UDC 004.4'4

A Structure Editor for the Program Composing Assistant

Zorica Suvajdžin, Miroslav Hajduković

Faculty of Technical Sciences, Computing and Control Department,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia and Montenegro
tweety@uns.ns.ac.yu, hajduk@uns.ns.ac.yu

Abstract. The Program Composing Assistant is an interactive generic development environment dedicated to programming languages. It provides a structure editor with graphical user interface as a main feature. The structure editor is based on an intuitive approach, and aims to integrate important practical aspects of structure editing.

1. Introduction

There is a long history of research on ways to enhance the quality of software development and increase the productivity of developers [30], [10], [11]. One result of this research is a variety of different software development environments with (language-based) structure editors as a component. Despite all this research, there is no structure editor in widespread, every day use. Therefore, there is a need for more effort in this direction. The ***PRO****GRAM* ***COMP****OSING* ***ASS****ISTANT* (PROCOMPASS) is a programming environment originating from one such effort. A central component of this environment is a structure editor. The *PRO-COMPASS* editor is described in this paper.

The PROCOMPASS editor offers a unified user interface aimed at supporting structured but still flexible syntax-oriented editing of program text with equal treatment of all programming language elements (e.g. expressions and statements). The consequences of using the editor are (1) no need of a full command of the programming language syntax, (2) the opportunity for serious reduction of syntactic (and semantic) errors and (3) an increase in the programmer's productivity. The first two points are probably more important for beginner and non-professional programmers; the third one is important for professional programmers.

Section 2 of the paper introduces structure editors, and discusses related work. Then, section 3 shows examples of the usage of the PROCOMPASS editor. Section 4 gives features of the PROCOMPASS editor. Its internal representation, implementation and specification of the input follow next. The conclusion discusses distinguishing characteristics of the PROCOMPASS editor.

2. Overview and Related Work

From the standpoint of syntax-oriented editing of program code, there are three kinds of editors depending on the way a user interacts with them. If a user changes the text, which is then parsed by an external tool to derive the corresponding syntax tree, the editor is said to be a *text editor*. If a user changes the tree, which is then pretty-printed to derive the corresponding text, the editor is a *structure editor*. If the user is allowed to change either the text or the tree, the editor is said to be a *hybrid editor* [4], [19].

Structure-oriented program editing environments support the concept of direct structure manipulation. The user interacts directly with program constructs and avoids the tedium of remembering the details of the syntax. While program text is displayed on the screen, the user directly modifies the underlying structure.

A structure editor provides editing operations only on structural elements and does not permit the user to construct syntactically incorrect programs. A problem with several structure-based editors is that they force a top-down approach to entering text, corresponding to a pre-order traversal of the abstract syntax tree. This inhibits a natural mode of entering text and can make certain changes difficult. For example, from the earliest days of structure editors, users have complained about awkwardness of entering expressions with infix operators. To illustrate this, consider the expression (example taken from [26])

```
a * b + c * d
```

It takes just 7 insertions actions, each invoked by a single keystroke, to enter this expression in an ordinary text editor. In a simple-minded structure editor, these elementary insertions are interspersed with tree navigation commands. The keystroke sequence might look something like this:

```
+ ↓ * ↓ A → B ↑ → * ↓ C → D
```

(where ↓, → and ↑ respectively navigate downwards to the first child, rightwards to the next sibling, and upwards to the parent).

The usual solution to the awkwardness of making modifications while respecting the structure of the abstract syntax tree is a hybrid approach, in which the editor supports both structure-based and unstructured operations. The user enters program fragments as text and asks the environment to complete the processing as far as possible. Using incremental parsing techniques, the environment converts the text fragment into a program structure. In most hybrid systems, the user can switch between two edit modes: structured and unstructured. These two modes are radically different, and in the unstructured mode all advantages of the structured approach are lost. Another approach is to let the choice between the two modes be determined automatically by the grammatical type of the portion of text; below a certain level the tree nodes consist of unstructured text. The hybrid approach, in both forms, violates the requirement of modelessness.

A structure editor is an inevitable component, and usually the main feature, of program editing environments that are automatically generated from formal language specifications. Most such systems use abstract syntax trees as the internal storage format for programs. The way programs can be entered depends on the kind of editor [31].

The Mentor and Centaur [5] systems generate environments with a structure editor. The user is also allowed to edit textually, by selecting a subtree in the structure editor and invoking a "text-edit" command. After editing the text, the user has to invoke a "parse" command which parses the changed text and replaces the selected subtree.

The successor of the Cornell Program Synthesizer, the Synthesizer Generator [28], is probably the most widespread system for generating programming environments. Generated environments include hybrid editors, in which, switching from text editing to structure editing or vice versa is implicit. More than one textual selection within the same editor is allowed. Which language constructs can be edited in what mode (i.e., textually, structurally, or both) is defined by editing rules.

Another system aiming at the generation of programming environments is PSG (Programming System Generator) [3]. PSG generated editors are of the hybrid kind. Switching from structure mode to text mode is implicit, but the reverse is explicit. There can be more than one textual selection within the same editor.

What differentiates the ASF + SDF Meta-environment (Algebraic Specification Formalism plus Syntax Definition Formalism) [6], [7] from other systems is the fact that the same editor is used both for editing language definitions and for editing programs. Its editors are of the hybrid kind.

Pan [4] is an editing and browsing system. Pan tried to find the middle ground in the lexical representation between a simple user model that supports pure textual editing and a rich structural representation that supports structure editing. It permits unrestricted text editing, performs full incremental language analysis on demand and provides feedback.

SmartTools [2], [23] is a semantic framework generator. Given extended abstract syntax (AST) definitions of a language, SmartTools can automatically generate a structured editor specific to the language. SmartTools can build and display one or more views of the program.

There are many other systems and editors, like Gem-Mex [1], GSE (Generic Syntax-directed Editor) [18], ASE (Agora Structure Editor) [14] and the ABC structure editor [22]. The concept of structure editing is applicable to both program editors and graphic editors [25], [29].

Although the work reviewed above show that structure editors are a subject of interest, there is no structure editor at the moment that is widely spread and applied practically. There are some text-based programming environments in contemporary use (for example, Eclipse [32]) that offer some sort of structure-like editing, but only to a small extent.

We have evaluated several structure-oriented editors [5], [18], [28], [14], [23], [22]. The evaluation was not an easy task, for two reasons. First, implementations of these editors are not made public and are not executable

on every platform. Second, many documents describe the editor design, but fail to describe the user interface (how the editor looks and feels on the outside). The evaluation has been based on several reports for each editor. From the evaluation it is clear that the design of present structure-oriented editors leaves room for improvements; our major objective was to tackle the problems suffered by existing structure-oriented editors.

Therefore, the PROCOMPASS structure editor follows an intuitive approach. It aims to include the features from traditional structure editors that are of clear benefit to the programmer. It is designed to be purely structural, which means it does not have separate structural and textual editing modes: textual editing is used only for entering lexical structures, and is smoothly integrated into structure editing. Editing of each structure follows the same principle. This means that expressions can be edited as easily as any other program structure. PROCOMPASS is designed to offer a readily accessible everyday tool for the ordinary user and, at least at the beginning, is targeted at novice and non-professional programmers.

3. PROCOMPASS Editing

The PROCOMPASS editor enables a user to form a complex structure by putting together simpler components. Composing the complex structure from simpler components is basically done by combining some of the following possible actions:

1. moving through the structure to mark an existing component,
2. inserting a new component (before or after the marked one),
3. modifying the marked component,
4. removing the marked component, or
5. structural transformation (refactoring) of the marked component.

The composition process is a recursive one, as each individual component can be a complex structure, composed of simpler components.

For example, a C variable definition statement consists of the variable type, the variable name and a possible variable initial value in the form of expression:

```
int a = 0;
```

Such statements may be represented as a template. The C variable definition template contains fields for the variable type, the variable name and the variable initial value. The first two template fields are mandatory and the third is optional. A template is empty if all of its fields are empty (containing question marks). Filling the fields of a template is done by substituting question marks with the suitable content. A template is filled when all of its fields are filled. A template is half-full when at least one of its fields is empty (contains a question mark).

The composition process is restricted by the syntactic and semantic rules of the language. The PROCOMPASS editor helps in applying these rules by

offering a subset of possible operations at each step of a composition process and by accepting only operations from the offered subset. The subset of permitted operations is context dependent. For example, insertion after a marked component is allowed only when that will not affect the correctness of the entire structure. The same applies to removing the marked component. Modifying the marked component is dependent both on the context and on the component itself — different components allow different kinds of modifications. The composition process rules depend on the structure being formed.

The following text contains a few simple examples with the intention of emphasizing the way of interaction between the user and the editor (more complex example(s) would probably blur the idea to some extent).

Operations applicable to a template are *insertion*, *modification*, *deletion* and *structural transformation*. In the following text, the first three operations are shown through examples.

Suppose that a variable definition template is filled with the variable type `int` and the variable name `a`, and that the optional field denoting the variable's initial value is not present. If the field denoting the variable name is selected, pressing the *insert* key starts the insertion operation that adds the optional terminal field of the variable's initial value (Fig. 1).

Fig. 1. Insertion example

At this point we can demonstrate the modification operation in the content of the terminal field of the variable's initial value (Fig. 2). This operation is invoked when the user presses the *enter* key. The modification operation starts by showing the lexical dialog, intended to accept text from the user. The text must match the regular expression attached to this field. If the matching is successful, the entered text becomes the new content of this field.

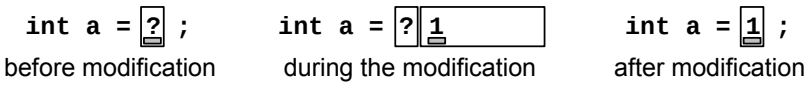

Fig. 2. Modification example

To delete the variable's initial value field, the user presses the *delete* key (Fig. 3).

Fig. 3. The first deletion example

Applying the deletion operation on the variable name component will not have the same effect as on the variable initial value component (Fig. 4)

because the deletion operation is context dependent, and has different meanings when applied to an optional field and to a mandatory field. The variable initial value field was completely removed because it was optional, but the variable name field is mandatory and the deletion operation will remove only its content.

Fig. 4. The second deletion example

The continuing example shows that expression editing is as easy as editing of any other program structure. The example assumes that the grammar defines an initial value as an expression that can contain one of the four most commonly used binary operators (+, -, * and /).

The insertion operation on a marked terminal field of the variable's initial value adds two fields: the first one for an operator and the second one for the second operand (Fig. 5). After insertion the cursor automatically moves to the next field to be edited, that is, the operator field.

Fig. 5. The expression editing example

The modification operation at this point (Fig. 6) shows the semantic dialog (which hides the second operand field). The dialog has two parts: the lower part contains all the possible options (all applicable operators, in this example) that the marked field can contain, and the upper part accepts text from a user in order to choose one of the options showed in the lower part. The selected alternative becomes the new content of this field. After modification, the cursor automatically moves to the next field to be edited, that is, the second operand field. Then, the content of this field is modified.

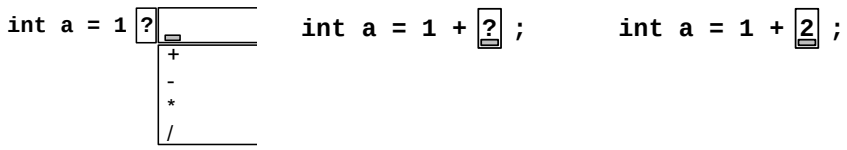

Fig. 6. The expression editing example (continued)

A structural transformation is a relation between two structural templates. One is a source template that specifies what program fragment the transformation is applicable to; the other is a destination template that

specifies what the source looks like after transformation. Transformation is done by replacing the marked source template by the destination template in a way that contents of some fields of the source template are used as contents of the fields of the destination template. For example, an `if` statement (source template) can be transformed into a `while` statement (destination template). It is done by replacing the `if` statement with a fresh `while` template that contains a condition field and body statements from the `if` statement.

Structural transformation is user-defined. A simple (meta) language is provided to describe structural transformation. This language contains rules. One such rule contains information about source and destination structure, and the relationship between their substructures.

4. PROCOMPASS Guiding Design Principles

The main goal of the PROCOMPASS structure editor was to increase users' productivity without giving up the user friendly interface. To achieve this, editor needs to offer a wide set of editing services, high quality visualization, and customizability [12].

The PROCOMPASS structure editor implemented all of the proven interaction paradigms that are used in traditional structure editors, such as:

1. avoiding editing errors (syntax and static semantic),
2. guiding a user, so a user does not need to be fully familiar with the syntax of the programming language,
3. enabling navigation (positioning and selection) by mouse as well as by keyboard; selection is implicit and position sensitive,
4. providing automatic indentation, and syntax highlighting,
5. supporting comments,
6. allowing semantically sensitive variable renaming.

Analysis of the traditional structure editors has enriched the PROCOMPASS structure editor features. A necessary characteristic of the structure editor is smooth integration of text and structure editing, without having to switch between two radically different modes. Another very important characteristic of the PROCOMPASS editor is complete language-independency. This starting idea enabled the implementation of a uniform and consistent user interface, so that, the way of handling each program structure follows the same principles. From the perspective of the user, the most important characteristic of the editor is that it is easy to learn with an intuitive user interface [21]. The majority of programmers have their own editing habits and they are often not willing to change them. They will only accept a tool that offers more advanced editing services for comfortable use, that is intuitive enough without special training. The PROCOMPASS editor tries to strike a balance between user and tool.

In PROCOMPASS the editor allows components to be filled from left to right; in the way the text is expected on the screen (not top-down, as some

structure editors request). Expression handling is easy, with automatic bracket matching. Editing time is significantly reduced as the result of the following: names are assigned only once, and then selected; keywords are automatically inserted into the program text; program text marking is automatic and the results of the program text modifications are automatically propagated down the rest of the program text.

The high quality visual design has a major role in user's comprehension of programs (documents). Textual display, generated by a pretty-printer [8], [9], [17], is enriched with additional information using typographical styles, which are specified by font and color characteristics.

Finally, an effective structure editor must be customizable in order to accommodate the variations among individual users. The PROCOMPASS editor is easily configured for different languages and tasks (fonts, colors, shortcuts to common editing operations, etc.).

An important aspect of any program editor is support for code maintenance. For this purpose, the PROCOMPASS editor offers structural transformation. This operation converts one component into another component. For example, it can convert `if` statement into `while` statement, convert an expression of one type into an expression of another type (cast), encapsulate a set of statements into a function body, split a function into several statement blocks, etc. The structural transformation is context dependent, so it offers a list of all refactoring alternatives available at the selected point of a program text. The list of refactoring alternatives is stipulated in the input specification of the editor.

Code maintenance operations also include copy and paste operations. The Copy operation copies a selected structure into a clipboard, and the paste operation pastes a component from the clipboard at the selected point of the program text. This operation is context dependent, so it shows only a part of the clipboard that is allowed to be pasted at the selected point of a program text – and the user will choose what is to be pasted (for example, this includes pasting all statements, or statements selected from all statements).

5. Discussion of Implementation and Specification

The internal representation of program text components in PROCOMPASS has a hierarchical structure. Terminal components (corresponding to template fields which contain the text of identifiers, numbers, etc) are at the bottom of the hierarchical structure. Higher levels contain nonterminal components, composed of other nonterminal or terminal components. The hierarchical structure means that the internal representation (implementation) of the structure is a graph in which each node corresponds to a component. An example of a graph for a C variable definition statement is shown in Fig. 7.

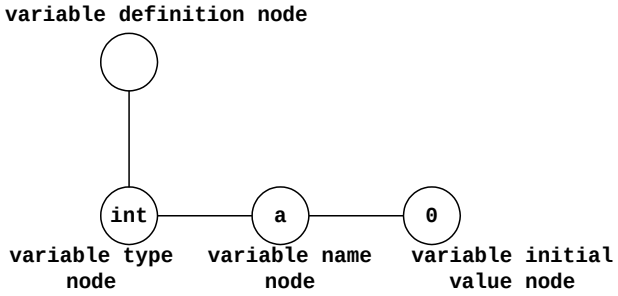

Fig. 7. Example of the graph for a C variable definition statement `int a = 0;`

Operations for program text template manipulation are associated with the graph and enable graph manipulation. For example, an insertion operation adds a new component after the marked one. That is, a new node with its subgraph is added after the marked node on the same level of the graph hierarchy. Some operations associated with individual nodes are specific to the kind of node used. For example, if a delete operation is applied to a mandatory terminal component, the node will remain, and only the content of the node will be deleted, but if it is applied to an optional terminal component, the node and content (or its subgraph) will be removed.

The specialization of the PROCOMPASS editor to a programming language is done by a formal language specification that can either be written in XML or in a format similar to the Lex [20] and Yacc [16] language specification. It contains both syntactic and semantic definitions of the programming language, including definitions of regular expressions, keywords, other lexical definitions (including the form of comments), data types, operators, name accessibility, the start symbol of the grammar, and grammar rules (productions). In addition, it includes pretty-printing information (colors, indentation, etc.) and rules describing structural transformations.

Each grammar symbol and each grammar rule may have associated semantic information [13]. The semantic information is described using attributes. Attributes are logical names (flags) that are attached to a symbol. A symbol can have default attributes, and can also obtain attributes from a context.

As mentioned earlier, the internal representation of the editing structure is a graph. Each (grammar) symbol corresponds to a node, described by the `Node` class. This class contains attributes intended to describe the specialities of a corresponding component, and common methods for manipulating the graph. The specialities of each component are extracted from the formal language specification, together with information about the symbol's semantics and component forming.

The production rules of the grammar are presented internally by instances of themselves. Therefore, the editor interprets the production rules during editing operations. As a consequence, one can plug-in an arbitrary grammar

in order to utilize the editor for writing programs in the corresponding language. Due to this, it can be said that the editor is generic.

Two prototypes of the concept described in this paper exist. The first prototype is the Structure Text programming language editor [27] (IEC 1131-3 [15]). The implemented prototype is not generic, but is for the ST language only. It provides the structure-oriented operations presented in previous sections, except for structural transformations and the copy and paste operations. The second prototype is generic, and supports all structure-oriented operations except any structural transformation that is currently ongoing. For testing purposes, the C programming language and a subset of the Java programming language are used.

6. Conclusion

The major objective of this research was to tackle the problems that existing structure-oriented editors suffer from. The result of the research is the PROCOMPASS structure editor. It includes the features of traditional structure editors that have proven productive and advantageous, and extended this set with features maintaining a user-friendly interface.

The PROCOMPASS editor is a generic (designed to be target language independent), pure structure-oriented (modeless) and a user-friendly (intuitive interface) program editor. It provides structural transformation as very powerful editing operation and code maintenance support.

Future work to be carried out includes multiple structure selection, more powerful structural transformation, more powerful search operations, and graphic editors.

Acknowledgement

We are grateful to a reviewer for helpful comments on earlier drafts of this paper and for constructive suggestions.

7. References

1. Matthias Anlauff, Philipp W. Kutter, Alfonso Pierantonio, Lothar Thiele, Generating an Action Notation Environment from Montages Descriptions, Proceedings of the 2nd International Workshop on Action Semantics (AS'99), BRICS Notes Series, Number NS-99-3, pp. 1-42. (March 1999)
2. Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier, SmartTools: a Generator of Interactive Environment Tools, Electronic Notes in Theoretical Computer Science, Vol. 44 (2) (2001). Available: Available: http://www1.elsevier.com/gej-ng/31/29/23/73/27/48/44_2_015.pdf

3. R. Bahlke and G. Snelting, The PSG system: From formal language definitions to interactive programming environments, ACM Transactions on Programming Languages and Systems, Vol. 8, No. 4, pages 547-576. (1986)
4. Robert A. Ballance, Susan L. Graham, Michael L. Van De Vanter, The Pan Language-Based Editing System, ACM Transactions on Software Engineering and Methodology. (1991)
5. P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, Centaur: the system, ACM SIGPLAN Notices, 24(2):14-24, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. (1989)
6. M.G.J. van den Brand, T. Kuipers, L. Moonen and P. Olivier, Implementation of a prototype for the new ASF+SDF Meta-Environment, Electronic Workshops in Computing, 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Amsterdam. (1997)
7. M.G.J. van den Brand and P. Klint, ASF+SDF Meta-Environment User Manual, Revision 1:134, September. (2003) Available: http://www.cwi.nl/projects/MetaEnv/meta/doc/asfsdfmanual/user-manual_0.html
8. M.G.J. van den Brand, M. de Jonge , Pretty-Printing within the ASF+SDF Meta-Environment: a generic approach , Technical report SEN-R9904, CWI, Department of Software Engineering. (March 1999)
9. M.G.J. van den Brand, E. Visser, Generation of Formatters for Context-free Languages, Programming Research Group, University of Amsterdam. (1995)
10. Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann (edited by Peter Fritzson), Overview of Software Development Environments, (1992) Available: http://www.ida.liu.se/~petfr/princprog/envpaper.pdf
11. Alan Dearle, Michael Oudshoorn, Karen Wyrwas, An Integrated Approach to the Generation of Environments from Formal Specifications, Australian Computer Science Communications, Volume 16, Number 1, pp 217-228. (February 1994)
12. Miroslav Hajduković, Zorica Suvajdžin, Žarko Živanov, Character oriented program editing - habit or necessity, Novi Sad Journal of Mathematics, Volume 33, Number 1. (2003)
13. Jan Heering, Paul Klint, Semantics of Programming Languages:A Tool-Oriented Approach, ACM Sigplan Notices, 35(3):39-48. (Mart 2000)
14. Koen De Hondt, Proceedings of the 1994 Groningen Student Conference on Computer Science, number CS-N9401 in Computing Science Notes, pages 27-35, (1994)
15. International Standard IEC 1131-3, Programmable Controllers – Part 3: Programming Languages, IEC. (1993)
16. S.C. Johnson, YACC: yet another compiler-compiler, Bell Laboratories, unix programmer's suplementary documents, volume 1. (1986)
17. M. de Jonge, A pretty-printer for every occasion, Technical report SEN-R0115, CWI, Department of Software Engineering. (May 2001)
18. J.W.C. Koorn, GSE: a generic text and structure editor, In J.L.G. Dietz, editor, Conference Proceedings of Computing Science in the Netherlands, CSN'92, pages 168-177. SION. (1992). Appeared as Report P9202, University of Amsterdam. Available by ftp from ftp.cwi.nl:/pub/gipe as Koo92b.ps.Z
19. J.W.C. Koorn, H.C.N. Bakker, Building an editor from existing components: an exercise in software re-use, In J.L.G. Dietz, editor, Conference Proceedings of Computing Science in the Netherlands, CSN'92, pages 168-177. SION. (1992). Appeared as Report P9202, University of Amsterdam. Available by ftp from ftp.cwi.nl:/pub/gipe as Koo92b.ps.Z

20. M.E. Lesk and E. Schmidt, LEX – A lexical analyzer generator, Bell Laboratories, unix programmer's suplementary documents, volume 1 (ps1) edition. (1986)
21. Linda McIver, Evaluating Languages and Environments for Novice Programmers, In J. Kuljis, L. Baldwin & R. Scoble (Eds). Proc. PPIG 14, Brunel University. (2002)
22. Lambert Meertens, Steven Pemberton and Guido van Rossum, The ABC Structure Editor: Structure-based editing for the ABC programming environment, Tech. Report CS-R9256. (1992)
23. Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau, Claude Pasquier, Joel Fillon, Christophe Held, Isabelle Attali, Aspect and XML-oriented Semantic Framework Generator: SmartTools, In Second Workshop on Language Descriptions, Tools and Applications, LDTA'02. ETAPS'2002, Electronic Notes in Theoretical Computer Science 65 No. 3. (2002). Available: http://www.elsevier.nl/locate/entcs/volume65.html
24. Vincent Quint and Irene Vatton, Making Structured Documents Active, Electronic Publishing - Origination, Dissemination and Design, Vol. 7(2), 55 -74. (June 1994)
25. J. Rekers, On the use of Graph Grammars for defining the Syntax of Graphical Languages, Technical Report tr94-11, Leiden University, Dep. of Computer Science. (1994)
26. Bernard Sufrin and Oege de Moor, Modeless Structure Editing, In: J. Davies, A. W. Roscoe and J.C.P. Woodcock (editors), Proceedings of the OxfordMicrosoft symposium in Celebration of the work of Tony Hoare, September 13-15, 1999.
27. Zorica Suvajdžin, Structured syntax driven editor for ST programming language, master thesis, Faculty of Engineering, Department of Computer Science, University of Novi Sad. (2000)
28. Tim Teitelbaum and Thomas Reps, The Cornell program synthesizer: a syntax-directed programming environment, Communications of the ACM, Volume 24, Issue 9, pages: 563 – 573. (September 1981)
29. S.M. Uskudarli, T.B. Dinesh, The VAS Formalism in VASE, Technical Report University of Amsterdam. (1996)
30. Michael L. Van De Vanter, Practical Language-Based Editing For Software Engineers, volume 896 of Lecture Notes in Computer Science, pages 251-267. Springer-Verlag. (1995)
31. Michael L. Van De Vanter and Marat Boshernitsan, Displaying and Editing Source Code in Software Engineering Environments, Second International Symposium on Constructing Software Engineering Tools (CoSET'2000). (2000)
32. www.eclipse.org

Zorica Suvajdžin has graduated at the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad, in 1998. She got Master Degree in 2000 from the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad. She is currently a teaching assistant in Computer Science courses at the Faculty of Technical Sciences, University of Novi Sad.

Miroslav Hajduković has graduated at the Faculty of Electrical Engineering, University of Sarajevo, in 1977. He got Master Degree in 1980 and his Ph. D. in 1984 from the Faculty of Electrical Engineering, University of Sarajevo. In 1985 he was on post-doctoral studies in Computer Laboratory at the Cambridge University in Great Britain. He is currently a Professor of Computer Science at the Faculty of Technical Sciences, Computing and Control Department, University of Novi Sad.