# A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation

**Gail E. Kaiser**
**Peter H. Feiler***
**Fahimeh Jalili**+
**Johann H. Schlichter**=

June 1988
(revised December 1988)

CUCS-356-88

## Abstract

DOSE is unique among structure editor generators in its interpretive approach. This approach leads to very fast turn-around time for changes and provides multi-language facilities for no additional effort or cost. This article compares the interpretive approach to the compilation approach of other structure editor generators. It describes some of the design and implementation decisions made and remade during this project and the lessons learned. It emphasizes the advantages and disadvantages of DOSE with respect to other structure editing systems.

Prof. Kaiser is supported in part by grants from AT&T, IBM, Siemens and Sun, in part by the Center of Advanced Technology and by the Center for Telecommunications Research, and in part by a DEC Faculty Award. The DOSE system was developed at Siemens Research and Technology Laboratory in Princeton, NJ. *Carnegie Mellon University, Software Engineering Institute, 5000 Forbes Avenue, Pittsburgh, PA 15213. Dr. Feiler is supported by the Department of Defense. +AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. =Xerox Corporation, Joseph Wilson Research Center, 800 Phillips Road, Webster, NY 14580.

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(8), 733–748 (AUGUST 1988)

# A Retrospective on DOSE: An Interpretive Approach to Structure Editor Generation

GAIL E. KAISER
*Columbia University, Department of Computer Science, 450 Computer Science Building, New York, NY 10027, U.S.A.*

PETER H. FEILER
*Carnegie Mellon University, Software Engineering Institute, 5000 Forbes Avenue, Pittsburgh, PA 15213, U.S.A.*

FAHIMEH JALILI
*AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.*

AND

JOHANN H. SCHLICHTER
*Xerox Corporation, Joseph Wilson Research Center, 800 Phillips Road, Webster, NY 14580, U.S.A.*

**SUMMARY**

**DOSE is unique among structure editor generators in its interpretive approach. This approach leads to very fast turn-around time for changes and provides multi-language facilities for no additional effort or cost. This article compares the interpretive approach to the compilation approach of other structure editor generators. It describes some of the design and implementation decisions made and remade during this project and the lessons learned. It emphasizes the advantages and disadvantages of DOSE with respect to other structure editing systems.**

KEY WORDS Language-based editor Programming environment Structure editor generation

## INTRODUCTION

The Display Oriented Structure Editor system (DOSE) is a structure editor generator. Like other structure editor generators, DOSE can be instantiated with a programming language and a corresponding collection of language-based tools. The result is an integrated programming environment for the desired programming language. Unlike other structure editors, the instantiation process involves interpretation rather than translation technology. This approach supports rapid prototyping and multiple language capabilities.

0038–0644/88/080733–16$08.00

*Received 27 December 1985*
*Revised 20 March and 10 December 1987*

## Structure editors

A *structure editor* is a special kind of editor that is different from a text editor. It has more in common with a forms editor. The basic idea is a fill-in-the-blanks approach to program construction and modification. A new Pascal program is constructed as follows. First, the user asks for a new program template. The structure editor creates a new template and displays it as shown in Figure 1. The editor provides the keywords, punctuation and indentation. The user must supply the missing parts — the actual content of the program.

```
program $name (input, output);
const
    $constant;
type
    $type;
var
    $variable;

$routine;

begin
    $statement
end.
```

*Figure 1. Template for a Pascal program*

The missing parts, called *metanodes*, are initially displayed as *$category*. The category indicates the set of alternative templates that may replace the metanode. For example, the *$statement* metanode can be replaced with a while template, an if template, a case template, etc. If the user attempts to replace the *$statement* metanode with a template that is not a member of the statement category, then the editor displays an error message and disallows the replacement. It is not possible for the user to replace *$statement* with the + template or the = template. The structure editor guarantees that the program is syntactically correct at all times.

Say the user wished to replace the *$name* metanode with test, an identifier for the name of the program. The user moves the cursor to highlight *$name*, using a mouse or by entering commands similar to the cursor movement commands of text editors. The important difference is that the cursor always points to a syntactic unit, either a template or a metanode. It is not possible to position the cursor at an arbitrary character. Once the cursor is correctly positioned at *$name*, the user types 'test'. The entered text is echoed in the appropriate place on the screen, as in a text editor. The updated display is shown in Figure 2.

```
program test (input, output);
const
    $constant;
type
    $type;
var
    $variable;

$routine;

begin
    $statement
end.
```

*Figure 2. Filling the $name metanode*

To add a type definition, the user moves the cursor to highlight the *$type* metanode and types 'type' (this might be abbreviated by the unique prefix 't'), and the structure editor adds the type template. The user input is echoed on a command line, as are the commands in many text editors. Alternatively, the user might select type from a menu listing only the legal replacements for the current metanode. In either case, the relevant portion of the updated display is shown in Figure 3. Notice that the editor automatically adds a new *$type* metanode, so the user can add more types to the program.

```
type
    $name = $type_definition;
    $type;
```

*Figure 3. Filling the $type metanode*

Say the user does not want to include any constant definitions in the program. The user moves the cursor to highlight the *$constant* metanode and types the Delete command (this might be abbreviated as (ctrl)-d, as in a text editor, or selected from a menu listing the commands valid in the current context). This has the effect of deleting the *$constant* metanode. The relevant portion of the program is then displayed as shown in Figure 4.

```
program test (input, output);
<no constants>
type
    $name = $type_definition;
    $type;
```

*Figure 4. Deleting the $constant metanode*

The user can change his mind, and add one or more constants by moving the cursor to highlight (no constants) and selecting constant. The structure editor replaces the empty list of constants with a constant template, as shown in Figure 5.

```
program test (input, output);
const
    $name = $value;
    $constant;
type
    $name = $type_definition;
    $type;
```

*Figure 5. Creating a* constant *template*

This style of program construction and modification is sometimes called *template-based* editing. All changes in the program are performed in terms of templates that represent the structure of the programming language, so the program is always syntactically correct although it may be incomplete. Some users familiar with text editing are uncomfortable with template-based editing, particularly for expressions. These users complain about the necessity of entering expressions in prefix form and being unable to move the editing cursor according to the characters shown on the display rather than conforming to the hierarchical structure of the program. These problems have been addressed by several editing systems.[1-3] These systems use incremental parsing technology[4-6] to provide a text-oriented user interface, but immediately detect and report syntax errors.

## Structure editor generators

The earliest structure editors, for Lisp, were in active use by 1965.[7] Emily,[8] developed in 1969, was the first structure editor for a block-structured language. Many structure editors for many different languages have been developed in the past ten to twelve years. Some of these structure editors were developed for a specific programming language. In this case, the source code of the structure editor includes templates and display information for each construct in the programming language. The Cornell Program Synthesizer[9] is probably the best known example of a hand-coded structure editor.

Other structure editors have been generated using a program called a *structure editor generator*. The Synthesizer Generator[10] and the Gandalf ALOE system[11] are two well-known structure editor generators. The source code of a generator system does not include any information about any particular programming language. Instead, it includes routines that handle the manipulation and display of arbitrary templates. The implementor generates a structure editor for a particular programming language by writing a description of the language as a form of context-free grammar. For example, the implementor specifies the Pascal program template as shown in Figure 6. The implementor must describe both the metanodes and the display of the program template. The metanodes are defined by giving a *production* that specifies the categories for the components of the template. The display information is omitted.

```
program => name: identifier
           constants: seq of constant
           types: seq of type
           variables: seq of variable
           routines: seq of routine
           body: seq of statement
```

*Figure 6. Description of* program *template*

The program production defines the program template as consisting of six components. Whenever the user of the structure editor requests the creation of a program template, the editor creates a template with these six components. If the user moves the editing cursor to highlight the entire program template and gives the Delete command, the editor destroys the entire template with all its components.

The first component of each program template is called name. The name component is defined to be an *identifier*, a built-in production that defines a terminal. A *terminal* is a special template that does not have any components; instead, it has a *value*, which must be entered by the user as characters. The test identifier illustrated in Figure 2 is an example of a terminal. Most structure editors provide at least *integer*, *real*, *string* (single line) and *text* (multiple lines) as well as *identifier* (no blanks) as built-in terminal productions.

The remaining five components of a program are *sequences*. 'seq of category' means a sequence of one or more templates that are members of the category, where the category may be either a production name or a class name. A *class* specifies a set of alternative productions. For example, the sixth component is a list of instances of productions that are members of the statement class, given in Figure 7. Any one of these alternatives can be chosen to replace a *$statement* metanode. When a class appears as the category for a sequence, the elements of the sequence may be the same or may be different alternatives.

```
statement ::= assign call case compound
              for goto if repeat while with
```

*Figure 7. Description of statement class*

A structure editor generator consists of two separate programs. One program, called the *kernel*, provides the language-independent manipulation and display routines. The kernel typically includes a command interpreter, a window manager, an interface to the file system and the operating system, and so on. The second program is called the *translator*. It takes as input the description of the language given by the implementor, compiles this description into tables, and then links the tables together with the kernel. The result is a structure editor for the particular programming language.

## DOSE

In addition to those structure editors that were hand-coded and those that were generated in the manner described above, there are also structure editors that have been developed by instantiating DOSE. DOSE is an 'interpreter' structure editor generator; the generators described above are 'compilers'. Both require the implementor to provide a description of the desired programming language. The most important distinction is that DOSE interprets the description, whereas other structure editor generators translate the description into some other form. A secondary difference is that DOSE is a single program, the interpreter (kernel), whereas other generators consist of two separate programs: the compiler or translator and the run-time environment (kernel). Both differences are illustrated by Figures 8 and 9. The first shows the compiled approach of a typical structure editor generator when applied to multiple programming languages; the second illustrates the analogous operation of DOSE.

DOSE is used in a different fashion to other structure editor generators. In the latter case, the programming language description is developed in some manner, often using a special structure editor where the 'programming language' is actually the notation for descriptions. The description is then input to the translator. The translator first analyses the description, checking for static semantic errors such as a production name that is used but not defined. If there are no errors, the translator generates tables representing the description, compiles these tables and links the resulting object code together with the kernel. The output is a structure editor for the desired language. This manufacturing process takes from a few minutes to a few hours, depending on the size and complexity of the description. If the user wants to change languages, he must exit the generated structure editor for the first language and enter the editor for the second language, or change windows on a workstation; there is a separate program for each language.

In the case of DOSE, the programming language description is written using DOSE itself, where the 'programming language' is the description notation. The new description can then be selected immediately as the current description, resulting in an editor for the specified programming language. Alternatively, the new description can be stored in a file and loaded during a separate execution of DOSE. In either case, DOSE first checks for static semantic errors, which may take from a few seconds to a few minutes depending on the size and complexity of the language. If no errors are detected, DOSE then behaves as a structure editor for the desired language. No translation or additional processing time is required.

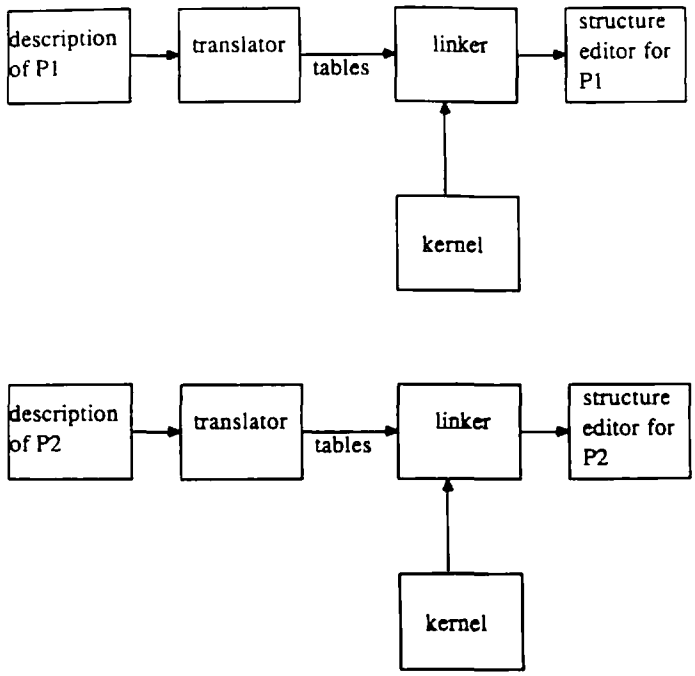



*Figure 8. Compiled structure editor generation*

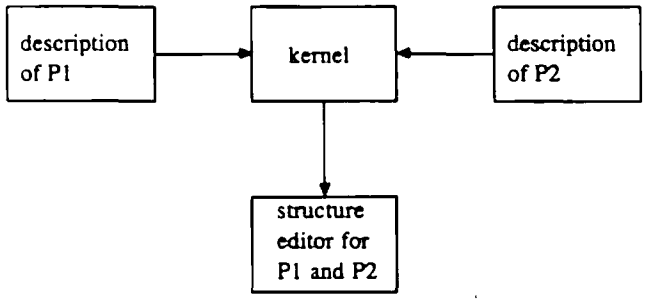



*Figure 9. Interpreted structure editor generation*

If the user is currently editing a program and wants to modify its description, the Grammar command returns to editing the description. The user may edit the description arbitrarily, and then select the Program command to return to editing the program. If the description is still compatible with the program — for example, no productions actually used in the program have been deleted — the user can continue editing the program. The user can switch back and forth between program and description editing as desired.

If the user wants to change languages, the LoadGrammar command loads the description for the second language from secondary storage. A few seconds are required for the actual file I/O and for checking for static semantic errors (since DOSE permits users to store incorrect descriptions). The user can switch back and forth between editing programs in any of the loaded languages without ever leaving the structure editor. Up to ten languages, and twenty-five programs in any subset of these languages, may be loaded simultaneously, where ten and twenty-five are arbitrarily chosen

implementation limits. Since the user must select a specific description when beginning a new program and every existing program keeps an indication of its description, there is no ambiguity if the same name is used in multiple descriptions. For example, there may be several descriptions loaded simultaneously that all have an expression class. The only restriction is that each of the descriptions themselves must have a unique name, such as Pascal and C.

Rapid prototyping and multi-language capabilities are the primary advantages of DOSE over structure editor generators. These are discussed in more detail elsewhere.[12–14] The rest of this article concentrates on the implementation of DOSE in comparison to other structure editor generators.

## STRUCTURE EDITOR IMPLEMENTATION

In DOSE, as well as in most other structure editors, the program is represented internally in the form of a tree, either an abstract syntax tree or a parse tree. An abstract syntax tree is preferred, since it is more compact; in particular, 'syntactic sugar' such as keywords and punctuation are not represented explicitly in the tree.[15] Each node in the program tree is implemented by a pointer to a record similar to the one shown in Figure 10. This particular record is for a *non-terminal* node, meaning that the node has one or more children. The optype (operator type) field refers through some mechanism to the production that defines the node; this is explained below. The parent field is a pointer back to the node's parent and the children field is an array of pointers to its children.

```
TYPE NodeRef = POINTER TO Node;

TYPE Node =
    RECORD
        optype: ...;
        parent: NodeRef;
        children: ARRAY [1..arity] OF NodeRef;
    END RECORD;
```

*Figure 10. Implementation of a non-terminal node*

The program tree is maintained by the structure editing kernel. The kernel provides the primitive capabilities for manipulating and traversing the tree, including routines to create a new node, destroy a node, insert a copy of an existing node as a particular child of another node, etc. The kernel also includes routines to move the cursor from one node to another, following parent–child and child–parent links. The kernel typically provides powerful traversal and manipulation routines to support search operations and program transformations.

### Compilation

The kernel as described so far is more-or-less the same for both traditional structure editor generators (compiled) and for DOSE (interpreted). The difference lies in how the system tailors its kernel to a specific programming language. For compilation, some description of the programming language is processed by a translator into a table in the implementation language of the system. Each entry in the table represents all the information specific to a particular kind of node.

For example, the table might be represented by an array of records. One possible set of fields for the entry record is shown in Figure 11. Here each entry has a string field that gives the name of the production or class. The other fields are determined by the appropriate variant, whether the entry represents a class, a terminal production, a non-terminal production or a sequence. A class has a number of members, each described by another entry in the table. A terminal has a value of a particular type, such as integer, string or identifier. A non-terminal has a fixed number of children, where the type of each child is described by another entry; a sequence has an arbitrary number of elements of the same type, which is represented by another entry. In addition to these fields, each entry also includes display and semantic information, not shown here. The table is compiled and linked with the kernel to produce a language-specific editor.

```
TYPE Offset = 1..MaxTable;
TYPE Table = ARRAY [Offset] OF Entry;
TYPE Entry =
    RECORD
        name: STRING;
        CASE tag: NodeCategory OF
            Class:          (members: ARRAY [1..MaxMember] OF Offset);
            Terminal:       (value: ValueType);
            NonTerminal:    (arity: PosInt;
                             components: ARRAY [1..arity] OF Offset);
            Sequence:       (elements: Offset);
        END CASE;
    END RECORD;
```

*Figure 11. Possible record format for a language table*

Consider a generated structure editor for Pascal. Say the user requests that the kernel createa a new if node as the first child of a particular + node. The kernel responds as follows. It accesses the optype field of the + node. As shown in Figure 12, this field is of type Offset, and gives the table index for the + entry. The kernel accesses this entry, finds that + is a non-terminal node, and obtains the offset for the descriptor of its first child. This offset happens to point to the expression class. The kernel proceeds by checking the entry for each offset in its members array until it discovers that none of these offsets points to an entry whose name is if. Then the kernel prints an error message and does not actually perform the creation.

```
TYPE Node =
    RECORD
        optype: Offset;
        parent: NodeRef;
        children: ARRAY [1..arity] OF NodeRef;
    END RECORD;
```

*Figure 12. Compiled implementation of a non-terminal node*

Calling this scenario 'compilation' may seem incorrect, since the tables are in some sense interpreted by the kernel. A true compilation system would not generate tables; instead, it would generate a language-specific kernel. There would then be a distinct type for each kind of node; examples for the program production and statement class are illustrated in Figure 13. In practice, a language-specific kernel could be implemented only in a language that supports some form of dynamic typing. Otherwise, it would not be possible to write generic routines for manipulating arbitrary program trees containing nodes of many types.

```
TYPE ProgramNode =
    RECORD
        parent: NodeRef;
        name: IdentifierRef;
        constants: ARRAY [1..Max] OF ConstantNode;
        types: ARRAY [1..Max] OF TypeNode;
        variables: ARRAY [1..Max] OF VariableNode;
        routines: ARRAY [1..Max] OF RoutineNode;
        body: ARRAY [1..Max] OF StatementNode;
    END RECORD;

TYPE StatementNode =
    UNION
        CompoundNode, IfNode, WhileNode, GotoNode, ...
    END UNION;
```

*Figure 13. Node types for a language-specific kernel*

## Interpretation

DOSE tailors a language-independent kernel by interpreting the original form of the language description. In this case, no translator is needed. Instead, the kernel understands the format used for the language description; this involves slightly more advanced capabilities than understanding the format of tables, but these capabilities are required anyway in the translator in order to produce the tables. The difficulty is not one of capability, but of performance.

Interpretation would be intolerably slow if the language description was maintained in textual form. The kernel would have to parse the description each time it needed to find the relevant information for an editing operation. The alternative is to maintain the description in some internal representation that can be speedily accessed. Of course, the obvious internal representation is a table, as used in compilation. However, there is another original form for language descriptions other than text — an abstract syntax tree — and this form does not need to be parsed. As already noted, several structure editor generators combine their translator with a structure editor specific to the description notation. The user edits descriptions as well as programs in terms of templates, and the descriptions are maintained in the same internal representation as are programs. In the compilation approach, the translator then translates from this internal representation to a table.

In contrast, the DOSE kernel interprets the description directly. Since the description is in the same form as the programs, the kernel already contains all the appropriate facilities for traversing and accessing the description. A node in the program tree is implemented by a pointer to the record shown in Figure 14; this particular record is similar to the nonterminal node illustrated in Figure 12 for the compilation approach. The only difference is that the type of the optype field is Offset in the compilation case whereas NodeRef provides an appropriate internal representation for interpretation. This field points to the record for the production that defines the node.

Consider the case where the record in Figure 14 represents a Pascal program template, as depicted in Figure 1. Then the optype field points to the internal representation of the program description, whose logical representation is shown in Figure 6. Since the program production is itself a non-terminal node, its internal form is another record of exactly the same type as the program template node.

The question arises as to the meaning of the optype field of the program production node. If the notions of production node, class node, etc. were hard-coded into the kernel, then the optype could be some special value that indicates the kind of node. This value would be used by the kernel when it interprets a node in the language

```
TYPE Node =
    RECORD
        optype: NodeRef;
        parent: NodeRef;
        children: ARRAY [1..arity] OF NodeRef;
    END RECORD
```

*Figure 14. Interpreted implementation of a non-terminal node*

description to determine how it should be used. In the case of a production node, the kernel would know that its first child is the name of the production, the second is the arity and the third child is the list of component descriptions; in the case of a class node, the kernel would know the first child is the name of the class and the second is the list of member descriptions.

DOSE takes an alternative approach: the optype of every node points to a node, whether or not the node is in a program tree or in a description tree. In the case of a program tree, the optype points to a node in the corresponding description tree; that is, the program template node points to the program production node. In the case of a description tree, the optype points to a node in a distinguished description tree, which defines the description for descriptions. This description is called the 'grammar grammar'. Thus every tree has a description, including description trees; the grammar grammar tree acts as its own description.

The portion of the grammar grammar for productions and classes is illustrated in Figure 15. The production production defines a production node as having three children: its name is an identifier, its arity is an integer and its components are a sequence of component nodes. Each production node in a language description, such as the program production node above, points to the node in the grammar grammar tree that represents the production production.

```
production => name: identifier
              arity: integer
              components: seq of component

component => label: identifier
             sequence: boolean
             type: identifier

class => name: identifier
         elements: seq of identifier
```

*Figure 15. Portion of the 'grammar grammar'*

Since the grammar grammar is its own description, the optype fields of the production production, the component production, etc. point into their own tree. The production production node points to itself, and all the other production nodes point to it as well. Any classes in the grammar grammar point to the class production node, which in turn points to the production production node. Thus, DOSE has three levels of interpretation. The kernel manipulates a program node by interpreting the program production node; it manipulates the program production node by interpreting the production production node; and finally, it manipulates the production production node by interpreting the production production node itself.

There seems to be a flaw here somewhere: an infinite circularity in interpretation. It looks as if the kernel would never get any useful work done. DOSE solves this problem by hard-coding certain portions of the grammar grammar in order to break

the circularity. However, the full grammar grammar is in fact represented internally as a tree along with all the descriptions and programs, and this representation is interpreted for most operations.

When any 'program' tree, including a description, is stored on secondary storage, the tree is linearized in the obvious manner. At the beginning of the file is the name and version number of the description for this tree. The optype of each node is stored as the name of a production in this description. When a program tree is loaded, DOSE first looks at the name and version number of its description. If this description tree is not already loaded, DOSE invokes the loading routine recursively to load it. For efficiency, the grammar grammar tree is automatically reconstructed, rather than loaded, each time DOSE begins execution. As the nodes in the program tree are finally read from the file, each production name is looked up in the description tree and the optype field of the program tree node linked to the appropriate description tree node.

### Discussion

The advantages of interpretation are many, and the disadvantages are few. The obvious potential disadvantage is performance, since it seems that interpretation would be significantly slower than compilation. However, this is not necessarily the case. In performance comparisons between the language description editor compiled using ALOE[16] and the language description editor interpreted by DOSE, there were no significant differences in response time. Profiles of the two systems showed that both spent the majority of their time updating the display after changes and executing the routines that perform semantics processing.

The primary advantage of interpretation is the fast turn-around time while developing an editor. Using DOSE, the implementor can switch back and forth between editing a test program in the target language and editing its language description with one or two keystrokes. Switching from editing a program to editing a language description is apparently instantaneous, whereas switching in the opposite direction results in a brief delay for static semantic analysis. In practice, the delay has run from ten seconds to thirty seconds. In contrast, the compilation and linking required by editor generators using compilation technology is often a matter of minutes, not seconds, and the implementor must leave the language description editor to invoke the produced language-specific editor. The single DOSE program supports multiple editors, whereas other environment generators produce separate programs for each editor.

## IMPLEMENTATION CONSIDERATIONS

Several implementation avenues distinguish DOSE from other structure editing projects. The most significant departure was the interpretive approach, but several interesting alternatives were considered for both the formal description and the internal representation of program information.

### Syntax description

DOSE uses an unusual notation for language description. Most other systems use a form of context-free grammar where each production and class, respectively, have forms similar to those shown in Figure 16. Each type, indicates the name of another

production or class. This style of notation makes it awkward to define optional children and sequences, as illustrated by Figure 17.

```
productionname => type1 type2 ... typen
classname => type1 | type2 | ... | typem
```

*Figure 16. Production in context-free grammar*

```
productionname => type1 type2 optionalchild sequencename ...
productionname => type1 type2 sequencename ...
sequencename => elementtype sequencename
sequencename => <empty>
```

*Figure 17. Sequences and optional components*

DOSE extended the typical syntax description to a notation based on IDL, the Interface Description Language.[17–19] The resulting notation supports names for the children of non-terminal nodes, optional children, children that are sequences, and enumerated sets of terminals. IDL syntax descriptions are more readable and support a more concise semantics description, since sibling components with the same type can be distinguished by name. The optional children and sequence examples are shown in Figure 18 in the notation adopted for DOSE, which was also used for all the earlier examples in this paper. This same style of notation was later adopted for the MacGnome structure editor generator.[20]

```
productionname => name1: type1
                  name2: type2
                  name3: type3 (optional)
                  name4: seq of elementtype
                  ...
```

*Figure 18. DOSE productions*

## Internal representation

As explained above, the most commonly used internal representation for a structure editing environment is the abstract syntax tree. In those systems such as DOSE that go beyond basic syntax-directed editing, the tree is augmented with information related to semantics processing. This is typically done in one of two ways: *attributes* or *invisible components*.

An attribute is simply a name/value pair. Each node has a property list containing an arbitrary number of attributes, where each attribute has a unique name. Each attribute is itself an attributed syntax tree described by a syntax description and (sometimes) a semantic description. Attributes are used by all structure editors whose semantics processing is generated from attribute grammars,[21] for example the Synthesizer Generator, and also by some structure editors whose semantics processing are hand-coded as special routines. In DOSE, both the allocation of attributes and the semantics processing are hand-coded, so attributes are not specified as part of the language description as is done for attribute grammars.

Invisible components work as follows. As discussed above, each non-terminal node has a fixed number of components. Most of these are displayed, in a particular order,

with a specific concrete representation. In general, all components that represent the abstract syntax of the language are displayed so that they may be manipulated by user commands. However, there may be other components that are never displayed. The user is not normally aware of these components, which usually contain semantic information.

Every component, visible or invisible, represents a parent–child link between two nodes. In a tree, each node is the child in at most one parent–child link. However, invisible components may represent parent–child links to nodes that are the child in more than one parent–child link. Such links are called *graph links*. For example, each identifier might have an invisible component that is the definition site of the identifier. This is not a copy of the definition site, but the actual definition site that was constructed elsewhere in the syntax tree. Graph links transform the tree into a general graph structure. Each node may have multiple parents and it is possible to construct cycles. The IPSEN project[22] took this approach, where the graph structures are specified by *graph grammars*.[23] Graph grammars permit the user to explicitly construct graph links.

DOSE initially provided limited support for general graph structures. Unlike IPSEN, the graph links in DOSE were manipulated only during semantics processing and could not be directly constructed or deleted by the user. The goal was to support applications other than programming environments, where the application-specific internal representation was conceptually a network rather than purely hierarchical. The role of display information was expanded from simply providing the concrete syntax and formatting information for the display to specifying the subset of the components of a node that could be viewed. The new extended 'display' formats were used to select subsets of the internal representation for semantics processing as well as for external display. Since multiple display formats could be written for each production of the abstract syntax, it was possible to define different views of the graph structure. The processing support assumed that each view was hierarchical, and DOSE did not detect cyclic views. (More recently, a general theory of views that supports both display and semantics processing has been developed.[24, 25])

Actual experience with the graph version of DOSE demonstrated that the number of graph links was tiny relative to the number of normal parent–child links representing the basic abstract syntax tree. The performance and complexity overhead of the full graph mechanism was not justified, so DOSE's support for general graphs was eventually abandoned in favour of attributed abstract syntax trees. This decision was made after consideration of several factors. One issue is space efficiency. A component is always represented whether or not it has been filled in: a metanode must be present when the component has no value. In contrast, an attribute need not be represented at all if it has no value, or the default value. Another issue is flexibility. Components are hard-wired at the time the language description is written. Attributes are usually defined at the same time, but this is not necessarily the case. Attributes may also be introduced on the fly during execution.[26]

The third issue is conceptual. Certain information is conceptually part of the abstract syntax of the programming language, whereas other information is not. The conceptually separate information may be redundant but in a format that is more suitable for some specific processing. Or it may include information temporarily maintained during program execution. Different notation and realization should be used for the information that is conceptually distinct from the node than for the information that is an integral part of the node.

The final issue is physical. Separation of attribute information from the basic node permits the two kinds of information to be stored independently on disk. This permits, for example, the basic abstract syntax tree to be loaded for display without loading all of the semantic information stored in attributes.

Unfortunately, there were some cases in DOSE where attributes could not provide all the functionality of the earlier graph links. These links were replaced with symbolic references. A *symbolic reference* is a relatively fast mechanism for locating a node elsewhere in the tree. For example, a symbolic reference might be implemented as a key into a table that contains entries for each referenced node. Symbolic references are similar in function to the non-local productions of the PoeGen system.[27]

The change from graph links to attributes and symbolic references drastically improved the performance and decreased the complexity of many parts of the DOSE kernel, including display, tree traversal and the size of the representation for secondary storage. It also simplified the job of the implementor of semantic processing routines, who no longer had to consider views onto the graph structure. There are only a few potential locations for symbolic references, and those are specified as such in the syntax description, so checking is handled by the editor kernel. Attributed trees with symbolic references are sufficient for structure editor-based programming environments, but a general graph structure may be more appropriate for other applications. This style of attributes and symbolic references was later adopted for the ALOE system.

### Terminal node representation

Several other changes to the internal representation supported by DOSE are not visible to either the user or the implementor of a particular editor. For example, it seemed possible to reduce the run-time memory requirements by representing metanodes and terminal nodes in an unusual manner; in most other structure editors, they are represented in the same way as non-terminal nodes, as pointers to records. The original DOSE implementation instead represented metanodes by nil pointers and terminal nodes by their actual values. For example, a terminal node that represented an integer was represented by the integer itself rather than by a record with optype and value fields. Since not all components carried their own optype fields, it was necessary to examine the parent's production to determine first whether each component was a terminal or non-terminal node and secondly, in the case of terminal node, what type of terminal node (integer, boolean, string, and so on). This complicated virtually every kernel routine because non-terminal nodes and terminal nodes had to be handled differently even when the distinction was not relevant to the normal processing performed by the routine. Returning to the more common representation of both metanodes and terminal nodes as records resulted in a uniform representation of nodes, reduced the size of the DOSE kernel substantially, and significantly improved performance.

## STATUS

The DOSE System was developed at Siemens Research and Technology Laboratories in Princeton, NJ beginning in early 1981 and ending in early 1985. Several DOSE environments have been developed, including the editor/interpreter for the grammar grammar, language-based editors with type checking for C and Pascal, a forms editor,

a small database entry/query interface, and an editor/interpreter/debugger for TML, a tree manipulation language for writing semantics routines for DOSE environments. DOSE has been in production use outside the development group since August 1985 to support rapid prototyping of configuration management languages and tools. DOSE has also been used at Carnegie Mellon University to develop an environment for module interface checking and version control.[28]

DOSE was originally written in Perq Pascal for the Perq workstation, first under POS (Perq Operating System) and later under Accent using two successive window managers, Canvas and Sapphire. DOSE was the first structure-editing system to take advantage of a bit-mapped display on a powerful workstation. The Accent/Sapphire version consists of approximately 60,000 lines of source code. DOSE was ported in 1986 to C for the Sun workstation under Sun 3.0.

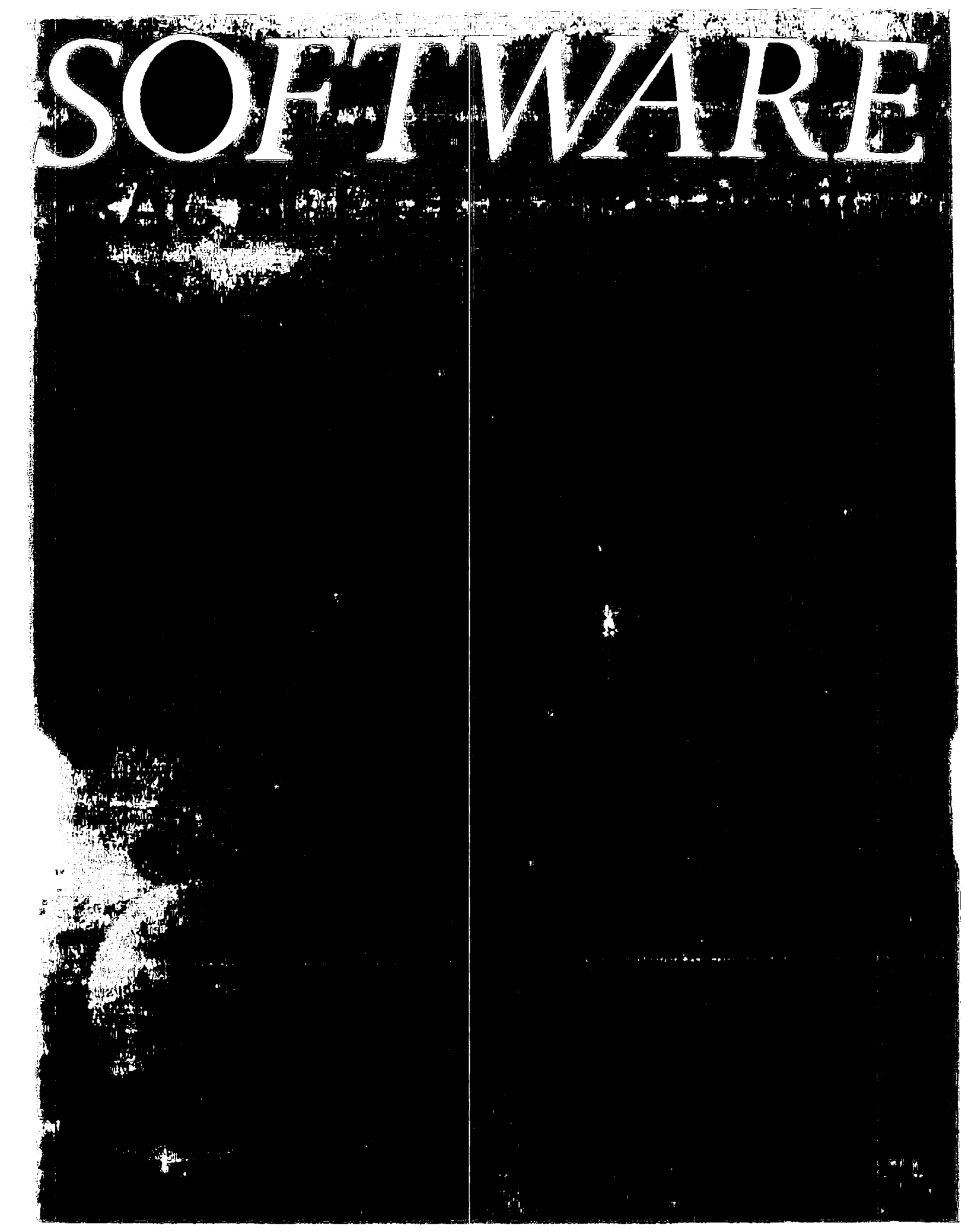
## ACKNOWLEDGEMENTS

In addition to the authors, Larry Engholm, Peter Neuss, Steve Popovich and Bill Smith participated in DOSE's development. DOSE was ported to C/Sun by Mike Edge, Mike Platoff and Scott Vorthmann under the supervision of Bob Schwanke.

## REFERENCES

1. Mark N. Wegman, 'Parsing for structural editors', *21st Annual Symposium on Foundations of Computer Science*, October 1980, pp. 320–327.
2. Joseph M. Morris and Mayer D. Schwartz, 'The design of a language-oriented editor for block-structured languages', *SIGPLAN SIGOA Symposium on Text Manipulation*, Portland, OR, June 1981, pp. 28–33.
3. Peter B. Henderson, 'More on expression tree transformations', *16th Princeton Conference on Information Sciences and Systems*, 1982.
4. Carlo Ghezzi and Dino Mandrioli, 'Augmenting parsers to support incrementality', *Journal of the ACM*, **27**, (3), 564–579 (1980).
5. Wilf R. LaLonde and Jim des Rivieres, 'Handling operator precedence in arithmetic expressions with tree transformations', *ACM Transactions on Programming Languages and Systems*, **3**, (1), 83–104 (1981).
6. Gail E. Kaiser and Elaine Kant, 'Incremental parsing without a parser', *The Journal of Systems and Software*, **5**, (2), 121–144 (1985).
7. Peter Deutsch, Private communication, July 1985.
8. Wilfred J. Hansen, 'User engineering principles for interactive systems', *Fall Joint Computer Conference Proceedings*, 1971.
9. Tim Teitelbaum and Thomas Reps, 'The Cornell Program Synthesizer: a syntax-directed programming environment', *Communications of the ACM*, **24**, (9), 563–573 (1981).
10. Thomas Reps and Tim Teitelbaum, 'The synthesizer generator', *SIGSoft/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 41–48.
11. A. N. Habermann and D. Notkin, 'Gandalf: software development environments', *IEEE Trans. Software Engineering*, **SE-12**, (12), 1117–1127 (1986).
12. Peter H. Feiler and Gail E. Kaiser, 'Display-oriented structure manipulation in a multi-purpose system', *IEEE Computer Society's Seventh International Computer Software and Applications Conference*, Chicago, IL, November 1983, pp. 40–48.
13. Gail E. Kaiser and Peter H. Feiler, 'Generation of language-oriented editors', *Programmierumgebungen und Compiler*, B. G. Teubner, Stuttgart, April 1984, pp. 31–45.
14. Peter H. Feiler, Fahimeh Jalili and Johann H. Schlichter, 'An interactive prototyping environment for language design', *Nineteenth Hawaii International Conference on System Sciences*, Honolulu, HI, January 1986, pp. 106–116.
15. Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn and Bernard Lang, 'Programming environ-

ments based on structured editors: the Mentor experience', in David R. Barstow, Howard E. Shrobe and Erik Sandewall (eds), *Interactive Programming Environments*, McGraw-Hill Book Co., New York, 1984, pp. 128–140.
16. David Notkin, 'The GANDALF project', *The Journal of Systems and Software*, 5, (2), 91–105 (1985).
17. David Alex Lamb, 'IDL: sharing intermediate representations', *ACM Transactions on Programming Languages and Systems*, 9, (3), 297–318 (1987).
18. Richard Snodgrass and Karen Shannon, 'Supporting flexible and efficient tool integration', in Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (eds), *Lecture Notes in Computer Science, Volume 244: Advanced Programming Environments*, Springer-Verlag, Berlin, 1986, pp. 290–313.
19. Peter H. Feiler, 'Relationship between IDL and structure editor technology', *SIGPLAN Notices*, 22, (11), 87–94 (1987).
20. Ravinder Chandhok, David B. Garlan, Dennis Goldenson, Philip L. Miller and Mark Tucker, 'Programming environments based on structure editing: the GNOME approach', in Anthony S. Wojcik (ed.), *1985 National Computer Conference*, AFIPS, Chicago, IL, July 1985, pp. 359–370.
21. Donald E. Knuth, 'Semantics of context-free languages', *Mathematical Systems Theory*, 2, (2) 127–145 (1968).
22. G. Engels, R. Gall, M. Nagl and W. Schafer, 'Software specification using graph grammars', *Computing*, (31), 317–346 (1983).
23. Hartmut Ehrig, Manfred Nagl and Grzegorz Rozenberg (eds), *Lecture Notes in Computer Science, Volume 153: Graph Grammars and their Application to Computer Science*, Springer-Verlag, Berlin, 1984.
24. David Garlan, 'Views for tools in integrated environments', *PhD Thesis*, Carnegie Mellon University, May 1987, CMU-CS-87-147.
25. David Garlan, 'Views for tools in integrated environments', in Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (eds), *Lecture Notes in Computer Science, Volume 244: Advanced Programming Environments*, Springer-Verlag, Berlin 1986, pp. 314–343.
26. Alan Demers, Anne Rogers and Frank Kenneth Zadeck, 'Attribute propagation by message passing', *SIGPLAN '85 Symposium on Language Issues in Programming Environments*, Seattle, WA, June 1985, pp. 48–59.
27. Gregory F. Johnson and Charles N. Fischer, 'Non-syntactic attribute flow in language based editors', *Ninth Annual ACM Symposium on Principles of Programming Languages*, January 1982, pp. 185–195.
28. A. Nico Habermann, 'Automatic deletion of obsolete information', *The Journal of Systems and Software*, 5, (2), 145–154 (1985).


SOFTWARE