Syntax-directed Editing Environments: Issues and Features

Amir Ali Khwaja
Joseph E. Urban

Arizona State University

Abstract

Syntax-directed editing environments are designed for the development and maintenance of formal documents such as specifications, computer programs, and structured software documentation. The purpose of such environments is to provide a variety of facilities for the construction of the formal documents in an integrated and easy to use manner. Based on this premise, the paper identifies the usability and design of syntax-directed editing environments as critical issues and investigates the factors contributing to these issues. A number of features are formulated for the design and/or the use of such environments. Some of the existing editing environments are analyzed using the formulated features and the result of the analysis is presented in a tabular fashion.

Keywords and Phrases: Syntax-directed editors, support environments, usability, design.

Introduction

Computer languages have highly structured syntactic forms and hence can not be treated as mere text composed of strings of characters [3, 8, 29]. A text editor that acknowledges this structure and uses this awareness to the creation and alteration of programs is termed as a syntax-directed editor. Hence, a syntax editor has the knowledge about the language syntax and can detect syntax errors. Interactive languages, such as BASIC, have been successful for the most part due to the editing of the program within the language environment [4, 26]. The immediate feedback for each statement entered is helpful in the creation of a program and reduces the effort of detecting and correcting errors after the program is constructed. Moreover, such a development paradigm encourages software development in an integrated fashion. In such an environment the output of one tool is automatically fed to other tool(s) without any human interventions, contributing to increased productivity. The absence of declarations in languages like Lisp made possible incremental program development and resulted in interactive programming environments such as Interlisp [30]. This capability of immediate feedback on the syntax of the edited statement is possible in block-structured and strongly-typed languages [2].

A syntax-directed editor can be viewed as a device with an internal memory and an internal database [3]. The editor manipulates concrete programs as abstract objects by performing language-dependent operations on the internal representation without the awareness of the users [8]. The language-dependent operations are stored in the internal database. This capability of the editor helps developing reliable programs by preventing syntax and static semantic errors and plays a key role in a programming development environment. Sandewall [27] identified a specialized editor, capable of at least understanding the syntax of the chosen programming language, as an integral part of an interactive programming system.

Active research in the area of syntax-directed editing environments has resulted in a number of such environments for a variety of computer languages. This paper puts together a number of issues regarding syntax-directed editing environments. These issues are decomposed in two categories: usability and design issues. The paper further identifies nine features that can be considered in the development and/or use of such systems. Finally, an overview of seventeen editing environments is presented in the context of the identified features.

Usability Issues

The success of any system with its users depends on what type of interaction the system provides for its users. Communication between two entities calls for consideration of the strengths and weaknesses of both entities. In light of current discussion, the two entities are a language-specific editor and a user of such an editor. The usability issues are important and, as indicated by Nielsen [22], do not just appear because they are desired, instead they have to be considered in the design process. A user should be aware of what a language-specific editor can do and where it falls short of providing desired features. On the other hand, a language editor should be designed with consideration of its users. For the purpose of practicality, it is better to design editors for experienced users than for novices. The reason is that novices have to learn a particular language and a conventional text editor anyway. They can learn a language-based editor and benefit from its features in

ACM-SAC '93/2/93/IN, USA
© 1993 ACM 0-89791-568-2/93/0002/0230...$1.50

building programs and other documents. Some issues were identified from the research and experience of several language-specific editors. These issues are summarized in the next four sections.

Unacceptibility by the User Community

Program editors, or more generally, language-based editors are still striving for recognition from the user community even after more than two decades. Claims by the user community about the restrictiveness, inflexibility, and inefficiency of these editors can only partly be credited. Researchers and developers ignored a number of features by not consulting the actual users of such editors, but these pioneering efforts have provided a strong base and good experience for building practical editing environments. Recent research has tried to overcome some of these weaknesses [3, 21, 32]. Most of these claims are the result of the same reaction displayed when structured programming was introduced to replace the more free, flexible, and fallible goto style of programming. Of course, such a reaction can be expected if restrictions and limitations are imposed on the way something is carried out.

The construction of a system, or development of any product in general, requires planning, methods, structure, and effort. For the same reason the software community tried to "structure" the otherwise unstructured, and hence error prone, programming discipline some twenty years ago. Likewise, some kind of structure is needed to be imposed on the actual creation of software and other formal documents. The writer of such documents should be allowed to follow certain principles. These principles are not new and are actually part of the language. The only difference is that these principles are now being enforced through the language-specific editors. As pointed out by Hansen [17], that even though systems are designed to fit the users, some interactive systems will require user retraining and even alteration of thinking habits of many years standing. Adoption of new technology causes a disruption in the normal course of achieving objectives and users do not like strange ideas even if they seem useful [7]. This change in mentality is difficult but important to benefit from the new technology.

Affect of Language Definition

The number of computer applications are increasing rapidly, resulting in large and complex software. New concepts such as modularization, abstraction, and encapsulation are now being introduced in computer language definitions to control complexity. These new features further result in additional syntactic complexity which, in turn, is caused by the way compilers operate [6].

Computer languages are basically designed for batch-oriented processing following the conventional "edit-compile" cycle, where editing and compiling are two distinct operations [6]. This lack of consideration for syntax-directed editing in the definition of computer languages directly affects the usability of syntax-directed editors for these languages. User frustration is obvious when several trivial editing operations become tedious and awkward within a syntax-directed editor [3]. This weakness is partly remedied by providing specific transformation rules [25]. Direct conversion of language constructs is specified using these transformation rules, such as if-to-while conversion, and are included as part of the normal editing operations. Nevertheless, application of such rules in the creation and alteration of software is not natural and the need for syntax-directed editing consideration in the language definition remains for the development of practical syntax-directed editors. Fischer [12] noted that redefinition and generalization of traditional language structures is required to handle their dynamic representations, a consequence of the syntax-directed editing. Hood [18] suggested that providing abstract and efficient recursive data structures and operations at the language level for the implementation of structured editor can ease their manipulation at the user level. The experience with the existing syntax-directed editing environments provides guidelines for the design of programming languages based on language semantics [9].

Enforced Construction Discipline

Syntax-directed editors impose some form of discipline for the construction of software, an attribute of the language-dependent nature of these editors. While appropriate for a novice user, a strict discipline becomes too restrictive for experienced software developers. Practical application of syntax-directed editors require these editors to support a wide variety in the construction discipline [32]. Not everybody thinks in a like manner, neither does everybody follow the same approach to software development. If a group of developers prefer program development by selection, as in the Cornell Program Synthesizer [29], the other group might regard it unnatural and follow the more traditional sequential or some other hybrid development paradigm examplified in the SED System [2]. Wide acceptance of such editing systems requires an analysis of the user preferred approaches to software construction and to incorporate an approach based on majority, if a clear majority exists. If no consensus on a particular approach is found, then a provision for multiple disciplines, with different levels or modes of operation for a range of users, should be provided.

User View of Language Structures

A syntax-directed editor operates according to the underlying syntactic structure of a computer language. All editor operations are bound to the syntactic units of the language resulting in direct manipulation of the language constructs. This structured editing is in direct accord with the way users develop their mental models. In between the user model and the internal editor operations, there is a user-editor interface. The interface provides an external representation or view of the underlying language structure. How this view is presented to the users is important since it can be in conflict with the user model and editor operations, causing confusion and frustration on the part of the developer. One of the major reasons for the limited success of such editing environments has been associated with the deficiencies in this user-editor interface [28]. Welsh [32] presents a comprehensive discussion on this subject and argues that an external view biases a user's perception and approach towards editing operations.

Varying views are provided by different editors, from textual to some form of tree representation. According to Welsh [32], a pure tree representation is equally poor as a pure textual representation since developers do not always think in tree form. Arefi [3] promoted a graphical representation suitable for language structures since textual forms do not reflect this structure properly. Stelovsky [28] suggested that structural views displaying a range of granularity level

should be provided within an editor. These views consist of textual to specific parse tree representations. In addition, the visualization of interrelationships and interdependencies of object definitions and declarations and the global representation of modules and procedure blocks collection results in the enhanced mental model of the edited program [28]. Ideally, a language-based system should support multiple views, as in PECAN [24] and Macpeth [28], representing different aspects of the underlying language structures.

Design Issues

Design is an activity aimed to achieve a specific goal and this specific objective is used in making important decisions during the design process [13]. In the context of the foregoing discussion, the specific goal is a special editor that recognizes the syntax (and probably static semantics) of a computer language. Design, in general, covers several issues such as functionality, reliability, and maintainability of a system and the design process can be directed towards either all or some of these features. In this broader sense, the usability issues discussed in the previous section also fall within the design process and have to be considered in the design of the overall system. This section does not deal with the design issues in the broader sense, but aims to identify those design issues directly related to the language for which the editor is being designed, the internal structure of the editor, and its environment. Of course, some of the usability issues are related to the language-specific decisions and the internal structure of the editor, but no attempt is made in relating these issues.

Choice of Editing Unit

An editing unit corresponds to that part of a document which can be inserted, deleted, or changed as a whole. The choice of a proper editing unit is crucial for the design of a syntax-directed editor. The editing unit depends upon the type of document being edited. For example, the editing unit in ordinary text editors is character, word, or line. In languages like BASIC or FORTRAN, the editing unit can also be a line since program statements in these languages are represented on individual lines [26]. Problems arise when decisions have to be made on the choice of editing unit for block-structured languages such as C, Pascal, or Ada. Moreover, new programming paradigms, such as declarative style of programming and object-oriented programming, also stress the identification of a proper unit if practical syntax-directed editors are to be developed for these languages. An editing unit should be simple and yet satisfy the language requirements, since complex editing units can affect the editor operations at the user level. Robson [26] suggests a definition of an editing unit for a sub-set of the Pascal language which avoids nesting and promotes a single editing unit per line.

Syntax and Static Semantic Errors

The design of a syntax-directed editor requires an analysis of the underlying language definition for the identification of possible syntax and static semantic errors. The types of errors identified and their complexity varies from language to language and hence can have a significant influence on the design decisions. An important design objective is to determine which types of errors are handled by the editor, that is, only syntactic or both syntactic and static semantic. In addition, the type and nature of errors handled by a language editor should be made consistent with the errors checked by the underlying language compiler or interpreter. When writing programs or other formal documents, developers make both anticipated and unanticipated errors. Many times, developers omit certain items on purpose, like type definitions in a program, or do not complete some items because they are not sure or they do not want to worry about it at that point. While a syntax-directed editor should take care of the unanticipated syntax errors, the above mentioned deliberate static semantic issues should be irrelevant to the editor.

Different views are supported by different editors. A number of researchers promoted the inclusion of static semantic checks in syntax-directed editors by using the incremental processing approach [4, 12, 29]. Zavodnik [34] abjured such an incremental scheme and associated it with only toy programs. According to him, the semantic issues can be handled by a separate semantic transformation using the abstract syntactic structure produced by the editor [34]. Welsh [32] approved of the idea of a separate semantic check upon user request by pointing out the fact that semantic correctness is required only at certain points in the overall development process. Unlike Zavodnik though, he did not reject the idea of incremental processing and for efficiency purposes considered such behaviour important even in the case of occassional semantic checks [32].

Another issue regarding errors is their scope. Some of the errors that are introduced either during input or editing have a limited scope, while others have a larger scope. The scope of an error can be defined as the portion of document being affected by that error. Allison [2] used the terminology of short-distance and long-distance syntax, whereas Robson [26] identified these errors as local and non-local errors. An error with limited scope is usually restricted to an expression, a statement, or a construct and falls within the category of syntactic errors. Errors with larger scope are not restricted to a particular part of a document and their affect is propagated to multiple constructs and even multiple modules. Such long range errors need to be detected by performing semantic evaluations. Syntactic and static semantic considerations need to be taken into account during the design process for the purposes of performance and complexity of syntax-directed editors.

Internal Structure Design

Users manipulate formal documents in terms of the concrete representation of these documents. All operations are performed on this representation. In actuality, these documents are maintained and manipulated in some machine interpretable form which is, or should be, closely related to the syntactic structure of these documents. All user level operations then basically are abstract operations which are mapped to the operations on this internal structure. Horgan [19] proposed the concept of a *transaction* which represents a group of atomic user level operations. These transactions are used internally as editing commands for the particular structure. All syntax-directed editors cited in this paper use some form of a tree structure to represent the target documents. These tree structures are either simple binary tree or threaded binary tree, parse tree or some form of template tree, fixed arity tree or variadic arity tree. Whatever type of tree structure is employed, the aim is to provide efficient storing, retreiving, updating, and movement about the structure.

YALE represents the language grammar rules as template data structures internally and stores these structures collectively in a file [34]. According to Donzeau [9], tree structures are easier to implement and maintain than more complicated structures, such as control flow graphs, where the maintenance cost of the information is much higher than the benefits obtained. The level of detail stored in the internal structure affects the complexity and size of the internal data structure. Large data structures with more efficient manipulation mechanisms are required to handle semantic information than the ones for only syntactic information. Hence, a compromise is required between the amount and complexity of information stored and the ease of implementation of the data structure.

Another significant issue to be considered in the design is the level of abstractness employed for the terminal nodes of the structure. Donzeau, et al. [8, 9] used the level of operators and operands to represent the terminal nodes of the internal tree structure in MENTOR. Teitelbaum and Reps raised this level one step up and used phrases or expressions as the terminal nodes in their synthesizer [29]. Allison [2] identified that the movement around large constructs and procedures is efficient compared to locating a variable within an expression. He promoted a higher level for terminal nodes, typically simple statements and data declarations [2].

Integration in an Environment

A syntax-directed editor is a tool used by developers in the construction of formal documents. These formal documents span the entire software development life cycle, from requirements analysis to maintenance. The different phases of the life cycle are supposed to complement each other in terms of exchanging information and control. Various tools for each phase should be well integrated to form a comprehensive development environment. This integration has both a user view requiring a collection of seamless tools to facilitate construction of systems, and a designer view which is concerned with the feasibility and design of this integrated environment [31]. User consideration is important. In fact, one of the purposes of tool integration is to provide a consistent user-interface and to hide the interworkings of individual tools [11]. How such an interface is provided is dependent on the way tool integration is achieved. Alternatives have been suggested to achieve such an integration. These alternatives range from providing a homogeneous command language for all tools to the use of a common data base as an integrating and unifying medium for interfacing various tools [20].

The integration of a language-specific editor in an enviornment requires certain design issues to be considered and analysed for feasibility. First of all, there is the factor of realization. The designer of any tool should realize that the tool has to be a part of an integrated enviornment and not a stand alone tool. Lack of such a realization will result in improper and inflexible design for the later interfacing with other tools. Secondly, an evaluation of existing tools is required for the integrated design of the editor. This evaluation will consist of the interfaces of the existing tools, media of communication, type of exchanged information, and products of each tool. An analysis of the underlying language features is also important for a language-specific editor. These language features provide a framework for the basic design and guide the design process. After all, a language-specific editor is a tool for a specific language. Version management, configuration management, traceability support, and automatic documentation are a few of the necessary features of a quality software development process and should be provided to support the notion of syntax-directed editors. Consideration of integration of such editors with the above mentioned necessary features is important for these editors to become pragmatic tools and be part of an integrated CASE environment.

Features of Syntax-Directed Editors

Based on the discussion of the usability and design issues in the previous two sections, certain features can be identified for the development or use of syntax-directed editors. Developers can ensure the addressing of at least these features in the creation of the editors. Users, on the other hand, can analyze a variety of such environments in light of these features and see if a particular editing environment satisfies the features they prefer. Some sort of standardization of these features is recommended across languages. These features are: external representation, internal structure type, level of abstractness, target documents, types of errors, error handling, incremental system, execution level, and environmental support. The features identified are enumerated below along with a small description of each feature.

External Representation

External representation can consist of either textual templates or some form of graphical symbolic templates. Indented text has been identified as one form of visual representation [15]. Here, the term visual or graphical is taken in the strict sense of some pictorial form.

Internal Structure Type

Internal structure corresponds to the type of data structure employed and its manipulation mechanism. This information reflects the complexity and size of the structure and hence storage consumption and response time.

Level of Abstractness

Closely associated with the internal structure, the level of abstractness feature identifies the terminal node format for the data structure. The type of formats can be character level, expression level, simple statement level, or declaration level. This feature also affects the storage and manipulation mechanisms of the internal data structure.

Target Documents

The type of documents a particular editor prepares, identifies its range of applicability. The documents can range from specifications through programs to some form of formatted documentation.

Types of Errors

Two types of errors are identified: syntactic and static semantic. A particular editor can either handle syntactic errors or both. The choice is application dependent.

Error Handling

A particular editor can either abort on errors or continue

editing even in the presence of errors. In the first case, the editor halts and does not proceed until the error is corrected. In the latter case, the editor marks errors for later correction. An extreme approach is to automatically correct the errors encountered. An editor employs either one or a combination of these error handling mechanisms.

Incremental System

The incremental approach is used by a number of editors to avoid recompilation or reparsing. The presence or absence of incremental processing depends upon the nature of application and the types of documents prepared and can be consideed either a strength or a weakness.

Execution Level

The execution level feature is applicable to only those documents which falls within the operational domain. An editor can either support partial through complete or only complete document execution. Partial execution may be good for top-down development. Some editors do not support any form of execution.

Environmental Support

An editor can be a stand alone tool or complemented by a variety of tools integrated in an environment. The types of supporting tools and the level of integration is important for a particular editing environment and can be used as a criterion for evaluating such environments.

Analysis of Syntax-directed Editing Environments

A number of syntax-directed editing environments exist, ranging from small-scale programming environments to more extensive environments, further evolving into syntax-directed editor generator systems. This section presents an analysis of some of these editing environments. Seventeen environments were identified for the analysis. The analysis is carried out in the context of the features identified in the last Section and is presented in a tabular form. All information presented in the table for the selected environments is gathered from the sources cited in the reference section. Feature areas marked "unknown" in the table indicate that either the information is not available in the cited source or it is not clear. The editing environments are organized in a chronological order and are presented in Table 1.

Summary

The paper identified a number of issues related to syntax-directed editors and their environments. The issues were used to formulate certain features for such editing environments. These features are expected to provide a framework for guiding the development of syntax-directed editing enviornments. A number of existing editing environments were analyzed based on the formulated features. A summary of the analysis was presented in the tabular form for a quick reference of the required or desired features.

Amir A. Khwaja is a graduate student in the Department of Computer Science and Engineering at Arizona State University.
Joesph E. Urban is a professor of computer science at Arizona State University.

Authors' email address is
{ khwaja | jurban }@enuxha.eas.asu.edu

References

[1] M. B. Albizuri-Romero, "GRASE - A Graphical Syntax-directed Editor for Structured Programming," *ACM SIGPLAN Notices*, Vol. 19, No. 2, February 1984, pp. 28-37.

[2] L. Allison, "Syntax Directed Program Editing," *Software - Practice and Experience*, Vol. 13, No. 5, May 1983, pp. 453-465.

[3] F. Arefi, C. E. Hughes, and D. A. Workman, "Automatically Generating Visual Syntax-Directed Editors," *Communications of the ACM*, Vol. 33, No. 3, March 1990, pp. 349-360.

[4] L. V. Atkinson, J. J. McGregor, and S. D. North, "Context Sensitive Editing as an Approach to Incremental Compilation," *The Computer Journal*, Vol. 24, No. 3, August 1981, pp. 222-229.

[5] D. R. Barstow, "A Display-Oriented Editor for INTERLISP," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall (Editors), McGraw-Hill, 1984, pp. 288-299.

[6] M. Caplinger, "Structured Editor Support for Modularity and Data Abstraction," *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 25-28, 1985, pp. 140-147.

[7] R. N. Charette, *Software Engineering Environments: Concepts and Technology*, Intertext Publications, New York, N. Y., 1986.

[8] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J. J. Levy, "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming," *Proceedings of the International Computing Symposium*, Antibes, France, June 2-4, 1975, pp. 113-120.

[9] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The MENTOR Experience," in *Interactive Programming Environments*, D. R. Barstow, H. E. Shrobe, and E. Sandewall (Editors), McGraw-Hill, 1984, pp. 128-140.

[10] J. W. M. Dooley and S. R. Schach, "FLOW: A Software Development Environment Using Diagrams," *The Journal of Systems and Software*, Vol. 5, No. 3, August 1985, pp. 203-219.

[11] E. Fedchak, "An Introduction to Software Engineering Environments," *Proceedings of the IEEE Computer Society 10th Annual International Computer Software and Application Conference (COMPSAC 86)*, Chicago, Illinois, October 8-10, 1986, pp. 456-463.

[12] C. N. Fischer, G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock, "The Poe Language-based Editor Project," *Proceedings of the ACM SIGSOFT/SIGPLAN Software*

Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, April 23-25, 1984, pp. 21-29.

[13] P. Freeman, "Fundamentals of Design," in *Tutorial on Software Design Techniques*, 4th edition, P. Freeman and A. I. Wasserman (Editors), IEEE Computer Society Press, 1983, pp. 2-22.

[14] E. R. Gansner, J. R. Horgan, D. J. Moore, P. T. Surko, D. E. Swartwout, and J. H. Reppy, "SYNED - A Language-based Editor for an Interactive Programming Environment," *Digest of Papers, Spring COMPCON 83, 26th IEEE Computer Society International Conference*, San Francisco, California, February 28 - March 3, 1983, pp. 406-410.

[15] E. P. Glinert, "Nontextual Programming Environments," in *Principles of Visual Programming Systems*, S. -K. Chang (Editor), Prentice-Hall, New Jersey, 1990, pp. 144-230.

[16] K. Halewood and M. R. Woodward, "NSEDIT: A Syntax-directed Editor and Testing Tool Based on Nassi-Shneiderman Charts," *Software - Practice and Experience*, Vol. 18, No. 10, October 1988, pp. 987-998.

[17] W. J. Hansen, "User Engineering Principles for Interactive Systems," *Proceedings of the Fall Joint Computer Conference*, 1971, pp. 523-532.

[18] R. Hood, "Efficient Abstractions for the Implementation of Structured Editors," *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, Washington, June 25-28, 1985, pp. 171-178.

[19] J. R. Horgan and D. J. Moore, "Techniques for Improving Language-based Editors," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 23-25, 1984, pp. 7-14.

[20] W. Howden (Ed.), "Contemporary Software Development Environments," *NBS Programming Environment Workshop*, Rancho Sante Fe, California, April 29 - May 2, 1980, pp. 9-27.

[21] D. Kiong and J. Welsh, "Incremental Semantic Evaluation in Language-based Editors," *Software - Practice and Experience*, Vol. 22, No. 2, February 1992, pp. 111-135.

[22] J. Nielsen, "The Usability Engineering Life Cycle," *Computer*, Vol. 25, No. 3, March 1992, pp. 12-22.

[23] M. C. Pong and N. Ng, "PIGS - A System for Programming with Interactive Graphical Support," *Software - Practice and Experience*, Vol. 13, No. 9, September 1983, pp. 847-855.

[24] S. P. Reiss, "Graphical Program Development with PECAN Program Development Systems," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 23-25, 1984, pp. 30-41.

[25] T. W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-based Editors*, Springer-Verlag, New York, 1989.

[26] D. J. Robson, "Towards a Conversational Language-Sensitive System for Pascal," *Software - Practice and Experience*, Vol. 13, No. 11, November 1983, pp. 1013-1017.

[27] E. Sandewall, "Programming in an Interactive Environment: The LISP Experience," *ACM Computing Survey*, Vol. 10, No. 1, March 1978, pp. 35-71.

[28] J. Stelovsky, D. Ackermann, and P. Conti, "Visualizing of Program Structures: Support Concepts and Implementation," in *Visualization in Programming, 5th Interdisciplinary Workshop in Informatics and Psychology*, Scharding, Austria, May 20-23, 1986, Springer-Verlag, Heidelberg, 1987, pp. 37-52.

[29] T. Teitelbaum and T. W. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM*, Vol. 24, No. 9, September 1981, pp. 563-573.

[30] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," *Computer*, Vol. 14, No. 4, April 1981, pp. 25-34.

[31] I. Thomas and B. A. Nejmeh, "Definitions of Tool Integration for Environments," *IEEE Software*, Vol. 9, No. 2, March 1992, pp. 29-35.

[32] J. Welsh, B. Broom, and D. Kiong, "A Design Rationale for a Language-based Editor," *Software - Practice and Experience*, Vol. 21, No. 9, September 1991, pp. 923-948.

[33] J. Wilander, "An Interactive Programming System for Pascal," *BIT*, Vol. 20, No. 2, 1980, pp. 163-174.

[34] R. J. Zavodnik and M. D. Middleton, "YALE - The Design of Yet Another Language-based Editor," *ACM SIGPLAN Notices*, Vol. 21, No. 6, June 1986, pp. 70-78.

[35] M. V. Zelkowitz, "A Small Contribution to Editing with a Syntax-Directed Editor," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, April 23-25, 1984, pp. 1-6.

Editor	External Represent	Internal Structure	Level of Abstract	Target Language	Types of Errors	Error Handling
Pathcal [33] 1980	Textual	Parse Tree	Unknown	Pascal	Syntax Semantic	Halts on Errors - Does not Proceed Until Corrected
Context Sensitive Editing [4] 1981	Textual	Tree	Operator Operand	Algol 60	Syntax Static Semantic	Halts on Errors - Does not Proceed Until Corrected
Cornell Program Synthesizer [29] 1981	Textual	Derivation Tree	Phrases or Expressions	PL/I	Syntax Static Semantic	Continues but Marks Errors
Syned [14] 1983	Textual	Tree	Unknown	C	Syntax Static Semantic	Unknown
SED [2] 1983	Textual	Parse Tree	Assignment Declaration	Pascal	Syntax	Allows Errors
PIGS [23] 1983	Graphical	Tree	Unknown	Pascal	Syntax Static Semantic	Halts on Errors - Does not Proceed Until Corrected
GRASE [1] 1984	Graphical	Tree	Unknown	Pascal	Syntax Static Semantic	Unknown
DED [5] 1984	Textual	List	Expression	Lisp	Unknown	Unknown
MENTOR [9] 1984	Textual	Tree	Operator Operand	Pascal	Syntax	Unknown
Poe [12] 1984	Textual	Parse Tree	Token	Pascal	Syntax Static Semantic	Automatic Error Repair Continues but Marks Errors
SUPPORT [35] 1984	Textual	Tree	Expression	Pascal	Syntax Static Semantic	Unknown
Yg [6] 1985	Textual	Tree	Unknown	Multiple	Syntax Static Semantic	Unknown
FLOW-EDIT [10] 1985	Graphical	Tree	Unknown	Pascal Fortran, Cobol	Syntax Static Semantic	Halts on Errors - Does not Proceed Until Corrected
YALE [34] 1986	Textual	Binary Tree	Expression	Multiple	Syntax	Unknown
Macpeth [28] 1987	Textual Graphical	Parse Tree	Operator Operand	Modula-2 Pascal	Syntax Static Semantic	Continues but Marks Errors
NSEDIT [16] 1988	Graphical	Parse Tree	Text Strings	Pascal	Unknown	Unknown
UQ1 & 2 [32] 1991	Textual	Tree	Unknown	Multiple	Syntax Static Semantic	Rejects Incorrect Input Marks Edit Errors & Continue

Table 1
Editor Chart

Editor	Incremental System	Execution Level	Environmental Support
Pathcal [33] 1980	Yes	Partial through Complete	Interpreter, Debugger
Context Sensitive Editing [4] 1981	Yes	Complete	Interpreter
Cornell Program Synthesizer [29] 1981	Yes	Partial through Complete	Interpreter, Debugger
Syned [14] 1983	Yes	Unknown	None
SED [2] 1983	No	Unknown	None
PIGS [23] 1983	Yes	Partial through Complete	Debugging Interpreter Dynamic Dataflow Analysis for Testing
GRASE [1] 1984	Unknown	Unknown	Unknown
DED [5] 1984	Yes	Partial through Complete	Unknown
MENTOR [9] 1984	No	None	Program Normalization & AutomaticDocumentation Tools, Source Debugger
Poe [12] 1984	Yes	Complete	None
SUPPORT [35] 1984	Unknown	Partial through Complete	Program Design Language (PDL) Support Debugger
Yg [6] 1985	Unknown	Unknown	Source Code Control System Debugging Interpreter
FLOW-EDIT [10] 1985	Unknown	Partial through Complete	Automatic Documentation, Debugger, Compiler Software Production Database, Code Transformer
YALE [34] 1986	No	None	Unknown
Macpeth [28] 1987	Unknown	Unknown	None
NSEDIT [16] 1988	No	Partial through Complete	Test Coverage Tool
UQ1 & 2 [32] 1991	Yes	Unknown	Unknown

Table 1
(Continued)