively for his (her) specific needs or taste.

The concrete grammar specifies how the program is presented to the user. Different users often have different opinions about how a program should be formatted. In SbyS the user is not stuck to the form decided by the implementor, the concrete presentation may be modified according to the users taste.

By setting up a configuration with several synthesizer-instances editing the same target-structure, but with different grammars, different aspects of the target structure may be presented or edited. Multiple views with different grain of detail is one possible application. The user may specify the grammars for each needed view.

The interaction in a synthesizer is to a large extent depending on the design of the abstract grammar for a specific language. The problem is discussed in a paper on GNOME[8] where some hints for modeling the grammars are given. In SbyS the abstract grammar may be modelled interactively by the user in order to obtain a comfortable interaction.

Some languages, such as Lisp and Smalltalk, only have a rudimentary syntax. This is a strength because of the extensibility of the languages and a weakness because the poor readability and the extra demands put on the user in remembering the language constructs. A synthesizer supports the editor in presenting templates of the available

constructs which may be filled in step by step. In SbyS the readability may be enhanced by the possibility to define alternative concrete grammars. The style of programming in those languages includes extensive use of tool-boxes or surface languages. The user often utilizes a surface language for a specific application or as primitives in building another surface language at a higher level of abstraction. SbyS facilitates this style of programming as the grammars may be extended incrementally by the user along with the development of the surface language.

Finally SbyS is a fast vehicle for language development. The grammars may be developed and tested interactively with a very short turn-around time. This may be useful for development of application-languages, command-languages, or for other hierarchical structure that can be described by a context free grammar. SbyS has been used within our department in the design of specialized grammars to be used in the Mjølner project. It has turned out to be a flexible and fast tool for rapid prototyping and experimenting.

4. The grammar-structures

The grammar structures in SbyS are split into one abstract and one concrete part. One reason for this is that the two grammars describes two different aspects of a language. The abstract grammar describes the logical *structure* of the language and the concrete grammar describes the physical *presentation*.

The abstract grammar describes how a structure may be created or manipulated. It contains only the non-terminals of a traditional BNF grammar and is thus only describing the logical structure of the language without keywords and syntactical sugar. Abstract grammar structures in SbyS have an BNF-like form including productions describing constructions, alternatives, repetition, and lexical tokens. The production types are similar to those proposed by Cameron and Ito[9]. Figure 3 gives an example of productions in an abstract grammar. A construction production describes a syntactical element with a specified number of sub-elements of various types, e.g a block or an assignment. A repetition production states that a syntactic element may have an arbitrary number of subcomponents all of the same type. The statement-list is an example of a repetition production. An alternative production specifies that one of several syntactical categories may be selected. A statement for instance may be an assignment, a while statement, an if statement, etc. Finally, the lexem production describes terminal elements that contain texts, e.g. identifiers.

block ::= declaration-list & statement-list	Construction
statement-list ::= statement *	Repetition
statement ::=assign | while | if | procedure-call	Alternative
assign ::= id & expression	Construction
id ::= LEXEM	Lexem

Figure 3
Productions in an abstract grammar of some hypotetical language

The concrete grammar, or the unparsing scheme, contains keywords for the language together with formatting information. The notation for the concrete grammar is currently quite rudimentary but may be extended to a more extensive language like those suggested by Garlan[10].or Oppen[16]. Each production in the abstract grammar has a production in the concrete grammar describing the presentation of that construct. An exception is the alternation production. An alternation only specifies what syntactical categories are legal as sub-elements to a certain element and are thus

not presented to the user. A production in the concrete production consists of a template, different for each production type, that describes the presentation of a syntactical element . The template contains texts, indentation information, line-breaks, and references to the presentation of sub-components.

Figure 4 shows an example of a concrete grammar corresponding to the abstract grammar of figure 3. A block, for instance, is described by a construction production. It contains first the text and formatting information that shall precede the first sub-element. "..." denotes texts, >>> indentation, <<< end indentation, and <nl> new line.The @1 means that the first sub-element (the declaration list) shall presented. After that the text and formatting information that shall appear between the two sub-elements are stated followed by the second subelement and the text and formatting that conclude the template. Templates for repetition productions have a slightly different form. They consist of three parts: the *before*, *in*, and *after* parts. Each part consists of text and formatting information and are eventually empty which is denoted by an underscore. The before, in , and after parts describes what shall be presented prior to the first sub-element, between the following elements, and after the last element respectively.The template for a lexem production resembles the repetition template. It has only a before and an after part which describe the presentation before and after the lexem text.

```
block ::=                          Construction
□   "begin" >>> <nl>
    @1 ";" <nl>
    @2<<< <nl> "end"
statement-list::=                  Repetition
    before: _
    in: ";" <nl>
    after: _
assign ::=                         Construction
  @1
  ":= "  @2
id ::=                             Lexem
  before:_
  after: _
```

Figure 4
Productions in a concrete grammar of some hypothetical language

As the grammar-structure of an editor instance may be manipulated simultaneously as its target structure is edited cases of inconsistency may occur. The concrete grammars may be modified freely as they only affect the presentation but modifications to abstract grammars can cause trouble. Suppose that the abstract grammar-structure G is interpreted in editing the target structure T_G. If G is modified to G' the target structure will subsequently be edited as $T_{G'}$. $T_{G'}$ may, however, contain elements from G which are inconsistent with G'.

In the current implementation inconsistencies are ignored. The disadvantage of this is not too serious as many modifications, as extending the grammar, modifying parts of the grammar that are used in the actual target structure, and modeling the selection productions (which only affects the interaction), do not cause inconsistencies.

Identifying modifications to the grammar that potentially cause inconsistencies and transformations between grammar-structures and the corresponding target-structures might be an interesting area for further investigation.

5. Implementation

A prototype of the SbyS editor architecture has been implemented in Simula in order to demonstrate the ideas. The implementation will be enhanced and is intended to be an alternative editor-component in the Mjølner environment.. Figure 2 showed a simplified picture of SbyS. In this section a more elaborate description will be given.

The structures, or abstract syntax trees, representing programs and grammars are basically represented in the same way by a class AbstractSyntaxTree. This class contains methods for tree manipulation such as node insertion/removal, copying, file input/output, etc. A grammar is a subclass of AbstractSyntaxTree extending it with properties such as finding productions that describe a certain syntactic category and checking if one syntactic cathegory is derivable from another.

As a matter of fact a synthesizer instance in figure 2 consists of two distinct components described by two classes, namely the editor and the view. The editor class is responsible for the structured manipulations of the syntax tree according to the abstract grammar. It have properties such as expansion of an incomplete part of the tree, copying, cutting, and pasting. During these operations the editor is searching the abstract grammar for applicable productions in order to guarantee that only syntactically legal operations are performed. The view class is responsible for the presentation of the syntax tree according to the concrete grammar. Whenever the syntax tree is modified by the editor the view is notified for updating the screen representation incrementally. It also contains a hit function, i.e. a mapping from cursor positions to nodes in the syntax tree. To do this the view has a shadow representation which essentially is a linearization of the syntax tree together with information about screen positions and references back to the syntax tree.

The possibilities of dynamic instance creation in Simula is here of great use. Figure 2 showed a configuration with three editor-view instances, one for the target, one for the abstract grammar , and one for the concrete grammar. Many other configurations than that may be useful. If SbyS is used as a conventional synthesizer a simpler configuration with only one instance, the target instance, may be used. More complex configurations may be wanted in other situations. If several editor or view instances are created with the same target tree but with different abstract and concrete grammars the target tree may be viewed and manipulated in different ways. One program may, for instance, be simultaneously presented and edited with a different grain of detail. Different configurations are set up in a straightforward way in SbyS. The editor-view classes are instantiated dynamically and are connected to the appropriate instances of program or grammar trees. This can easily be done on request from the user.

The editor-view instances are executing in parallel as coroutines with other editor-view instances. In this way a program, different views of a program, and the grammars can be manipulated simultaneously. The internal state of each editor-view will be preserved and a consistent interaction style is obtained as grammars and programs are manipulated in exactly the same manner. Furtermore a modification to a grammar will come into effect instantly as the editor and view instances are interpreting the grammars on-the-fly.

An editor or a view may be specialized by subclassing. A program synthesizer may, for instance, be extended with an interpreter for a language, or a document synthesizer may be supplied with specialized properties for that purposes. The synthesizer may also be one component in a larger system, such as the Mjølner

environment. In this case it must be integrated in that system, i.e. be able to communicate with other components. Since SbyS is intended to be a generic synthesizer possible to use in a large variety of applications a general approach both for integration with other components and for specializations must be used. This is solved by letting the editors and views call virtual "notify" procedures whenever something important is done. The notify procedures are declared as dummies in the editor and view classes. A subclass can, however, redefine the notify procedures and perform appropriate actions.e.g. by notifying other components of the change. This mechanism is similar to "semantic actions" in parser-generators. The synthesizer notifies a subclass when a change occurs and the appropriate semantic action can be performed either by the subclass itself or by a call to some other component.

The response times are crucial in an interactive environment. It is important that the grammar interpretation is not considerably slowing down the interaction compared with an editor that is generated from a grammar. Using a grammar for Simula the time overhead seems to be small and the response-times are at a rough estimate fractions of a second. The dynamic search in the grammars do not slow down the interaction speed considerably. The shadow representation maintained by the views facilitates fast incremental screen updates but tends, however, to be space consuming. It also give rise to a considerable initialization time when the shadow-representation is built during the instantiation of a view. Finding a better time-space tradeoff that offers reasonable response times but with a smaller shadow representation is one important issue to make the synthesizer practically useful.

The implementation is done using Lund Software Simula on a Sun workstation. It comprises at present approximately 4000 lines of sparse source code.

6 A sample

In this section a sample of the user's view of SbyS is given using grammars for a hypothetical (toy) language. Figure 5 shows the screen layout for the same configuration as shown in figure 2.

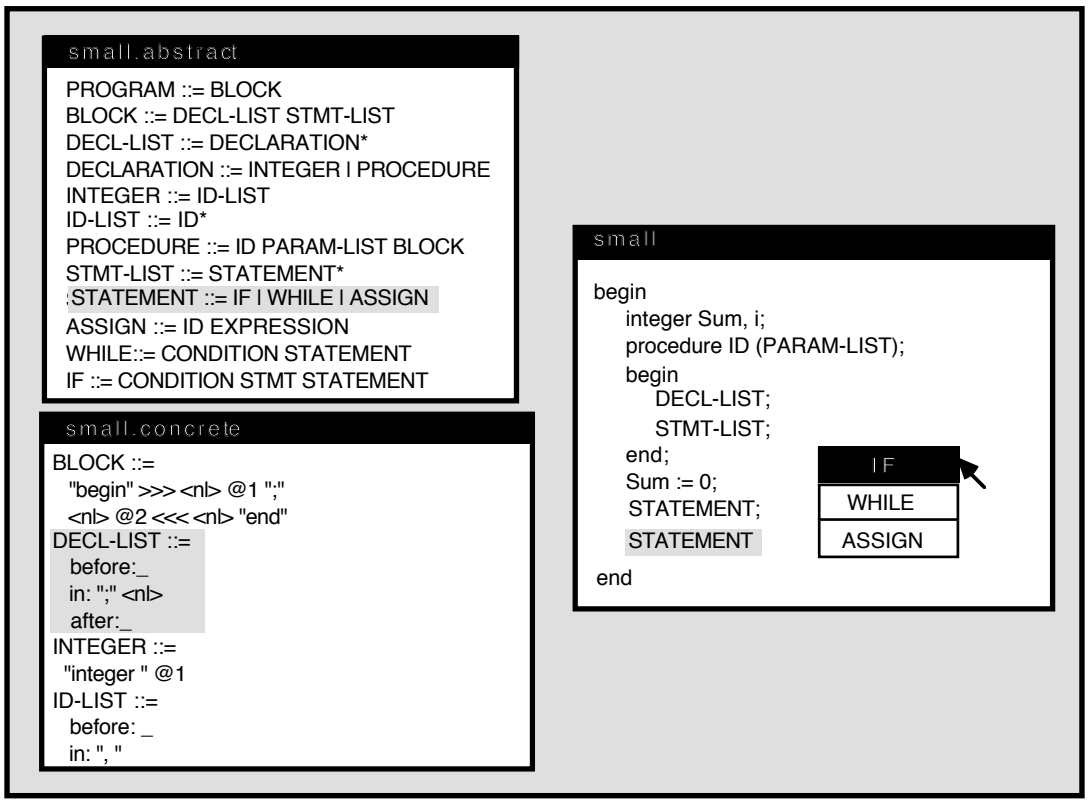

Figure 5
The user's view of SbyS

Each synthesizer instance has a corresponding window on the screen. One for the program structure "small", one for its abstract grammar "small.abstract", and one for the concrete grammar "small.concrete". The selected branch of the target structure is in each window marked with a structured cursor. The abstract grammar is presented in the usual BNF notation while the concrete grammar contains the keywords of the language together with formatting information. The metanode STATEMENT in the program window is expanded and the alternatives corresponding to the abstract grammar is presented to the user in a menu.

7. Future work

SbyS is only a prototype of a grammar-interpreting synthesizer. In order to obtain an editor that is robust and practically usable a lot of more or less straightforward work remains to be done. Some issues that may be interesting for further research are however discernible.

The editor described only handles the context-free properties of a structure or a program. The possibilities to describe context-sensitive properties using e.g. attribute grammars[5], action equations[14], or action routines[15], would be desirable. The favor of the possibility for the *user* to modify the context-sensitive properties of a language is most likely limited to language design activities as a programmer in general has no reason to change these properties of a language.

How to deal with inconsistencies between different versions of a grammar and corresponding target structures has to be further investigated. The inconsistency problem is two-fold. Some modifications to an abstract grammar may not cause inconsistencies, e.g. extensions to the grammar and rearranging the selection

productions. One subject is to identify modifications that may cause inconsistencies. The other is to take care of the inconsistencies and transform branches of target-structures that refer the modified parts of the grammar either automatically or according to some transformation rules.

The unparsing of a structure is currently quite rudimentary and solely textual. A more sophisticated concrete grammar with support for graphical presentation would enhance the presentation considerably. It would facilitate presentation of e.g. program-constructs and command-structures graphically. The latter may be an interesting area as command interaction today normally utilizes a variety of ad hoc techniques. Using a graphical grammar-interpreting synthesizer a general user-configurable framework for user interaction may be obtained.

8. Summary

An architecture for a grammar-interpreting synthesizer has been presented. The architecture is solely built by uniform editor and view instances, extensions to them, and structures representing grammars and programs. By this architecture a high degree of flexibility is achieved; the user may to a large extent reconfigure the editor in terms of the configuration of instances and structures, the interaction style, and the language for which it is used to manipulate. In most synthesizers this is only possible for the implementor. The synthesizer approach for editing has been used throughout the synthesizer both for editing programs and grammars. The user interface is thus uniform and comprehensible. The object oriented design and implementation facilitates straightforward extensions and integration with other tools in an environment.

Some examples of the applicability in the software engineering area, where the dynamic characteristics of the architecture may be useful, has been given. The approach may, however, be useful in many application areas where hierarchical structures are manipulated.

Acknowledgements

This work was supported by the Mjølner project funded partially by a grant from the Nordic Fund for Technology and Industrial Development. The author wish to thank the colleagues in the Mjølner project at Lund Institute of Technology for many valuable comments on the contents and style of this paper.

References

1. Teitelbaum, T., Reps, T., *The Cornell Program Synthesizer: A Syntax-Directed Programming Environment*, Communications of the ACM 24, 9, 1981

2. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., *Programming Environments Based on Structure Editors: The MENTOR Experience*, In Barstow et. al. (ed.) *Interactive Programming Environments*, McGraw-Hill, 1984

3. Medina-Mora, R., *Syntax-Directed Editing: Towards Integrated Programming Environments*, PhD thesis, Carnegie-Mellon University, 1982

4. Reiss, S., *Graphical Program Development with PECAN Program Development Systems*, Proceedings of the Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT/SIGPLAN, 1984

5. Reps, T., Teitelbaum, T., *The Synthesizer Generator*, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software

Development Environments, 1984

6. Feiler, P., Faimeh, J., Schlichter, J., *An Interactive Prototyping Environment for Language Design*, Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986

7. Magnusson, B., Minör, S., *III - an Integrated Interactive Incremental Programming Environment Based on Compilation*, Proceedings of the 1985 ACM SIGSMALL Symposium on Small Systems, 1985

8. Garlan, D., Miller, P., *GNOME: An Introductory Programming Environment Based on a Family of Structure Editors*, Proceedings of the ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984

9. Cameron, R., Ito, R., Grammar Based Definition of Metaprogramming Systems, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 1, 1984

10. Garlan, D., *The VIZ Unparse Specification Language and the UAL Unparser*, Dept. of Computer Science, Carnegie-Mellon University, 1985

11. Dahle, H-P., Lehrman Madsen, O., Löfgren, M., Magnusson, B., *Project Mølner - A Higly Efficient Programming Environment for Industrial Use,* Mjølner Project Report no. 1, Dept. of Computer Science, Lund Institute of Technology, Lund, Sweden, 1986

12. Birthwistle, G., Dahl, O-J., Myhrhaug, B., Nygaard, K., *Simula begin,* AUERBACH Publishers Inc., Philadelphia, USA, 1973

13. Goldberg, A., Robson, D., *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1985

14. Kaiser, G., *Semantics for Structure Editing Environments,* Dept. of Computer Science, Carnegie-Mellon University, 1985

15. Feiler, P., *A Language Oriented Interactive Programming Environment Based on Compilation Technology,* Dept. of Computer Science, Carnegie-Mellon University, 1982

16 Oppen, D.C., *Prettyprinting,* ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, 1980