N87-28303

P15

TREE-ORIENTED INTERACTIVE PROCESSING WITH AN APPLICATION TO THEOREM-PROVING

David Hammerslag

Samuel N.Kamin

Roy H.Campbell

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

June, 1985

Tree-Oriented Interactive Processing with an Application to Theorem-Proving

David Hammerslag
Samuel N. Kamin
Roy H. Campbell

ABSTRACT

This paper describes our concept of "unstructured structure editing" and ted, an editor for unstructured trees. Ted is used to manipulate hierarchies of information in an unrestricted manner. The tool has been implemented and applied to the problem of organizing formal proofs. As a proof management tool, it maintains the validity of a proof and its constituent lemmas independently from the methods used to validate the proof. It includes an adaptable interface which may be used to to invoke theorem provers and other aids to proof construction. Using ted, a user may construct, maintain, and verify formal proofs using a variety of theorem provers, proof checkers, and formatters.

Keywords
Theorem Proving, proof management, structure editing, tree editing

1. INTRODUCTION

The manipulation and maintenance of detailed information in an organized manner is a problem in software engineering projects. This paper describes a management tool that aids the construction, modification, and maintenance of hierarchies of information. The tool has been implemented and applied to the problem of maintaining formal proofs.

The tool is based on a tree editor which organizes and manipulates hierarchies of text, program, or data. As a proof management tool, it maintains the validity of a proof and its constituent lemmas independently from the methods used to validate the proof. It includes an adaptable interface which may be used to to invoke automated proof methods. Using the tool, a user may construct, maintain, and verify formal proofs using a variety of theorem provers, proof checkers, and formatters.

1.1. Structure Editing

Our approach can perhaps best be described by the phrase "unstructured structure editing." Our editor allows constrained hierarchies of information to be edited but does not impose any restrictions on the editing process itself. We can explain our approach by analogy with syntax-oriented program editors [Cam84], [Tei81], [Fis84], and [Don80][1]. We see program editors as being of two types:

- These are the traditional editors, which are marked by flexibility but little power for editing structured data such as programs. For example, the operation *place* **begin** and **end** *around this statement* is not readily accomplished, because the editor has no notion of what a *statement* is. It is important to note that there is a lot of structure in the text, but that structure is not used until compilation time.

- The newer "syntax-directed" editors have knowledge of the structure being edited, and strive to maintain that structure at all times. These editors facilitate structure-oriented operations, but are generally characterized by inflexibility. For example, it is difficult to transform a while loop to a repeat loop, because this involves changing the type of statement and also interchanging the two components (Boolean expression and statement) of the statement; in whichever order these are done, the tree is temporarily in an inconsistent state.

We want particularly to emphasize that structure editors have taken *two* steps away from traditional editors, only one of which we feel is helpful:

(1) **Trees** are edited (or possibly a mixture of trees and text) rather than just **text**. Since programs have a natural tree structure, this is useful.

(2) The structure which the program must possess when editing is done, it must in effect possess throughout the editing process. In traditional text editors, it is the user's

[1] We have in fact produced a prototype program editor based on our ideas, which is described in section 5.3. However, most of our work has been done on proof editing. Our point in giving this example is to explain our structure editing philosophy in a more familiar setting.

responsibility to construct a program which is syntactically correct; the editor does not "look over his shoulder" as he is editing. Yet syntax-directed editors do not trust the user to construct a valid tree, and impose constraints on what the user can do to ensure that the tree is in a valid state *at all times.*

We believe, and our tree-editor to some extent demonstrates, that it is possible to move from text-editing to tree-editing without making the editing process any more constrained than it is in text editors.

Our editing approach exploits the manipulation of abstractions without imposing the constraints of a template editor. The editor manipulates the abstract structure without verifying that the detailed syntax and semantics are correct. Further tools are used to verify these details. A program editor based on our tree editor permits the creation of proper syntactic structures, but never *requires* syntactic correctness. Each node in the abstract syntax tree can be individually validated at the user's request. To change a while loop to a repeat loop, the order of the children would be reversed (a simple tree-editing operation), the parent node would be changed from *while* to *repeat* and the "node validator" would be invoked to verify its syntactic and semantic correctness.

2. APPLICATIONS TO THEOREM PROVING

We have constructed an editor, ted, for unstructured tree editing. The editor is more fully desribed in section 3. In brief, ted maintains an internal tree structure which is edited by the user via commands such as t (copy a node at another place in the tree), t* (copy a sub-tree at another place in the tree), e (edit the contents of a node), and m* (move a sub-tree to another place).

The editor was originally designed for use with proof trees. The editor is used to create proof trees, and external programs, such as automatic theorem provers, are used to certify that the tree created is, in fact, a proof tree.

2.1. An Example

As an example, consider one of the first (and easiest) theorems proved in our system, cancellation on the left in a group; that is that for all a, b, and c, $ab = ac \rightarrow b = c$. Initially we tried to prove this directly from the axioms for a group. Figure 1 depicts the tree we initially tried[2]. Unfortuately, the theorem provers being employed were not able to verify this fact in a reasonable amount of time, so the problem was decomposed into two lemmas, the children of the root in Figure 2. Each of these lemmas was verified by the theorem prover, and finally, the theorem was proved by using the two lemmas. Note that in the final tree, the proof of the theorem does not use the axioms concerning group theory, rather the supporting lemmas are assumed to be true, and the validity of the inference is checked.

[2] These figures are not meant to show how trees are displayed in our system, but rather to represent to tree structure being discussed

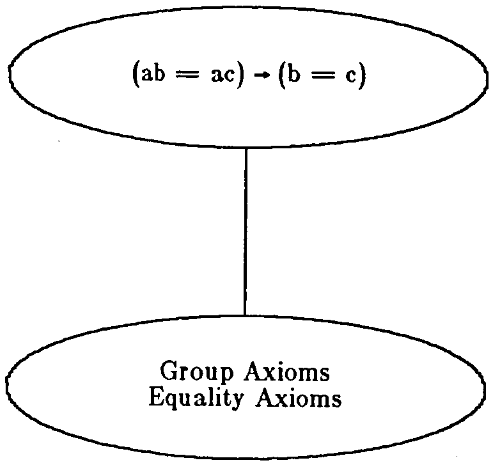

Figure 1.

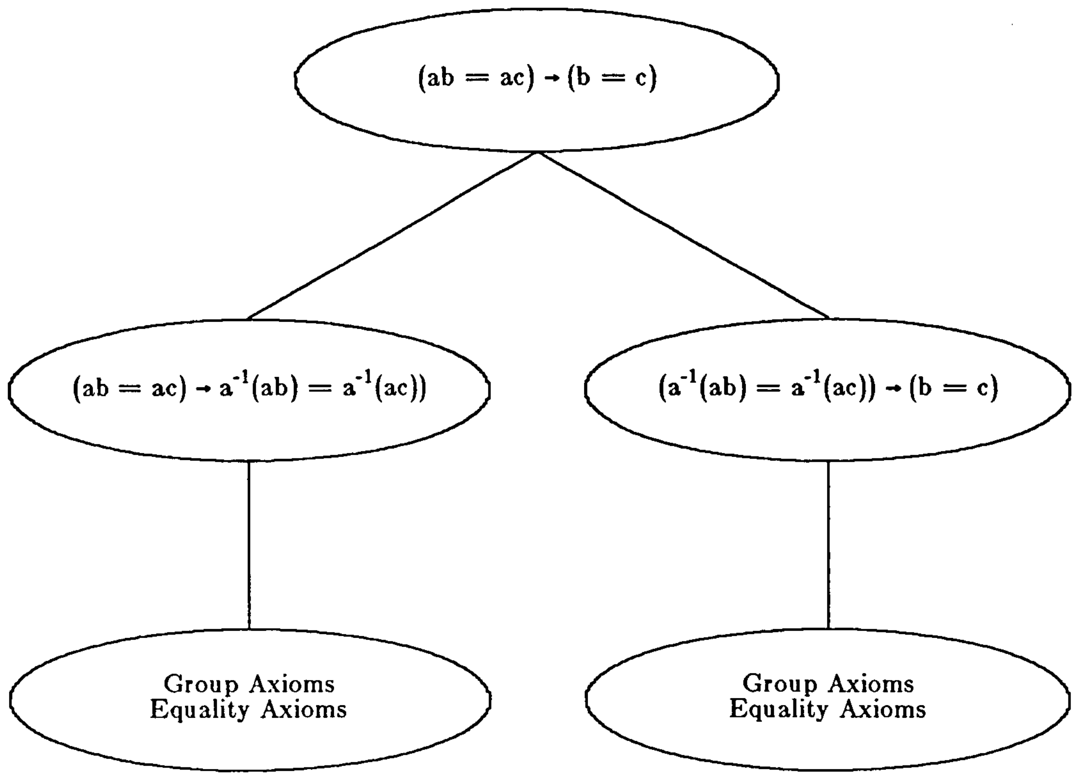
$(ab = ac) \rightarrow (b = c)$
$(ab = ac) \rightarrow a^{-1}(ab) = a^{-1}(ac))$
$(a^{-1}(ab) = a^{-1}(ac)) \rightarrow (b = c)$
Group Axioms
Equality Axioms
Group Axioms
Equality Axioms

Figure 2.

2.2. Related Work

Before the editor was built, other systems (those with well-documented interfaces) were investigated [Wey77, Wey74], [Ble83,Ble73], [Gor79], [Ger79, ISI79], and [Rep84]. The systems examined can be divided into two groups: those that explicitly maintain a proof tree, and those that do not. For constructing formal proofs, a system which does not retain a proof tree leaves the user disadvantaged in a number of ways. If the user becomes lost in a proof, there is no way to retrace the steps taken. There is no way to reconstruct the proof when proving a similar or related theorem. Among systems in which an editable history is not kept are LCF [Gor79] and the UT interactive prover [Ble83,Ble73]. The UT prover makes use of user interaction to make a theorem prover faster and more efficient. However it prover provides no real flexibility: all the user can do is attempt to guide the prover down the proper path. In Edinburgh LCF, a user can write procedures which map formulas or theorems to theorems. The user defines "tactics" which are applied, under the user's control, to the proposition to be proved. While LCF provides the power of programability, it is very easy to "become lost" in a proof. The user must often resort to naming many intermediate results just so that they can be referred to later.

Three systems which actually allow the user access to the tree being created were investigated: Affirm [Ger79,ISI79], FOL [Wey77,Wey74], and Reps' interactive proof checker [Rep84]. Although these systems allow the user to, in some sense, consult the tree, they still lack the flexibility afforded by general purpose tree editing. Affirm, a specification and verification system, includes a method for doing interactive proof development; proof tree management is emphasized in the Affirm literature. Although the user of Affirm is free to view the tree, the tree can only be manipulated by asking the system to make a specific (legal) transformation. This prevents the user from re-using parts of the tree or parts of other proof trees, or modifiying a proof by making small changes in each node. In FOL, the user inputs a rule of inference with associated parameters, where some of the parameters are usually line numbers referring to previously checked proof steps. While FOL provides a very versatile methodology for referencing previous lines, the user is still restricted to proving theorems in a bottom up manner: any proposition being used on a deduction step needs to have been previously proven. A proof checking editor has been implemented by Reps and Alpern using the Cornell Synthesizer Generator. When using the editor, the user edits program source code with imbedded assertions for the partial correctness proof in a Hoare-style logic. As the user edits the program and assertions, the editor checks the correctness of the program and proof; program fragments without valid proofs are highlighted. The interactive proof checker is very close to our approach in spirit: a user is free to move about in the tree, and proofs of supporting lemmas can be tried in any order. However, in the interactive proof checker, the proof tree is really a derivation tree for the proof in a proof language; the user has no direct access to the tree, and it can only be indirectly modified by a reparse.

3. EDITOR OVERVIEW

The prototype unstructured tree editor is written in Franz Lisp [Fod83]. The intent was to explore the uses of this form of tree editor for proof management (and possibly other uses). To make the editor easier to use, a familiar command set was desirable. Consequently, much of the editor addressing scheme, as well as the commands provided and the basic command structure, is based on that of the editor ex [Joy80]. The editor, called ted, is presented in the following manner. First a brief description of the trees being edited is given, then the method of addressing used in the editor is described. Following that the editor commands are presented.

The editor is used to create, modify and maintain proof trees in the system. Throughout the development of the editor, it was kept as general purpose as possible; that is, whenever possible, no assumptions were made concerning the interpretation to be placed on the trees being edited. As might be expected, this turned out to be difficult, and unresolved problems remain.

3.1. Tree Structure

The editor edits arbitrary trees. Each node in a tree contains an element of text and a status element. Because the editor is ignorant of the content of a node, the user is free to manipulate the node text at any time. The status of a tree node is used to indicate the consistency of the structure begin edited and access to it is restricted.

3.2. Tree Addressing

Structural addressing, similar to that used in Mentor [Don80], is used to reference individual nodes and subtrees within the tree being edited. Each node's address is dependent only upon its position in the tree at any given time. The address of a node is derived by starting at the root (denoted by "/") and, for each link traversed in getting to the desired node, appending the number of the child that that node represents to the address. In addition to using the full address of a node, a node may be specified by a base address and a combination of address offsets. There are three base addresses: the root, the current address, and address variables. The root is just "/". The previous address, the current address, and stored addresses are referenced as in ex.

In addition to base addresses, address offsets can be used to specify an address relative to some other address. The offsets are: a digit, "^", ">", and "<". A digit *n*, where $1 \leq n \leq 9$, addresses the *n*th child of a node. "^" refers to the parent of a node. ">" and "<" denote the right and left siblings of a node, respectively. For example, "1>1^" represents the second child of the current node (that is the parent of the first child of th right sibling of the first child), which may also be written simply as "2."

3.3. Editor Commands

The syntax of commands is: [tree address][command][tree address], with all parts optional. Each command has a default address (as in ex, usually ".", but "/" for w). The default command is an abbreviated print. Unless otherwise noted, the commands are the tree analogue to the corresponding ex command. There are, however, two exceptions: 1) the *'ed versions of commands refer to entire sub-trees, while the un-*'ed versions refer to single nodes; 2) in commands that have a target address (*t*,*m*), the node or sub-tree is inserted into the tree in such a way that the target address becomes the address of the new node or sub-tree. The editor commands are (default addresses in parentheses):

(†)a add a new node. † is the rightmost son of the current node.

(.)c <external prog>
invoke an external program.

(.)d, (.)d*

(.)e Call the node editor.

f [<filename>]

h [<cmd>]
help.

(.)k <letter>

(.)m<address>, (.)m*<address>

(.)p print the data at this node.

(.)p*[n]
  print the "first line" of every node in the subtree down to level n (previous n if n is omitted).

q

(.)r [<filename>]

(.)s <status>
  set the status of a node.

(.)t<address>, (.)t*<address>

(/)w [<filename>]

(.)=

4. A Detailed Example

In this section we present transcript of an editor session. The session shows the creation and validation of the proof tree discussed in section 2.

The following is an example of editor use for theorem proving. The example used is exactly that given in section 2.1. When the edit session starts the tree is that of figure 1. Then the axioms are deleted and the two (previously proven) lemmas are added to the tree as children of the root, giving the tree of figure 2. Finally, the proof is completed.

In the node formulas, "A" is universal quantification. It is followed by a list of quantified variables and then the formula in which they are quantified. Commands typed by the user appear in **boldface**, the text in *italics* is comment added to help explain effects of the command, all other text is output by the editor

```
% ted treel
ted version 0.2
```

First the entire tree is displayed. Note that the status is "unproven."

```
⊦ p*
/ : unproven
 (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))

   /1 : AXIOM
    (A (x y z) (= (* x (* y z)) (* (* x y) z)))

   /2 : AXIOM
    (A (x) (= x x))

   /3 : SIMP
    (A (x) (= (* x id) x))
```

We now show the content of each of the axiom (and simplification) nodes

```
⊦ /1p
/1 : AXIOM
  (A (x y z) (= (* x (* y z)) (* (* x y) z)))

⊦ >p
/2 : AXIOM
  (A (x) (= x x))
  (A (x y) (IMP (= x y) (= y x)))
  (A (x y z) (IMP (AND (= x y) (= y z)) (= x z)))
  (A (x y) (IMP (= x y) (= (inv x) (inv y))))
  (A (w x y z) (IMP (AND (= w x) (= y z)) (= (* w y) (* x z))))

⊦ >p
/3 : SIMP
  (A (x) (= (* x id) x))
  (A (x) (= (* id x) x))
  (A (x) (= (* (inv x) x) id))
  (A (x) (= (* x (inv x)) id))
```

We now return to the root of the tree, delete the children just shown, and add the two lemmas as children.

```
⊦ ^
/ : unproven
  (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
        & & &
⊦ 3d
⊦ 2d
⊦ 1d
⊦ 1 r lemma1
```

we have now read in the first lemma, and take a look to see what's there.

```
⊦ p*
/1 : (c pv1)
  (A (a b c)
    (IMP (= (* a b) (* a c)) (= (* (inv a) (* a b)) (* (inv a) (* a c)))))

    /11 : AXIOM
      (A (x y z) (= (* x (* y z)) (* (* x y) z)))

    /12 : SIMP
      (A (x) (= (* x id) x))

    /13 : AXIOM
      (A (x) (= x x))

⊦ >r lemma2
```

Now the same for the second lemma.

```
⊦ p*
/2 : (c pv1)
```

```
(A (a b c) (IMP (= (* (inv a) (* a b)) (* (inv a) (* a c))) (= b c)))

  /21 : AXIOM
   (A (x y z) (= (* x (* y z)) (* (* x y) z)))

  /22 : SIMP
   (A (x) (= (* x id) x))

  /23 : AXIOM
   (A (x) (= x x))
```

Finally, we return to the root and validate the proof by invoking one of the provers, pv1.

```
⊢ ^
/ : unproven
 (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
     & &

⊢ c pv1
Skolemizing:
Pri: 6 7 10 7 [3]
---------------------------------------------
Stored (Non_input + input = total):  3 + 4 = 7
1 proof, tree size: 6
PROOF:
 3 CONTRADICTION    <- (7 1)
Time (sec):    CPU: 0.45     GC: 0.0     Total: 0.45
```

The proof is successful, so the root's status has been updated to show that the node was certified by calling pv1.

```
⊢ p
/ : (c pv1)
 (A (a b c) (IMP (= (* a b) (* a c)) (= b c)))
```

The tree representing the completed proof is written to the current file.

```
⊢ w
⊢ q
%
```

5. DISCUSSION

This section describes work that has been done using the editor as a front-end for theorem proving and other applications to and extensions to unstructured structure editing.

5.1. Experience

The editor has been used extensively as a front-end for theorem provers. The editor was originally conceived of as a theorem prover front-end; a number of proofs have been completed using the editor in this capacity.

Sam Kamin and Myla Archer have successfully used the editor to do proofs in group theory and category theory. David J. Carr has been a major user of the proof system, and has used it to prove the homomorphism conditions necessary to show the implementation of the tree-address data type correct with respect to a final algebra specification [Kam83] of that data type. (A further discussion of ted and proving specifications follows below.) Finally, Carol Beckman and David Hammerslag have used the system to prove (most) of the verification conditions for a program from the Basic Linear Algebra Subprogram package.

5.2. Other Applications

As the editor was designed to be application independent, it has been used in other applications. The editor was modified to manipulate trees that represent hierarchical information and to edit abstract syntax trees for a simple programming language.

5.2.1. Browse

The information tree editor, called browse, was created to allow users to peruse a hierarchy of information about programs, narrowing the class of programs under consideration as he moves down into the tree structure. Each internal node in the information tree contains information about a related group of programs; in general, a child contains information about a subset of those programs encompassed by the node's parent. Each leaf node in the tree contains information for a single program.

Except for re-writing the editor's output routines to reflect the new interpretation of node text, only minor changes were necessary (*e.g.* the *locate* command was re-written to search for nodes containing certain strings in their text). Browse was used to create and maintain a database of information concerning user contributed software.

5.2.2. FASE

Ted was originally developed to provide user-friendly access to theorem-provers, for use in program verification. As such, it is connected to a program specification system called **fase** [Kam83]. These connections, which we hope to strengthen in the future, include:

Syntax

fase has been designed to allow for user-defined syntax for discussing modules appearing in the program specification. This syntax can also be used in ted nodes to state properties of the modules to be proven.

Theory

Ted proofs inherit axioms relating to modules from the **fase** specifications. At present, this is done manually, but we expect to have it automated in the near future. In the same way, proofs of module implementations are based upon **fase** specifications and the associated formalism.

Thus, ted is really just one major component of a system which includes (executable) program specifications and program proofs. Eventually, we hope to expand this into a program development system with the inclusion of program-editing and execution facilities (based upon the peg model), and program transformations.

5.2.3. Peg

A third use of tree editing which has been explored is program syntax tree editing. The editor, peg, allows the user to construct programs by either explicitly constructing abstract syntax trees or by filling a node with source code for the language. In the system, the user is provided with tools to transform text into an abstract syntax tree and to compress syntax trees back into text. Each node in the tree is labeled to indicate which syntactic entity the node or subtree represents. As is the case with the proof system, the editor keeps track of which nodes have been invalidated.

Although peg has only been implemented for a very restricted subset of the programming language C, we feel that it demonstrates the advantages of unstructured syntax tree editing over the syntax directed editing schemes discussed in section 1.

5.3. Conclusion

Our experience in applying our philosophy of structure editing has been favorable. We have completed and used extensively a prototype system for managing formal proofs and have experimented with other applications. We feel that as this system is made more general (that is, less dependent on the application domain) and is given a better user front end more uses will be found for unstructured structure editing.

REFERENCES

[Ble73] W. W. Bledsoe and Peter Brunell, "A Man-Machine Theorem Proving System," *Third IJCAI,* pp. 56-65 (1973).

[Ble83] W. W. Bledsoe, "The UT Interactive Prover," The University of Texas, Austin, Austin (April 1983).

[Cam84] Roy H. Campbell and Peter A. Kirlis, "The SAGA Project: A System for Software Development," *SIGLPAN Notices* **19** pp. 73-80 (May 1984).

[Don80] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structure Editors: the MENTOR Experience," INRIA, France (May 1980).

[Fis84] C. N. Fischer, Anil Pal, Daniel L. Stock, Gregory F. Johnson, and Jon Mauney, "The POE Language-Based Editor Project," *SIGPLAN Notices* **19** pp. 21-29 (May 1984).

[Fod83] John K. Foderaro, Keith L. Sklower, and Kevin Layer, "The FRANZ LISP Manual," University of California Berkeley, Berkeley, California (June 1983).

[Ger79] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. W. Erickson, R. L. London, D. G. Taylor, and D. S. Wile, "An Overview of Affirm: A Specification and Verification System," USC Information Sciences Institute, Marina del Ray, Ca (November 1979).

[Gor79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth, *Edinburgh LCF,* Springer-Verlag Lecture Notes in Computer Science, No. 78, New York (1979).

[Joy80] William Joy, "Ex Reference Manual Version 3.5/2.13," Computer Science Division, University of California, Berkeley, Berkeley, California (September 1980).

[Kam83] Samuel N. Kamin, Stanley Jeffeson, and Myla Archer, "The Role of Executable Specifications: The FASE System," *Proc. IEEE Symposium on Application and Assessment of Automated tools for Software Development,* pp. 105-114 (November 1983).

[Rep84] Thomas Reps and Bowen Alpern, "Interactive Proof Checking," *Proceedings of Eleventh ACM Symposium on Principles of Programming Languages.,* pp. 36-45 (January 1984).

[ISI79] ISI Research Staff, "Program Verification: Annual Report, Fall 1979," USC Information Sciences Institute, Marina del Ray, California (1979).

[Tei81] Tim Teitelbaum and Thomas Reps, "The Cornell program synthesizer: a syntax directed programming environment," *Comm. ACM* **24** pp. 563-573 (Sept 1981).

[Wey74] Richard Weyhrauch and Arthur J. Thomas, "FOL: a Proof Checker for First-Order Logic," Stanford Univesity, Computer Science Department, Stanford, California (September 1974).

[Wey77] Richard Weyhrauch, "A Users Manual for FOL," Stanford University, Computer Science Department, Stanford, California (July 1977).