EASE -- AN EXTENSIBLE ABSTRACT STRUCTURE EDITOR

by

Deborah A. Baker

CU-CS-250-83 July, 1983

*Department of Computer Science, University of Colorado, Boulder, CO 80309 USA

Abstract

This paper describes EASE, an Extensible Abstract Structure Editor. EASE is Extensible as it is not based on a particular language, nor does it require that the language be defined at the outset. It is a Structure Editor as it builds a derivation tree, not flat text. Finally, it is an Abstract Structure Editor as the trees it builds and maintains may be constructed from an abstract syntax (one devoid of syntactic sugar).

EASE – An Extensible Abstract Structure Editor

Deborah A. Baker

Department of Computer Science
University of Colorado
Campus Box 430
Boulder, Colorado 80309

Abstract

This paper describes EASE, an Extensible Abstract Structure Editor. EASE is Extensible as it is not based on a particular language, nor does it require that the language be defined at the outset. It is a Structure Editor as it builds a derivation tree, not flat text. Finally, it is an Abstract Structure Editor as the trees it builds and maintains may be constructed from an abstract syntax (one devoid of syntactic sugar).

Key Words

syntax-directed editing, programming environments, software development

1. Introduction

This paper describes EASE, a syntax-directed editor. EASE is different from most editors in its genre in that it allows simultaneous development of a language and utterances of that language. That is, it is not a syntax-directed editor for any predetermined, fixed language. EASE is an Abstract Structure Editor as it builds and maintains derivation trees based on an abstract syntax as opposed to either derivation trees based on a concrete syntax or flat text. EASE is Extensible as it is not based on a fixed language. Thus, EASE is an **E**xtensible **A**bstract **S**tructure **E**ditor.

EASE is a first step in building a software engineering environment which will support an evolutionary approach to software development. A software product is traditionally viewed as developing in distinct phases, which are collectively known as the software life cycle. This life cycle framework has proved useful in software engineering; the identification of distinct activities composing the life cycle served as a focal point in addressing the "software crisis" and in establishing software engineering as a field of study. However, the distinction between phases of the software life cycle is somewhat arbitrary; a strict adherence to the separation may serve to obscure the similarities among phases. With the paradigm of evolutionary development, software development takes the form of uniform, small steps, each of which is verified to move the system in the desired direction.

A software engineering environment is a collection of tools which support software development activities. Such a collection of tools should be well integrated to work cooperatively in assisting development. Since the use of a specification language is a common feature of requirements capture and preliminary and detailed design, the convenience of such use becomes an important factor. Our primary goal in building EASE was that it would become a common editor for statements in these various specification languages. By designing EASE without dependence on a fixed, predetermined language, a system designer has the flexibility of a notation and vocabulary that can evolve in step with the design.

EASE was also developed to meet goals common to all syntax-directed editors. The use of a syntax-directed editor prevents the generation of syntactically incorrect text. The use of an editor which allows the user to manipulate the syntactic elements of the language in use will help the user conceptualize the structure of the developing text. EASE was written in C under Berkeley UNIX[1].

The Cornell Program Synthesizer (CPS) [5, 6] is a programming environment for developing (originally) PL/CS programs. Versions for other languages have been produced. The syntax-directed editor of the CPS is different from EASE in several important respects. CPS is used with a fixed, pre-determined language. It is not intended for developing vocabulary and notation simultaneously with statements. CPS provides facilities for compilation, execution and debugging of programs with its use. Thus it is meant for developing the code for a program but not for developing other renditions of it, such as specifications or designs. CPS is similar to EASE in that it does not require that the entire tree be derived, but allows low level syntactic entities (such as expressions) to be entered as text to avoid the tedium of long chains of derivations. Phrases entered in this way must be parsed before being added to the structure tree.

[1]UNIX is a trademark of Bell Laboratories.

The Incremental Program Environment (IPE) [2], part of the Gandalf project [1], is similar to CPS in that it is intended for the development of programs. IPE has a syntax-directed editor and facilities for incremental compilation, program execution and language-oriented debugging. The syntax-directed editor of IPE does require that the entire tree be derived, rather than allowing text to be entered at a low level. An ALOE (A Language Oriented Editor) generator [3] is also part of the Gandalf project. Given a grammar, the ALOE generator produces an ALOE for that language described by the grammar. This is similar to EASE in that an editor could be generated for any language. It is different from EASE in that EASE allows dynamic development of a language concurrently with its use; this is especially important for supporting the interactive development of problem-oriented abstractions.

The remainder of this paper is organized as follows. Section 2 contains an overall description of EASE and more detailed descriptions of each of the modes of operation. Section 3 consists of an example. Conclusions are drawn and the paper is summarized in Section 4.

2. EASE

EASE is a *structure* editor. A structure editor manipulates the syntactic units of the language. Thus, instead of building flat text, a structure editor builds a derivation tree. A derivation tree differs from a parse tree in that it is built directly rather than being the result of a parse. EASE builds abstract derivation trees in which everything that exists in an equivalent flat listing (such as semi-colons) may perhaps not exist in the tree and in which operand/operator relationships are clear. Using the concrete syntax of the language, the corresponding flat text can be generated from the tree.

EASE is also *adaptable* or *extensible*. It is possible to specify and extend the constructs of the language for which this is a structure editor. The language is specified with an extended BNF notation. The grammar is context free, but is not restricted to any particular parsing class such as LL or LR. It may be ambiguous. New productions may be added to the grammar interactively during a derivation.

EASE operates in several modes: grammar, utterance and management. In grammar mode, productions are added to the current grammar and display templates for each production are developed. In utterance mode, phrases in the language defined by the current grammar may be made. (Note that since every nonterminal of an abstract structure corresponds to a meaningful operation, EASE does not assume the existence of a distinguished root symbol.) The usual editing functions for travel and modification are provided. In management mode, the format in which trees and grammars are presented is controlled. There are several distinct types of objects manipulated by EASE: grammars, derivation trees, and user profile information. Though modeless editing is increasingly popular [4] the variety of objects manipulated by EASE has led to the several modes of operation. Another approach would be to used moded menus.

In a typical use of the editor, the system designer may be using a specification language to develop the specifications of a system. The system designer first "views" the existing specification tree. The editor's attention is directed to a particular node in the tree. The designer invokes a particular production and the tree is modified accordingly. The language as defined up to some point may become inadequate. Instead of invoking a command to modify the specification tree, the designer invokes a command to add a grammar production.

2.1. Grammar Mode

The language for which EASE is a structure editor is determined by the current grammar. A grammar is a set of productions. Each production has three parts: a name, a left hand side (a nonterminal) and a right hand side (a sequence of nonterminals and semantically significant terminals such as constants and identifiers). Once a parser is introduced to allow low-level syntactic entities to be entered directly rather than derived, it will likely impose some restrictions on the grammar, depending on the parsing technique used.

Associated with each grammar is one or more sets of display templates; each set of display templates has one display template for each grammar production. The display templates supply both the syntactic sugar necessary for a concrete syntax and the format in which instances of a production are to be shown to the user.

In Figure 1 there is a grammar production for an abstract data type, a possible display template for the production, the abstract syntax defined by the production, the concrete syntax defined by the production and the template, and, finally, the display of an instance of an abstract data type. The notation $\langle\text{nonterminal}\rangle_n$ is a shorthand for "at least n occurrences of <nonterminal>". In particular, $\langle\text{nonterminal}\rangle_0$ corresponds to the Kleene star and $\langle\text{nonterminal}\rangle_1$ to plus. The convention is to list multiple occurrences of a nonterminal one under another. A further display convention is that a nonterminal appearing once on the right hand side of a production may appear more than once in the display template; the instances of a nonterminal appearing more than once must be disambiguated.

The deletion of a production from a grammar is not allowed in EASE once utterances have been built using the grammar. If there were a derivation tree depending on a production that is deleted, that derivation tree would become invalid. Another approach to avoiding the problem of invalid trees would be to record which productions of a grammar have actually been used, and only allow deletion of those that have not been. A third approach would be to keep a count of dependencies on each production and allow deletion of those productions whose dependency count is zero. Either of these last two approaches would be more flexible if utterances were associated with versions of a language.

2.2. Utterance Mode

In utterance mode, EASE allows a user to build, traverse and maintain abstract derivation trees. A derivation tree is similar to a parse tree. Derivation trees, however, are constructed, while parse trees result from the application of a parser to a statement in the language. Each internal node of a derivation tree corresponds to what will be called an expanded nonterminal symbol. Each leaf node corresponds to either a terminal symbol or what will be called an unexpanded nonterminal symbol. An expanded nonterminal symbol is one for which a further derivation has been made by the application of an appropriate

```
Production:
  Name: adt
  Left Hand Side: <abstract data type>
  Right Hand Side: <name> <sort>$_1$ <signature>$_1$ <equation>$_0$

Display Template:
  <name> = abstract data type with
    sorts
      <sort>$_1$
    signatures
      <signature>$_1$
    equations
      <equations>$_0$
  end <name>

Abstract Syntax:
  <abstract data type> ::= <name> <sort>$_1$ <signature>$_1$ <equation>$_0$

Concrete Syntax:
  <abstract data type> ::= <name> = abstract data type with
                  sorts <sort>$_1$
                  signatures <signature>$_1$
                  equations <equation>$_0$ end <name>

Instance:
  stack = abstract data type with
    sorts
      {list of sorts of type stack}
    signatures
      {list of signatures of type stack}
    equations
      {list of equations of type stack}
  end stack
```

Figure 1

production. An unexpanded nonterminal symbol is one for which no further derivation has been made. The children nodes of an internal node are the tokens on the right hand side of the production used to expand the node. The parent node of a node corresponds to the left hand side of the production from which the node was derived.

Each node in a derivation tree has seven fields: pointers to its parent, leftmost child, left sibling, and right sibling, a nonterminal or terminal symbol, a production name, and a label (see Figure 2). The sibling pointers may be nil. The leaves of the derivation tree correspond to unexpanded nonterminal symbols. The leftmost child pointer is therefore nil for leaves of a tree. The parent pointer will be nil only for the root node of a tree. The nonterminal/terminal field is mandatory. If it contains a nonterminal, that nonterminal is the left hand side of every production that may be used to expand it. For internal nodes, the production name field contains the name of the production used to expand the nonterminal. For leaf nodes, the production name field is empty. For nodes that have been explicitly named, the label field contains that name; otherwise it is empty.

Editors typically operate with respect to a current position. In EASE this current position is a current node. If the current node is unexpanded (a leaf), an inquiry may be made so that EASE lists all the currently applicable productions. An applicable production is one whose left hand side is the same as the nonterminal in the current node. In Figure 3 there is an example of an unexpanded node whose nonterminal is <abstract data type>. Suppose the production named adt from Figure 1 is chosen from the list of applicable productions. The result of the expansion is shown in Figure 4.

Travel functions in EASE operate with respect to the current node. The editor's attention is directed to another node which becomes the new current node. Most of the travel functions provided by EASE are tree-oriented. The editor's attention can be directed to the root of the tree or the parent, leftmost child, or left or right sibling of the current node. The editor's attention can also be directed to a node which has previously been given a label (name). Finally, the editor's attentions can be directed by searching for a given terminal, nonterminal or production.

There are several ways a modification of a derivation tree may be accomplished

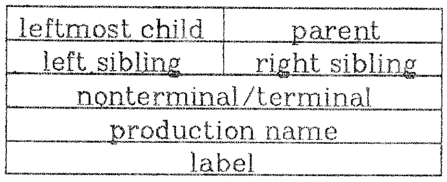

Figure 2

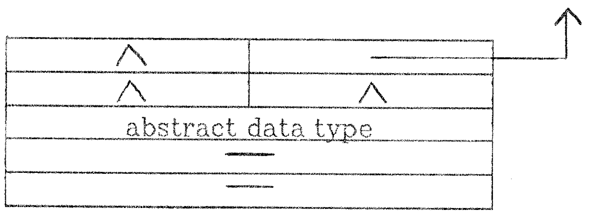

Figure 3

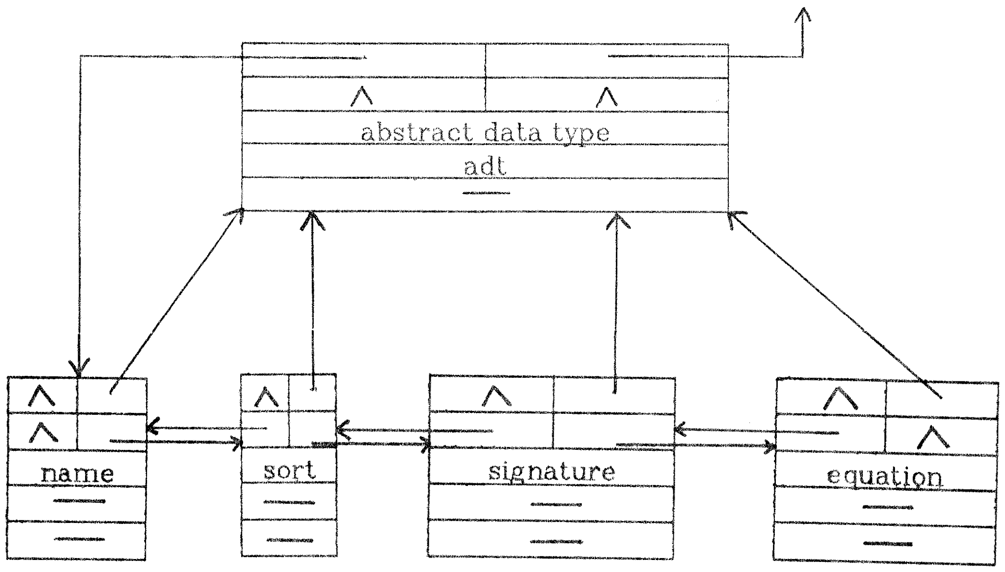

Figure 4

in EASE. The most basic modification mechanism is to expand an unexpanded nonterminal by the application of an appropriate production. Subtrees may be clipped, named and saved and then grafted elsewhere in tree, provided that the nonterminal at the root of the tree is the same as the nonterminal at the leaf node where the graft is to take place. Named subtrees may be copied or moved. An expanded node may be unexpanded or collapsed by deleting or moving the subtree of which the node is a root.

2.3. Management Mode

In management mode, the user of EASE manipulates what could be termed "user profile" information. If a grammar has more than one set of display templates defined for it, the user may choose among them. Also under user control are the number of levels of the tree above and below the current node that are to be displayed. This is a measure of the amount of detail that is to be displayed.

The operations of management mode are perhaps the most lacking in the current version of EASE. Planned additions are operations to set and change key bindings, to query EASE about the name and creation date of the current grammar during an editing session, and to query EASE about the utterances associated with a given grammar. Another planned addition is the option to retain or reset the current locus when leaving each mode.

3. An Example

In this section EASE is used to develop a specification language for abstract data types and to specify the type stack of integers in the developing language. In this case, the type stack of integers is an utterance in the specification language. For the sake of brevity in this example, the specification language will be small and simple, and the descriptions of mode switching, naming productions, and supplying display templates will be suppressed. Detailed descriptions of the specification tree being built will also be suppressed; the specification tree will be shown as formatted text that might result from a particular set of display templates. The system designer might immediately enter the following initial production, knowing that an abstract data type has a name and sets of sorts, signatures and equations.

<abstract data type> ::= <name> $\langle\text{sort}\rangle_1$ $\langle\text{signature}\rangle_1$ $\langle\text{equation}\rangle_0$

The following productions might then be defined.

```
<name> ::= <identifier>
<sort> ::= <identifier>
<signature> ::= <operation> <domain> <range>
<operation> ::= <identifier>
```

At this point, the designer might begin specifying the type stack of integers. The following specification is the one that exists after the initial node is expanded and the name of the data type is filled in. The display template used is from Figure 1.

```
stack of integers = abstract data type with
  sorts
    <sort>₁
  signatures
    <signature>₁
  equations
    <equations>₀
end stack of integers
```

There are several design paths that the designer might now follow. The grammar could be expanded (there are no expansions for <equation>, <domain> or <range> yet), the sorts of the type could be given, or, as shown below, the names of the operations could be given.

```
stack of integers = abstract data type with
  sorts
  signatures
    empty: <domain> -> <range>
    push: <domain> -> <range>
    pop: <domain> -> <range>
    top: <domain> -> <range>
    isempty: <domain> -> <range>
  equations
end stack of integers
```

The designer could next define expansions for <domain> and <range> and then continue the definition of stack by finishing the signature and supplying the sorts.

```
<domain> ::= <sort>0
<range> ::= <sort>

stack of integers = abstract data type with
  sorts
    boolean
    integer
    stack
  signatures
    empty:  -> stack
    push: stack x integer -> stack
    pop: stack -> stack
    top: stack -> integer
    isempty: stack -> boolean
  equations
end stack of integers
```

Finally, <equation> must be defined, and then the stack of integers may be completed.

```
<equation> ::= <left side> <right side>
<left side> ::= <term>
<right side> ::= <term>
<term> ::= <operation> <arguments>
<term> ::= <identifier>
<arguments> ::= <term>
<arguments> ::= <identifier>1

stack of integers = abstract data type with
  sorts
    boolean
    integer
    stack
  signatures
    empty:  -> stack
    push: stack x integer -> stack
    pop: stack -> stack
    top: stack -> integer
    isempty: stack -> boolean
  equations
    isempty(empty) = true
    isempty(push(s,i)) = false
    pop(empty) = error
    pop(push(s,i)) = s
    top(empty) = error
    top(push(s,i)) = i
end stack of integers
```

4. Conclusions

This paper presents EASE, an extensible abstract structure editor. This system gives the system designer the advantages of a syntax-directed editor as well as the flexibility to build the language in use. However, a software tool is almost useless without a suggested discipline of use. It is expected that a stable methodology being supported by EASE will result in a relatively permanent language. This is analogous to a stable system having stable specifications. Until a particular application is sufficiently well understood, the language in use to specify programs in that application will tend to change periodically. For a particular project, it could be that the language is specified at the outset by an EASE expert. EASE would then appear to be an abstract structure editor with a fixed language.

Display templates in EASE are basically semantic valuations which compute signals to drive display devices. In order to turn EASE into a more comprehensive tool for software development, however, it must be interfaced with a more powerful semantic metalanguage. This will allow more sophisticated semantic views such as (in the case of programming languages), execution, data flow, and verification condition profiles. As an example of the anticipated usefulness of such a tool, note that interactive, symbolic debugging can be accomplished by opening two windows on a program, one showing an execution profile and the other the source text.

The current version of EASE is a prototype. The Environments Research Group at the University of Colorado intends to experiment with it and continue its development. Some planned extensions to EASE were discussed in Section 2.3. Further extensions include a graphics interface and macro capabilities.

Acknowledgments

Hal Eden, David Gill, Hell Gill and Jon Shultis were instrumental in originating and developing EASE. This project could not have been accomplished without the aid of the Software Development Workshop at the University of Colorado, Boulder. The participants of the workshop were Craig Adler, Richard Cobb, Robert Kambic, Corning Lahey, Michael Schweitzer and Mark Wagner. Special thanks to Rick Cobb for his continuing interest in the development of EASE and to Hal Eden, Helen Gill and Jon Shultis for their comments on early versions of this paper.

References

1. A. N. Habermann, *An Overview of the Gandalf Project*, Computer Science Research Review, Carnegie-Mellon University, Pittsburgh, Pa. (1979).
2. R. Medina-Mora and P. H. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering* **SE-7**(5) pp. 472-482 (Sept. 1981).
3. R. Medina-Mora and D. S. Notkin, *ALOE Users' and Implementors' Guide*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (Nov. 1981).
4. N. Meyrowitz and A. van Dam, "Interactive Editing Systems: Parts I and II," *Computing Surveys* **14**(3) pp. 321-415 (Sept. 1982).
5. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* **24**(9) pp. 563-573 (Sept. 1981).

6. T. Teitelbaum, T. Reps, and S. Horwitz, "The Why and Wherefore of the Cornell Program Synthesizer," *Proceedings of the ACM Symposium on Text Manipulation*, pp. 8-16 (June 1981).