Interactive Proof Checking

Thomas Reps and Bowen Alpern
Cornell University

Abstract

Knowledge of logical inference rules allows a specialized proof editor to provide a user with feedback about errors in a proof under development. Providing such feedback involves checking a collection of constraints on the strings of the proof language. Because attribute grammars allow such constraints to be expressed in a modular, declarative fashion, they are a suitable underlying formalism for a proof-checking editor. This paper discusses how an attribute grammar can be used in an editor for partial-correctness program proofs in Hoare-style logic, where verification conditions are proved using the sequent calculus.

1. Introduction

This paper concerns the design of an editor for partial-correctness program proofs in Hoare-style logic [Hoare 1969], where verification conditions are proved using the sequent calculus [Gentzen 1935]. The chief innovation in the editor's design is that proofs are treated as objects with constraints on them. The editor keeps the user informed of errors and inconsistencies in a proof by reexamining the proof's constraints after each modification to it. This treatment of proof checking is analogous to the treatment of arithmetic-dependency checking in a spreadsheet system [Bricklin & Frankston 1979]; after each modification, the constraints of the system are reexamined, and changes are updated in the display.

The implementation approach used in the proof-checking editor differs from that of other systems. Rules of inference are embedded in the editor as an attribute grammar. This allows proof checking to be done in an incremental fashion, resulting in good response time. The editor has been implemented using the Synthesizer Generator, a system that creates editors from an attribute grammar description [Reps & Teitelbaum 1983].

This work was supported in part by the National Science Foundation under grants MCS80-04218, MCS81-03605, and MCS82-02677. Bowen Alpern is supported by an IBM Graduate Fellowship.

Authors' address: Department of Computer Science, Upson Hall, Cornell University, Ithaca, N.Y. 14853.

In its current form, the editor is just a prototype, suitable for demonstrating the principles on small examples, but not for proving sizable theorems or manipulating large programs. (To some extent this is due to the state of the Synthesizer Generator, which is still under development). Nonetheless, the editor represents a promising approach to interactive verification.

This paper discusses the use of attribute grammars for providing feedback about errors in a proof under development. While the attribute grammar described forms the basis for our proof-checking editor, this paper is not a discussion of that particular editor. Rather, the paper concerns the appropriateness of a particular data structure -- attributed derivation trees -- for representing proofs in proof-checking editors, regardless of what user interface is desired.

The paper is organized into 6 sections, as follows: Section 2 presents an example of a proof being modified, which motivates the formalization of interactive proof checking described in Section 3. Section 3.1, following [Gerhart 1975], shows how generating a program's verification conditions can be expressed with an attribute grammar; Section 3.2 shows how checking of predicate-logic proofs can be expressed with an attribute grammar. Section 4 discusses some enhancements to the basic approach aimed at making the user's task less tedious when proofs are created and modified. Section 5 discusses how the attribute-grammar approach differs from the approaches used in other interactive proof-development systems. Section 6 draws some conclusions about our experience with the attribute-grammar approach.

2. Interactive proof checking

The editor provides information about mistakes in a program proof by checking the program's statements against a formal specification of the program's behavior and indicating places where the code does not implement the specification. A program proof can be presented as a *proof outline* -- a program annotated with assertions. In the example given below, which is meant to suggest what one sees on the terminal screen during editing, program text in a proof outline that is inconsistent with the proof outline's assertions is highlighted in the program display. Of course, this mechanism could be replaced by, or used in conjunction with, diagnostic messages.

Example. Suppose we are trying to correct the following proof outline so that it will compute z to be the product of a and b, using repeated addition:

$\{b \geq 0\}$
$y := b$
$z := 0$
while $\boxed{y \geq 0}$ invariant $y \geq -1 \wedge a*y+z = a*b$ do
  $y := y - 1$
  $z := z + a$
  od
$\{z = a*b\}$

In this piece of code, the initialization of variables y and z establishes the loop-invariant on entering the loop, and the loop-body reestablishes the invariant on each succeeding iteration. However, the conjunction of the invariant and the negation of the loop-condition fails to establish the post-condition of the loop. This situation is signaled (above) by highlighting the loop-condition.

To fix the problem, the first clause of the invariant is changed to $y \geq 0$, but now the loop-body fails to reestablish the invariant, which is signaled (below) by highlighting the loop-body:

$\{b \geq 0\}$
$y := b$
$z := 0$
while $y \geq 0$ invariant $y \geq 0 \wedge a*y+z = a*b$ do
  $\boxed{y := y - 1}$
  $\boxed{z := z + a}$
  od
$\{z = a*b\}$

The problem is that the conjunction of the loop-condition and the invariant does not imply the weakest precondition of the invariant with respect to the loop-body; operationally, the loop-condition allows the loop to execute one too many times. By changing the loop-condition to $y > 0$, the loop-body will reestablish the invariant, and the invariant and the negation of the loop-condition will still establish the post-condition, so the entire program is displayed in the normal font:

$\{b \geq 0\}$
$y := b$
$z := 0$
while $y > 0$ invariant $y \geq 0 \wedge a*y+z = a*b$ do
  $y := y - 1$
  $z := z + a$
  od
$\{z = a*b\}$

This example illustrates that modifying one part of a proof outline may introduce an inconsistency in some other part of the proof outline and simultaneously correct an inconsistency in yet a third part of the proof outline. Thus, the editor must not only incorporate the notion of an inconsistent proof, but the notion of dependencies among parts of a proof, as well. The next section shows how these notions may be expressed using an attribute grammar.

3. Attribute grammars and formal logical systems

An attribute grammar is a context-free grammar extended by attaching attributes to the symbols of the grammar. Associated with each production of the grammar is a set of *semantic equations*; each equation defines one attribute as the value of a *semantic function* applied to other attributes in the production. Attributes are divided into two disjoint classes: *synthesized* attributes and *inherited* attributes. Each semantic equation defines a value for a synthesized attribute of the left-hand side nonterminal or an inherited attribute of a right-hand side symbol.

The axioms and inference rules of a formal logical system can be expressed as productions and semantic equations of an attribute grammar. Dependencies among attributes, as defined in the semantic equations of such a grammar, express dependencies among parts of a proof.

An attribute instance in an attributed derivation tree is said to be *consistent* if its value is equal to the value obtained by evaluating the right-hand side of its defining semantic equation. An attributed derivation tree is *consistently attributed* if all of its attribute instances are consistent.

A proof is represented as a consistently attributed derivation tree of the grammar. Proofs are modified by operations that restructure the derivation tree, such as pruning, grafting, and deriving. Restructuring a derivation tree directly affects the values of the attributes at the modification point; some of the attributes may no longer have consistent values. Thus, incremental proof checking can be performed by (incrementally) updating attribute values throughout the tree in response to modifications.

Fundamental to this approach is the idea of an *incremental attribute evaluator*, an algorithm to produce a consistently attributed tree after each restructuring operation. An incremental attribute evaluator works by following attribute-dependency relationships in the tree to reestablish consistent values. Several algorithms for this are given in [Reps 1982] and [Reps et al. 1983], including ones that are asymptotically optimal in time.

When proofs are developed in a top-down fashion, the editor must not only incorporate the normal rules of logic, but the notion of an incomplete proof, as well. Creating a proof top-down entails growing a derivation tree. During development, it is a partial derivation tree; that is, it contains unexpanded nonterminals. This is potentially a problem, because at an unexpanded nonterminal X we have no means for giving values to the synthesized attributes of X nor to any of the attributes that depend on them; this conflicts with our desire to maintain values for every attribute of the tree.

To avoid this problem, we provide a *completing production*, $X ::= \perp$, for each nonterminal symbol X. The symbol $\perp$ denotes "unexpanded," and the semantic equations of the completing production define values for the synthesized attributes of X. By convention, an occurrence of an unexpanded nonterminal is considered to have derived $\perp$. By this device, all partial derivation trees (from the user's viewpoint) are considered complete

derivation trees (from the editor's viewpoint), and as a proof is developed, its tree may be fully attributed at all stages.

To formalize the notion of inconsistent portions of a proof, we introduce *check expressions*. A check expression is a Boolean semantic function that indicates whether constraints of the formal system are satisfied. If an editing operation modifies the proof in such a way that constraints are violated, check expressions indicating satisfaction of constraints become false. These are then used to annotate the program display to provide the user with feedback about errors that exist in the proof. In the example given in the previous section, the font of the proof's print representation depends on the values of check expressions in the proof's derivation tree.

As an aside, note that our use of terms like "semantic equation" conforms with accepted attribute-grammar terminology, although strictly speaking such terms are misnomers. These terms are carryovers from the context in which attribute grammars were originally used, namely, defining the meaning of a context-free language, and they have remained standard even though attribute grammars are often used to describe the annotation of a tree with information that has nothing to do with its meaning. For example, correctness of formal proofs is a purely syntactic matter (to a logician), but because it cannot be expressed in a context-free formalism, we make use of the available non-context-free mechanism -- the "semantic" mechanism of an attribute grammar.

3.1. Generating verification conditions

Generation of a program's verification conditions can be expressed with an attribute grammar using two attributes: *pre* and *post*. *Pre* is a synthesized attribute of Stmt and StmtList whose value is a formula in the language of assertions; *post* is an inherited attribute of Stmt and StmtList whose value is also a formula in the language of assertions. The relationships among these attributes that express partial correctness of programs are given by the rules of the grammar presented in Figure 1, which is adapted from one given in [Gerhart 1975]. (For brevity, we have not shown the productions that can be derived from Id, Exp, Cond, and Assertion, nor have we shown the completing productions of the grammar).

In the semantic equations in Figure 1, as well as throughout the rest of the paper, we use conventionally accepted notation to express set operations, we use "." as the operator for selecting an attribute of a nonterminal, and we use subscripts to distinguish among multiple instances of the same nonterminal. We also make use of the nonstandard notion of a *syntactic reference*. Insofar as a component of the syntax tree is often *itself* a sufficient representation of the value to be associated with the component's root, we allow a nonterminal's name (when not qualified by an attribute selection) to denote the syntactic component as a value. For example, in production (1) of Figure 1,

(1) Program ::= {Assertion} StmtList {Assertion}
StmtList.post = $Assertion_2$
check: *IsTheorem* ($Assertion_1 \supset StmtList.pre$)

$Assertion_1$ and $Assertion_2$ refer to the components derived from the rule's two Assertion nonterminals. For further discussion of syntactic references in attribute-grammar specifications, the reader is referred to [Reps & Teitelbaum 1983].

---

(1) Program ::= {Assertion} StmtList {Assertion}
StmtList.post = $Assertion_2$
check: *IsTheorem* ($Assertion_1 \supset StmtList.pre$)
(2) StmtList ::= Stmt
Stmt.post = *StmtList.post*
StmtList.pre = *Stmt.pre*
(3) StmtList ::= Stmt StmtList
$StmtList_2.post$ = $StmtList_1.post$
Stmt.post = $StmtList_2.pre$
$StmtList_1.pre$ = *Stmt.pre*
(4) Stmt ::= Id := Exp
$Stmt.pre = Stmt.post^{Id}_{Exp}$
(5) Stmt ::= if Cond then StmtList else StmtList fi
$StmtList_1.post$ = *Stmt.post*
$StmtList_2.post$ = *Stmt.post*
Stmt.pre = ($Cond \supset StmtList_1.pre$)
$\wedge$ ($\neg Cond \supset StmtList_2.pre$)
(6) Stmt ::= while Cond invariant Assertion
do StmtList od
Stmt.pre = *Assertion*
StmtList.post = *Assertion*
check: *IsTheorem* (($Assertion \wedge \neg Cond) \supset Stmt.post$)
check: *IsTheorem* (($Assertion \wedge Cond) \supset StmtList.pre$)

Figure 1: Generating verification conditions.

The semantic equations of the grammar treat statements as backward predicate transformers [Dijkstra 1976]. In an assignment statement, for example, the relationship between the *pre* and *post* attribute is that the *pre* attribute is defined as the *post* attribute with the expression on the right-hand side of the assignment substituted for all occurrences of the left-hand side identifier. (In Figure 1 this is denoted by the expression $Stmt.post^{Id}_{Exp}$).

For while-loops, the post-condition of the loop-body and the pre-condition of the parent statement are defined in terms of the loop-invariant. This allows inconsistent code and assertions to be detected as violations of the check-expressions:

check: *IsTheorem* (($Assertion \wedge \neg Cond) \supset Stmt.post$)
check: *IsTheorem* (($Assertion \wedge Cond) \supset StmtList.pre$)

where the function IsTheorem is a decision procedure -- a procedure that returns true if its argument is a theorem in the assertion-language logic.

It is at this point that we reach one of the common stumbling blocks of verification systems -- the decidability of an assertion language strong enough to express the

verification conditions constructed by the rules given above. For instance, no decision procedure exists for first-order predicate logic [Turing 1937].

In the next section, we sidestep this problem by having the user create and manipulate the required proofs, instead of having the system try to establish theorems automatically. Further on, in Section 4.1, we discuss how to incorporate decision procedures for subtheories of predicate logic, so that the user need not prove theorems that automatic techniques *are* capable of establishing.

By having the user create and manipulate proofs of verification conditions, we make the user responsible for proving theorems that an automatic theorem prover might be incapable of proving. Grammar rules in Figure 1 that have a check expression involving IsTheorem are changed to include a "Proof" nonterminal for each (predicate-calculus) proof obligation of the production. Thus, the new rule for while loops has two Proof nonterminals, one at which the user must create a proof of

$$(Assertion \wedge \neg Cond) \supset Stmt.post$$

and a second at which the user must create a proof of

$$(Assertion \wedge Cond) \supset StmtList.pre.$$

3.2. Checking proofs of verification conditions

Because the editor is a tree-manipulating system, we need a formalization of the assertion-language logic that allows proofs of verification conditions to be conveniently represented as tree-structured objects. If the assertion language is a predicate logic, a suitable formalization is Gentzen's *sequent calculus* [Gentzen 1935, Kleene 1952].

A *sequent* consists of two sets of formulae, separated by an arrow, such as:

$$\{A_1, A_2, \ldots, A_m\} \rightarrow \{B_1, B_2, \ldots, B_n\} \qquad (1)$$

The set $\{A_1, A_2, \ldots, A_m\}$, on the left, is called the *antecedent*; the set $\{B_1, B_2, \ldots, B_n\}$, on the right, is called the *succedent*. A sequent is a theorem in the sequent calculus if it can be derived from the system's axioms and rules of inference.

It can be shown that a sequent is a theorem in the sequent calculus if and only if, in one of the more familiar forms of the predicate calculus, a formula in the succedent can be demonstrated taking the formulae in the antecedent as assumptions [Gentzen 1935]. Informally then, one can think of the formulae of the antecedent as known facts and the formulae of the succedent as goals, one of which is to be demonstrated; thus, the informal meaning of the sequent (1) is no different from asserting the formula:

$$A_1 \wedge A_2 \wedge \cdots \wedge A_m \supset B_1 \vee B_2 \vee \cdots \vee B_n$$

The inference rules of sequent calculus allow us to infer new sequents from old sequents. For each logical operator there are two inference rules: an *analysis* rule and a *synthesis* rule. The analysis rule for a (logical) operator $\otimes$ expresses how a formula of the form $A \otimes B$ may be introduced into an antecedent; the synthesis rule for $\otimes$ expresses how $A \otimes B$ may be introduced into a succedent.

For example, using the meta-variables A and B to represent single formulae and the meta-variables Γ and Δ to represent finite sets of formulae, the rules for the implication operator $\supset$ are expressed as:

$$\text{Implication analysis} \quad \frac{\Gamma \rightarrow \Delta \cup \{A\} \qquad \Gamma \cup \{B\} \rightarrow \Delta}{\Gamma \cup \{A \supset B\} \rightarrow \Delta} \qquad (2a)$$

$$\text{Implication synthesis} \quad \frac{\Gamma \cup \{A\} \rightarrow \Delta \cup \{B\}}{\Gamma \rightarrow \Delta \cup \{A \supset B\}} \qquad (2b)$$

The implication analysis rule given as (2a) says (roughly) that if we want to assume $A \supset B$, we must demonstrate A, and we must demonstrate our goal assuming B. The implication synthesis rule given as (2b) says (again roughly) that if we want to demonstrate $A \supset B$, we must show that by assuming A we can demonstrate B.

An attribute grammar can be used to express the rules of sequent calculus as follows. A sequent is represented by a Proof nonterminal that has two inherited attributes: an *antecedent* attribute and an *succedent* attribute. Each production of the grammar represents a rule of inference or an axiom scheme (see below). The right-hand sides of productions corresponding to inference rules contain additional Proof nonterminals whose *antecedent* and *succedent* attributes are defined in terms of the *antecedent* and *succedent* attributes of the parent Proof nonterminal. The check expressions of the sequent-calculus grammar express the constraint that a production derived from a Proof nonterminal represents an appropriate deductive step.

For example, the productions corresponding to the implication inference rules are shown in Figure 2.[1] In each production in Figure 2, there are two "Wff" nonterminals on the right-hand side that determine how an inference rule is instantiated. The subtrees derived from these Wff's determine the components of the formula being analyzed (or synthesized, as the case may be), as well as the antecedents and succedents of the right-hand side Proof nonterminals. The check expressions ensure that the Wff being analyzed (synthesized) really is in the left-hand side Proof nonterminal's *antecedent* (*succedent*) attribute.

[1] Productions for the rest of the rules of first-order predicate logic may be found in an appendix at the end of the paper.

/* Implication analysis */
Proof ::= show Wff Proof
assume Wff Proof
check: $(Wff_1 \supset Wff_2) \in Proof_1.antecedent$
$Proof_2.antecedent = Proof_1.antecedent - \{Wff_1 \supset Wff_2\}$
$Proof_2.succedent = Proof_1.succedent \bigcup \{Wff_1\}$
$Proof_3.antecedent = Proof_1.antecedent - \{Wff_1 \supset Wff_2\} \bigcup \{Wff_2\}$
$Proof_3.succedent = Proof_1.succedent$

/* Implication synthesis */
Proof ::= assume Wff show Wff Proof
check: $(Wff_1 \supset Wff_2) \in Proof_1.succedent$
$Proof_2.antecedent = Proof_1.antecedent \bigcup \{Wff_1\}$
$Proof_2.succedent = Proof_1.succedent - \{Wff_1 \supset Wff_2\} \bigcup \{Wff_2\}$

Figure 2: Grammar rules corresponding to the implication inference rules.

The axioms of the sequent calculus are expressed in three schemes:

$$\frac{}{\Gamma \bigcup \{A\} \rightarrow \Delta \bigcup \{A\}} \quad (3a)$$

$$\frac{}{\Gamma \bigcup \{\text{false}\} \rightarrow \Delta} \quad (3b)$$

$$\frac{}{\Gamma \rightarrow \Delta \bigcup \{\text{true}\}} \quad (3c)$$

The scheme given as (3a) says that if formula A is given, then A is demonstrated; (3b) says that if you start with false, then anything can be demonstrated; (3c) says that everything demonstrates true. These three axiom schemes can be combined into a single production whose check expression gives the condition under which application of an axiom completes a branch of the proof:

Proof ::= immediate
check: false $\in Proof.antecedent$
$\vee$ true $\in Proof.succedent$
$\vee$ $(Proof.antecedent \bigcap Proof.succedent \neq \phi)$

Finally, we need to define the atomic predicates of our logic; for instance, we need predicates for equality and less than:

Exp = Exp
Exp < Exp

Wff's are built out of predicates using logical connectives. The expressions (Exp) in the predicates can be any expression of the programming language; whatever is legal on the right-hand side of an assignment is a legal Exp.

To fully incorporate these predicates into our logical system we include their axiomatic definition. For example, the axioms for equality can be expressed as:

$$\frac{}{\Gamma \rightarrow \Delta \bigcup \{J=J\}} \quad (4a)$$

$$\frac{}{\Gamma \rightarrow \Delta \bigcup \{(J=K) \supset (K=J)\}} \quad (4b)$$

$$\frac{}{\Gamma \rightarrow \Delta \bigcup \{(J=K) \wedge (K=L) \supset (J=L)\}} \quad (4c)$$

where the meta-variables J, K, and L represent single expressions.

Each axiom adds an additional production to the grammar. For instance, the production corresponding to axiom (4a) is:

Proof ::= by reflexivity
check: *there exists* $s \in Proof.succedent$ *such that* s *is of the form* $J=J$

4. Enhancements to the basic approach to proof checking

Using the proof editor described in the previous section can be maddeningly tedious; the editor is a proof checker, not a theorem prover, and proofs must be complete formal proofs. This section discusses two ways of making the user's task less tedious when proofs are created and modified. First, we discuss how the editor can be extended to include some automatic deductive capabilities so that the user does not have to supply so much detail. Second, we show how pattern matching can be used within proof trees to facilitate editing.

4.1. Automatic deductive capabilities

In Section 3.2, we sidesteped the undecidability of predicate logic by having the user write proofs, instead of having the editor try to establish theorems automatically. In practice, this approach is untenable because it forces the user to provide absolutely every detail of a proof. Given an unexpanded Proof node that is a leaf of the tree, we can often check that its sequent can be proven using a decision procedure for a subtheory of predicate logic. Another possibility is to apply a *proof tactic* [Gordon et al. 1979], that does its best to construct a proof tree, but may leave some Proof nodes unexpanded for the user to fill in later.

4.1.1. Decision procedures

The editor can be extended with decision procedures by making use of known algorithms for deciding simple theories. For example, an algorithm for deciding the theory of equality with uninterpreted function symbols [Johnson 1981, Nelson 1981] can be used as the basis of a procedure for propositional inference. Our proof editor uses the propositional-inference procedure from the PL/CV2 proof-checking compiler [Constable et al. 1982]; it

is incorporated into the editor through the grammar rule:

Proof ::= automatic
  check: *IsAutomatic*(*Proof.antecedent*, *Proof.succedent*)

where IsAutomatic returns true if the PL/CV2 propositional-inference procedure can establish the second argument from the first argument.

The PL/CV2 automatic-inference procedure has certain limitations because it is a decision procedure for only a subtheory of propositional logic (a subtheory selected so that problems of inherent computational complexity do not have to be solved). Thus, the grammar production given above will not be applicable when the theorem to be established requires the use of a primitive inference rule that the automatic procedure never attempts to apply.

Example. Proving the sequent

$$\{a,\ b,\ d\} \rightarrow \{a \wedge (b \wedge (c \supset d))\} \qquad (5)$$

requires the use of the implication synthesis rule to establish the formula $c \supset d$, as in the starred branch of the following proof tree:

$$\dfrac{\{a,b,d\} \rightarrow \{a\} \qquad \dfrac{\{a,b,d\} \rightarrow \{b\} \qquad \dfrac{\{a,b,c,d\} \rightarrow \{d\}}{\{a,b,d\} \rightarrow \{c \supset d\}}\,(*)}{\{a,\ b,\ d\} \rightarrow \{b \wedge (c \supset d)\}}}{\{a,\ b,\ d\} \rightarrow \{a \wedge (b \wedge (c \supset d))\}}$$

Thus, a decision procedure that never applies implication synthesis cannot establish (5).

However, there is still a way to help cut down on tedious manipulations when automatic inference procedures are not applicable. Our editor incorporates a mechanism whereby the user can isolate the offending term and apply the appropriate inference rule explicitly. This mechanism is based on an additional rule of inference termed the *cut rule*, expressed as:

$$\text{Cut rule: } \frac{\Gamma \rightarrow \{A\} \qquad \Gamma \bigcup \{A\} \rightarrow \Delta}{\Gamma \rightarrow \Delta} \qquad (6)$$

The cut rule (6) says that if we want to prove some goal, we can demonstrate some formula A and then use A as an assumption in the proof of the goal. The cut rule allows a user to isolate a formula easily, because automatic inferences can be used to skip over the easy intermediate steps.

Example. Returning to the example above, if we choose $c \supset d$ as the cut formula A, the proof branches into two subproofs whose sequents are:

$$\{a,\ b,\ d\} \rightarrow \{c \supset d\} \qquad (7)$$

and

$$\{a,\ b,\ d,\ c \supset d\} \rightarrow \{a \wedge (b \wedge (c \supset d))\} \qquad (8)$$

We are then able to apply implication synthesis directly to (7), and the automatic inference rule can be used to establish (8) because a proof can be found that makes no use of implication synthesis:

$$\dfrac{\{a,b,d,c \supset d\} \rightarrow \{a\} \qquad \dfrac{\{a,b,d,c \supset d\} \rightarrow \{b\} \qquad \{a,b,d,c \supset d\} \rightarrow \{c \supset d\}}{\{a,\ b,\ d,\ c \supset d\} \rightarrow \{b \wedge (c \supset d)\}}}{\{a,\ b,\ d,\ c \supset d\} \rightarrow \{a \wedge (b \wedge (c \supset d))\}}$$

4.1.2. Proof tactics

A proof tactic is a method for applying inference rules repeatedly and recursively until none is applicable [Gordon et al. 1979]. In proof editors, a proof tactic may be employed to automatically construct a proof fragment, doing its best to construct a proof tree, but possibly leaving some unexpanded Proof nodes for the user to fill in later [Bates & Constable 1983].

Example. Given an unexpanded Proof nonterminal with the sequent given above as equation (5), a proof tactic could apply the and synthesis rule twice to produce the attributed derivation tree that corresponds to the inference:

$$\dfrac{\{a,b,d\} \rightarrow \{a\} \qquad \dfrac{\{a,b,d\} \rightarrow \{b\} \qquad \{a,b,d\} \rightarrow \{c \supset d\}}{\{a,\ b,\ d\} \rightarrow \{b \wedge (c \supset d)\}}}{\{a,\ b,\ d\} \rightarrow \{a \wedge (b \wedge (c \supset d))\}}$$

leaving an unexpanded Proof nonterminal with sequent:

$$\{a,\ b,\ d\} \rightarrow \{c \supset d\}$$

In the attribute-grammar framework, a proof tactic would require using inherited attributes to drive the tree construction process. As currently implemented, proof tactics cannot be incorporated into our editor, because attribute-driven tree construction is forbidden in the Synthesizer Generator; the Synthesizer Generator only allows attribution of a previously constructed abstract-syntax tree.

Attribute-driven tree construction has been explored for resolving ambiguities in attribute-grammar-based parsers [Watt 1977, Rowland 1977, Milton et al. 1979]. A somewhat different notion of "computing with attributed trees" is currently being studied for inclusion in the Synthesizer Generator [Reps & Teitelbaum 1983]; this mechanism appears to be powerful enough to express proof tactics.

4.2. A proof representation that uses pattern matching

The attribute grammar described in Section 3.2 for representing sequent-calculus proofs as attributed trees has a significant drawback: the representation often makes it tedious to modify previously developed proofs. This section describes a modification of the grammar that avoids this problem.

The grammar of Section 3.2 specifies that one or two Wff's be derived at each Proof node in order to (1) indicate which formula of the Proof's sequent is being analyzed or synthesized, and (2) determine the antecedents and succedents associated with the node's subordinate Proof nonterminals. Consider what happens

when we modify a proof, say by changing a variable name in an assumption from x to y. The proof now has errors at each Proof node where a Wff contains a use of x, because the Wff refers to a formula not in the Proof node's sequent. To reestablish a correct proof, it is necessary to (manually) change each of the Wff's that refer to x. The problem with the grammar of Section 3.2 is that the Wff nonterminals bind *too much information* into the proof representation by specifying too precisely how the inference rules are instantiated.

An alternative approach makes use of "WffPattern's" and pattern matching. A WffPattern is a partial Wff -- a Wff possibly containing unexpanded nonterminals -- and a WffPattern matches a Wff if the latter can be derived from the former. Instead of having Wff's at each Proof nonterminal, the grammar is changed to have WffPattern's, which are then used to determine the antecedents and succedents associated with the node's subordinate Proof nonterminals.

For example, Figure 3 gives the modified rule for implication analysis. In contrast to the rule with two Wff nonterminals given previously in Figure 2, the new rule has two WffPattern's. These WffPattern's give the components of a *pattern* for the formula being analyzed, which is selected out of the left-hand side Proof nonterminal's antecedent by a pattern-matching lookup. When there is more that one formula in the antecedent that matches the pattern, the user may have to specify a more detailed pattern. (Note that *premise* and *conclusion*, the components of the matched formula, appear in the context-free part of the rule; this is to suggest that the user would be given an indication in the display as to which formula in the antecedent the pattern matched).

/* Implication analysis */
Proof ::= analyse by (WffPattern⊃WffPattern)
show *premise* Proof
assume *conclusion* Proof
let *wff* = *FindMatch*((*WffPattern*$_1$⊃*WffPattern*$_2$), *Proof*$_1$.*antecedent*)
where premise and conclusion are the components of wff
Proof$_2$.*antecedent* = *Proof*$_1$.*antecedent* − {*wff*}
Proof$_2$.*succedent* = *Proof*$_1$.*succedent* ⋃ {*premise*}
Proof$_2$.*antecedent* = *Proof*$_1$.*antecedent* − {*wff*} ⋃ {*conclusion*}
Proof$_2$.*succedent* = *Proof*$_1$.*succedent*

Figure 3: Implication analysis rule that uses pattern matching.

The proof grammar that uses pattern matching in the semantic equations is a significant improvement over the old grammar. At most nodes of a proof tree, it is not necessary for the WffPattern's to contain variable names; in most cases, a structural pattern alone is sufficient to indicate which formula of the Proof's sequent is being analyzed or synthesized. Now if we change a variable name from x to y, the proof will still check as before, because all (structural) WffPattern's will still match a formula in the appropriate sequent, and the proper antecedent and succedent will be associated with each Proof nonterminal of the tree.

5. Comparison with alternative approaches

The attribute grammar discussed above formalizes an approach to interactive proof development that is different from what is done in other interactive verification systems. This section discusses some of the differences between our approach and the approaches used in other systems.

The Designer/Verifier's Assistant [Moriconi 1979] uses one alternative approach. There, the user interacts with the Assistant, which in turn decides what needs to be reverified based on an analysis of the dependencies among the procedures of a program. An important difference in the way proofs are treated by the Assistant and the way they are treated by our editor is the granularity and incrementality of proof checking in the two systems. In many respects, the Assistant is similar to compilation-control systems, such as Make [Feldman 1979]; the Assistant decides what to reverify on a per-procedure basis, and when a procedure is reverified it is reverified in its entirety. By contrast, our editor decides what to reverify on a per-inference-rule basis, and by virtue of the optimal behavior of the algorithm used for incremental attribute updating, reanalysis is confined to the attributes that actually need new values.

In the Edinburgh LCF system [Gordon et al. 1979], there is a notion of objects of type *thm*, but the objects of type *thm* are not proofs as such. *Thm* is an abstract data type whose constructor functions obey the invariant "all *thm* objects are provable," that is, derivable from the axioms by applications of inference rules. An attempt to use a constructor to make an inappropriate deductive step ends in failure. By contrast, our editor supports the development of actual proofs, which can be manipulated and restructured directly, and the editor incorporates the notion of a proof with errors and inconsistencies in it.

The approach taken in AVID is much closer to the approach taken in our editor. AVID is an editor for the top-down development of PL/CV2 proofs that incorporates a proof checker to provide information about a proof's errors and inconsistencies [Krafft 1981, Constable et al. 1982]. An important difference between our editor and AVID is that AVID is without a notion of logical dependencies analogous to the attribute dependencies in our editor's attribute grammar. Lacking the information needed to re-use previous verification information as a proof is checked, AVID carries out verification only when there is an explicit request by the user, and it always reverifies a proof in its entirety.

The closest relative of our editor is the proof-checking editor that is part of the PRL system [Bates & Constable 1983]. PRL provides machine aid in the crea-

tion of definitions, functions, and proofs based on a sequent calculus for constructive first-order predicate logic over integers and lists. One difference between our proof editor and the one used in PRL is the way the two editors handle inconsistencies in a proof. The PRL editor forbids inconsistencies by requiring each interior node of the proof tree to be an appropriate deductive step. One is able to go back and modify an interior node of the proof, but if this would introduce an inconsistency into a subproof, the subproof is deleted. By contrast, proofs constructed with our editor are allowed to have inconsistencies in them as they are developed; our editor uses knowledge about dependencies among parts of a proof (encoded in attribute dependencies) to keep track of such inconsistencies.

Our proof editor and the PRL editor also differ in the way they implement proof checking. The PRL approach may be characterized as the *semantic-action* approach; during editing, each operation that affects a node of type X invokes an action associated with the category X. An action is an imperative routine that can walk the program tree making updates to nodes of the tree as well as to global data structures.

Our approach to proof checking relies on two attractive properties of attribute grammars:

(1) Attribute grammars are declarative statements of relations that must hold among the parts of a proof; propagation of context-dependent information through the syntax tree need not be described explicitly, as it is implicit in the formalism.

(2) Attribute grammars allow automatic reestablishment of consistent attribute values when a proof is modified, without the need for explicit undoing or rollback actions; furthermore, such updating can be performed in an asymptotically optimal manner.

For further discussion of the relative merits of the semantic-action and the attribute-grammar approaches, the reader is referred to [Reps et al. 1983].

6. Summary and conclusions

Our concern is the design of editors that allow one to create and modify program proofs in Hoare-style logic. We have constructed an editor that treats a proof as an object with constraints on it; the editor keeps track of inconsistencies in a proof by reexamining the proof's constraints after each modification to it. The logical system is encoded in the editor as an attribute grammar.

We feel that the attribute-grammar approach to interactive proof checking is a promising one on a number of counts. Attribute grammars permit the specification of the constraints of a formal logical system, as described in Section 3. Attribute grammars are a good framework for incorporating previously developed solutions to verification problems, such as fast decision procedure for subtheories of predicate logic, as described in Section 4.1. Furthermore, there exist optimal algorithms for incremental attribute updating, which means proof checking can be done in an incremental and optimal fashion. Finally, there exist compiler-compilers and editor generators that produce major software components from an attribute grammar description of a language; this makes it particularly easy to implement systems based on the ideas discussed in this paper.

Acknowledgements

We were stimulated to write this discussion of our work on interactive proof checking after receiving encouragement from a number of people who had seen the prototype proof-checking editor; discussions with Rod Burstall, Bob Constable, Alan Demers, Edsger Dijkstra, Gerard Huet, Gilles Kahn, Dave McQueen, and Tim Teitelbaum were particularly interesting. We would also like to thank our colleagues who read the paper and commented on it; the suggestions of Bob Constable, Bob Harper, Susan Horwitz, Mark Krentel, Fred Schneider, and Tim Teitelbaum have been extremely helpful.

References

[Bates & Constable 1983]
Bates, J. and Constable, R. Proofs as programs. Tech. Rep. 82-530, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Feb. 1983.

[Bricklin & Frankston 1979]
Bricklin, D. and Frankston, B. *VisiCalc Computer Software Program for the Apple II and II Plus.* Personal Software, Inc., Sunnyvale, Calif., 1979.

[Constable et al. 1982]
Constable, R., Johnson, S., and Eichenlaub, C. *Lecture Notes in Computer Science*, vol. 135: *Introduction to the PL/CV2 Programming Logic.* Springer-Verlag, New York, 1982.

[Dijkstra 1976]
Dijkstra, E.W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[Feldman 1979]
Feldman, S.I. Make -- A program for maintaining computer programs. *Software -- Practice and Experience 9*, 4 (April 1979), 255-265.

[Gentzen 1935]
Gentzen, G. Investigations into logical deductions. In *The Collected Papers of Gerhard Gentzen*, M.E. Szabo (ed.), North-Holland, Amsterdam, 1969, pp. 68-131.

[Gerhart 1975]
Gerhart, S.L. Correctness-preserving program transformations. In Conference Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif., Jan. 20-22, 1975, pp. 54-66.

[Gordon et al. 1979]
Gordon, M., Milner, R., and Wadsworth, C. *Lecture Notes in Computer Science*, vol. 78: *Edinburgh LCF.* Springer-Verlag, New York, 1979.

[Hoare 1969]
Hoare, C.A.R. An axiomatic basis for computer

programming. *Commun. ACM 12*, 10 (Oct. 1969), 576-580, 583.

[Johnson 1981]
Johnson, S. A computer system for checking proofs. Tech. Rep. 80-444 and Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Jan. 1981.

[Kleene 1952]
Kleene, S.C. *Introduction to Metamathematics*. North-Holland, Amsterdam, 1952.

[Knuth 1968]
Knuth, D.E. Semantics of context-free languages. *Math. Syst. Theory 2*, 2 (June 1968), 127-145.

[Krafft 1981]
Krafft, D. AVID: A system for the interactive development of verifiably correct programs. Tech. Rep. 81-467 and Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1981.

[Milton et al. 1979]
Milton, D.R., Kirchhoff, L.W., and Rowland, B.R. An ALL(1) compiler generator. In Proceedings of the SIGPLAN Symposium on Compiler Construction, Denver, Colo., Aug. 6-10, 1979, *SIGPLAN Notices 14*, 8 (Aug. 1979), 152-157.

[Moriconi 1979]
Moriconi, M. A designer/verifier's assistant. *IEEE Trans. Softw. Eng. SE-5*, 4 (July 1979), 387-401.

[Nelson 1981]
Nelson, G. Techniques for program verification. Tech. Rep. CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Calif., June 1981.

[Reps 1982]
Reps, T. Generating language-based environments. Tech. Rep. 82-514 and Ph.D. dissertation, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Aug. 1982. To be published by M.I.T. Press, Cambridge, Mass., Feb. 1984.

[Reps & Teitelbaum 1983]
Reps, T., and Teitelbaum, T. The Synthesizer Generator. Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., Oct. 1983.

[Reps et al. 1983]
Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Trans. Program. Lang. Syst. 5*, 3 (July 1983), 449-477.

[Rowland 1977]
Rowland, B.R. Combining parsing and evaluation for attributed grammars. Tech. Rep. 308 and Ph.D. dissertation, Dept. of Computer Science, Univ. of Wisconsin, Madison, Wisc., Nov. 1977.

[Turing 1937]
Turing, A.M. On computable numbers with an application to the Entscheidungsproblem. *Proc. London Math. Soc., ser. 2, 42* (1936-7), 230-265.

[Watt 1977]
Watt, D.A. The parsing problem for affix grammars. *Acta Informatica 8* (1977), 1-20.

Appendix: Grammar rules for first-order predicate logic

/* Axiom schemes: $\dfrac{}{\Gamma \cup \{A\} \rightarrow \Delta \cup \{A\}}$

$$\frac{}{\Gamma \cup \{\text{false}\} \rightarrow \Delta}$$

$$\frac{}{\Gamma \rightarrow \Delta \cup \{\text{true}\}}$$ */

Proof ::= immediate
check: false $\in$ *Proof.antecedent*
V true $\in$ *Proof.succedent*
V ($Proof.antecedent \cap Proof.succedent \neq \phi$)

/* Implication analysis: $\dfrac{\Gamma \rightarrow \Delta \cup \{A\} \quad \Gamma \cup \{B\} \rightarrow \Delta}{\Gamma \cup \{A \supset B\} \rightarrow \Delta}$ */

Proof ::= show Wff Proof assume Wff Proof
check: ($Wff_1 \supset Wff_2$) $\in$ $Proof_1.antecedent$
$Proof_2.antecedent = Proof_1.antecedent - \{Wff_1 \supset Wff_2\}$
$Proof_2.succedent = Proof_1.succedent \cup \{Wff_1\}$
$Proof_3.antecedent = Proof_1.antecedent - \{Wff_1 \supset Wff_2\} \cup \{Wff_2\}$
$Proof_3.succedent = Proof_1.succedent$

/* Implication synthesis: $\dfrac{\Gamma \cup \{A\} \rightarrow \Delta \cup \{B\}}{\Gamma \rightarrow \Delta \cup \{A \supset B\}}$ */

Proof ::= assume Wff show Wff Proof
check: ($Wff_1 \supset Wff_2$) $\in$ $Proof_1.succedent$
$Proof_2.antecedent = Proof_1.antecedent \cup \{Wff_1\}$
$Proof_2.succedent = Proof_1.succedent - \{Wff_1 \supset Wff_2\} \cup \{Wff_2\}$

/* And analysis: $\dfrac{\Gamma \cup \{A, B\} \rightarrow \Delta}{\Gamma \cup \{A \wedge B\} \rightarrow \Delta}$ */

Proof ::= assume Wff and Wff Proof
check: ($Wff_1 \wedge Wff_2$) $\in$ $Proof_1.antecedent$
$Proof_2.antecedent = Proof_1.antecedent - \{Wff_1 \wedge Wff_2\} \cup \{Wff_1, Wff_2\}$
$Proof_3.succedent = Proof_1.succedent$

/* And synthesis: $\dfrac{\Gamma \to \Delta \cup \{A\} \quad \Gamma \to \Delta \cup \{B\}}{\Gamma \to \Delta \cup \{A \wedge B\}}$ */

Proof ::= show Wff Proof show Wff Proof
  check: $(Wff_1 \wedge Wff_2) \in Proof_1.succedent$
  $Proof_2.antecedent = Proof_1.antecedent$
  $Proof_2.succedent = Proof_1.succedent - \{Wff_1 \wedge Wff_2\} \cup \{Wff_1\}$
  $Proof_3.antecedent = Proof_1.antecedent$
  $Proof_3.succedent = Proof_1.succedent - \{Wff_1 \wedge Wff_2\} \cup \{Wff_2\}$

/* Or analysis: $\dfrac{\Gamma \cup \{A\} \to \Delta \quad \Gamma \cup \{B\} \to \Delta}{\Gamma \cup \{A \vee B\} \to \Delta}$ */

Proof ::= assume Wff Proof assume Wff Proof
  check: $(Wff_1 \vee Wff_2) \in Proof_1.antecedent$
  $Proof_2.antecedent = Proof_1.antecedent - \{Wff_1 \vee Wff_2\} \cup \{Wff_1\}$
  $Proof_2.succedent = Proof_1.succedent$
  $Proof_3.antecedent = Proof_1.antecedent - \{Wff_1 \vee Wff_2\} \cup \{Wff_2\}$
  $Proof_3.succedent = Proof_1.succedent$

/* Or synthesis: $\dfrac{\Gamma \to \Delta \cup \{A, B\}}{\Gamma \to \Delta \cup \{A \vee B\}}$ */

Proof ::= show Wff or Wff Proof
  check: $(Wff_1 \vee Wff_2) \in Proof_1.succedent$
  $Proof_2.antecedent = Proof_1.antecedent$
  $Proof_2.succedent = Proof_1.succedent - \{Wff_1 \vee Wff_2\} \cup \{Wff_1, Wff_2\}$

/* Not analysis: $\dfrac{\Gamma \to \Delta \cup \{A\}}{\Gamma \cup \{\neg A\} \to \Delta}$ */

Proof ::= show Wff Proof
  check: $(\neg Wff) \in Proof_1.antecedent$
  $Proof_2.antecedent = Proof_1.antecedent - \{\neg Wff\}$
  $Proof_2.succedent = Proof_1.succedent \cup \{Wff\}$

/* Not synthesis: $\dfrac{\Gamma \cup \{A\} \to \Delta}{\Gamma \to \Delta \cup \{\neg A\}}$ */

Proof ::= assume Wff Proof
  check: $(\neg Wff) \in Proof_1.succedent$
  $Proof_2.antecedent = Proof_1.antecedent \cup \{Wff\}$
  $Proof_2.succedent = Proof_1.succedent - \{\neg Wff\}$

/* All analysis: $\dfrac{\Gamma \cup \{A(t)\} \cup \{\forall s.A(s)\} \to \Delta}{\Gamma \cup \{\forall s.A(s)\} \to \Delta}$

where t is a term free for s in A(s) */

Proof ::= assume Wff substituting Exp for Id Proof
  check: $(\forall Id.Wff) \in Proof_1.antecedent$
  $Proof_2.antecedent = Proof_1.antecedent \cup \{Wff^{Id}_{Exp}\}$
  $Proof_2.succedent = Proof_1.succedent$

/* All synthesis: $\dfrac{\Gamma \to \Delta \cup \{A(b)\}}{\Gamma \to \Delta \cup \{\forall s.A(s)\}}$

where b is a variable not occurring free in A(s) */

Proof ::= show Wff substituting Id for Id Proof
  check: $(\forall Id_2.Wff) \in Proof_1.succedent$
  check: $NotFree(Id_1, Wff)$
  $Proof_2.antecedent = Proof_1.antecedent$
  $Proof_2.succedent = Proof_1.succedent - \{\forall Id_2.Wff\} \cup \{Wff^{Id_1}_{Id_2}\}$

/* Exist analysis: $\dfrac{\Gamma \cup \{A(b)\} \to \Delta}{\Gamma \cup \{\exists s.A(s)\} \to \Delta}$

where b is a variable not occurring free in A(s) */

Proof ::= assume Wff substituting Id for Id Proof
  check: $(\exists Id_2.Wff) \in Proof_1.antecedent$
  check: $NotFree(Id_1, Wff)$
  $Proof_2.antecedent = Proof_1.antecedent - \{\exists Id_2.Wff\} \cup \{Wff^{Id_1}_{Id_2}\}$
  $Proof_2.succedent = Proof_1.succedent$

/* Exist synthesis: $\dfrac{\Gamma \to \Delta \cup \{A(t)\}}{\Gamma \to \Delta \cup \{\exists s.A(s)\}}$

where t is a term free for s in A(s) */

Proof ::= show Wff substituting Exp for Id Proof
  check: $(\exists Id.Wff) \in Proof_1.succedent$
  $Proof_2.antecedent = Proof_1.antecedent$
  $Proof_2.succedent = Proof_1.succedent - \{\exists Id.Wff\} \cup \{Wff^{Id}_{Exp}\}$